Object-oriented programming - lab in .NET environment

Lecture 08

Themes

- Templates
- Binding
- MVVM

Control templates

- Most WindowsForms controls are wrappers around the Win32 API which is immutable
- WPF controls are written entirely in .NET
  - We can change their appearance as desired
- **Control templates** allow you to change the default appearance of a WPF control
  - Every WPF control has a Template property that is type of ControlTemplate
  - By setting the property to a new value, we change a look of control
    - Behavior remains unchanged!

Control templates

- Each WPF control has its own implicit template that defines which visual elements the control consists of
- To change the template of an element, we usually proceed as follows:
  1. We define a template as a resource type of `ControlTemplate`
  2. We apply the template to the desired elements
- It is possible to apply a template via a style by defining a style that sets the Template property to the new template

Template elements

- A template definition usually consists of:
  - The target element for which it is defined (property `TargetType`)
  - Consisting elements (visual tree)
  - Triggers that define dynamics

Template elements

- We often name visual tree elements so that we can access them from triggers
  - We name the element using attribute **`x:Name`**
  - We access the element using attribute **`TargetName`**
- The control for which we are creating a template has a set of properties and their values defined
- If we want to set individual properties within the template on the control to which the template is applied
  - We use the XAML extender **`TemplateBinding`** and its property **`Property`** to take the value from the control

```
<ContentPresenter
    HorizontalAlignment="{TemplateBinding Property=HorizontalAlignment}"/>
```

Binding

- Binding is a relationship that tells WPF to take data from the source object and set it as the value of the dependent property of the target object
  - The source object can be anything: a WPF element, DataRow, an instance of the class, …
  - The target property is always a dependent property
- We will go through two types of connections:
  - The source object is some WPF element
    - The property from which we take data is then the dependent property
  - The source object is any .NET object

Defining bindings through XAML

- We usually define the binding through XAML
- The term binding expression defines where and how we take data
  - We define it in XAML expander **`Binding`**
  - We reference the source element with a property **`ElementName`**
  - We reference the source property with the property **`Path`**
  - We define the connection direction with a property **`Mode`**:
    - `OneWay, OneWayToSource, TwoWay, OneTime`
- In case of a problem with the binding defined, WPF will not throw an exception
  - Details about the errors can be found in the debug window

Defining bindings through code

- Class `Binding` represents the connection object:
  - We reference the source element with a property **`Source`**
  - We reference the source property with the property **`Path`** and we set it to an instance type of **`PropertyPath`**
  - We define the connection direction with a property **`Mode`** and we set it to the enumeration **`BindingMode`** value
- When we have prepared the object, we call the method on the target element **`SetBinding()`**

Connecting to elements

- The simplest form of binding is binding to WPF elements
- For example, let's connect `TextBox` with `Slider`:

```
<Slider x:Name="slider"
        Minimum="1" Maximum="100"
        Value="50" TickFrequency="10" Margin="5"
        TickPlacement="BottomRight" />
<TextBox Margin="5"
         Text="{Binding ElementName=slider, Path=Value}" />
```

- By changing the `Slider`, the content of the `TextBox` is updated
- But, also by changing the value of the `TextBox`, the `Slider` is being updated
  - The default mode is `TwoWay`
  - Only after losing focus (property `UpdateSourceTrigger`)

An example of connecting to elements

- Let's define vertical and horizontal `Slider` which allows the Ellipse to move
- The Ellipse should be able to move around the entire parent container

```
<DockPanel LastChildFill="True">
    <Slider x:Name="h"
            DockPanel.Dock="Top"
            Minimum="0"
            Maximum="{Binding ElementName=canvas, Path=Width}"
            Value="0" />
    <Slider x:Name="v"
            DockPanel.Dock="Left"
            Minimum="0"
            Maximum="{Binding ElementName=canvas, Path=Height}"
            Value="0"
            Orientation="Vertical"/>

    <Canvas x:Name="canvas" Width="400" Height="200"
            HorizontalAlignment="Left"
            VerticalAlignment="Top">
        <Ellipse x:Name="ell"
                 Canvas.Left="{Binding ElementName=h, Path=Value}"
                 Canvas.Top="{Binding ElementName=v, Path=Value}"
                 Fill="Purple"
                 Width="100"
                 Height="100"/>
    </Canvas>
</DockPanel>
```

Binding to .NET object property data

- In order for changes to the property values of a .NET object to manifest on the user interface (UI), it needs to be used for binding **`MVVM`** (model-view-viewmodel) form:
  - The data encapsulates view model (**`VM`**) which implements **`INotifyPropertyChanged`** interface
  - **`VM`** is bound to the parent using a property **`DataContext`**
  - Controls reference properties viewable model by binding
    - Only the property **`Path`** is used because **`DataContext`** is a data source

MVVM – Model-View-ViewModel

- **Presentational pattern**
  - Fixes tight connection between Model and View in MVC pattern
    - The data for the presentation are presented as **ViewModel** entity that encapsulates all the data needed for the presentation
    - It can also connect several entities
- **Observable pattern**
  - WPF allows implementation of INotifyPropertyChanged interface that allows the connected element to receive notifications of changes and is updated accordingly