WPF

Events & Commands

Krzysztof Mossakowski
Faculty of Mathematics and Information Science

Events

- Event handler can be attached either using the XAML markup or C# code

```
<Image Name="FionaImage"
       Margin="5"
       Source="Images/Fiona_67x77.gif"
       MouseEnter="FionaImage_MouseEnter"
       MouseLeave="FionaImage_MouseLeave"/>
```

```
FionaImage.MouseEnter += new MouseEventHandler(FionaImage_MouseEnter);
FionaImage.MouseLeave += FionaImage_MouseLeave;
```

Events Routing

- There are three types of events differ by a way of routing them between controls and their containers:
  - Direct events originate in one element and don't pass to any other
    - examples: **MouseEnter**, **MouseLeave**
  - Bubbling events travel up the containment hierarchy
    - example: **MouseDown** – it is raised first by the element that is clicked, next it's raised by that element's parent and so on
  - Tunneling events travel down the containment hierarchy
    - example: **PreviewKeyDown** allows to intercept a key press first at the window level
- WPF usually defines bubbling and tunneling events in pairs, e.g. for the **MouseUp** bubbling event there exists the **PreviewMouseUp** tunneling event (all tunneling events are named using the 'Preview' word)
  - Events in the pair share the same instance of the **RoutedEventArgs** class, so if the tunneling event is marked as handled, the bubbling event will not occur

The RoutedEventArgs Class

- All WPF event argument classes inherit from **RoutedEventArgs**
- There are four properties of the **RoutedEventArgs** class:
  - **Source** – the object that raised the event
  - **OriginalSource** – the object that originally raised the event
    - usually the same as the Source object, but in some cases it goes deeper in the object tree
  - **RoutedEvent** – a special object for the event triggered by the event handler
  - **Handled** – when set to true, the event does not travel any further the containment hierarchy

Keyboard Input

- The following keyboard events are available for all elements:
  - **PreviewKeyDown, KeyDown, PreviewTextInput, TextInput, PreviewKeyUp, KeyUp**
- Focus
  - A control is able to accept the focus, only when its **Focusable** property is set to true
  - The **TabIndex** property can be used to order controls from the point of view of the focus
- The **KeyboardDevice** property of the **KeyboardEventArgs** class allows to check the state of keyboard modifiers keys (Shift, Ctrl, etc.) in the moment of raising the event
- The current state of the keyboard can be discovered using the **Keyboard** class and its static properties and methods

Mouse Input

- The following mouse events are available for all elements:
  - **MouseEnter, MouseLeave**
  - **MouseLeftButtonDown, MouseLeftButtonUp, MouseRightButtonDown, MouseRightButtonUp**, and their corresponding **Preview...** tunneling events
- The **Control** class adds the following events:
  - **MouseDoubleClick, PreviewMouseDoubleClick**
- The **Mouse.Capture** method allows for an element to receive mouse events even if they occur after the mouse has moved off the element
- The **DragDrop.DoDragDrop** method and the element's: **AllowDrop** property and **Drop** event allow to implement custom drag and drop operation between elements

Commanding

- WPF allows you to define tasks known as command and connect controls them so there is no need to write repetitive event handling code
  - The command feature manages the state of your user interface by automatically disabling controls when the linked commands aren't available
- The WPF command model consists of the following parts:
  - The command is the action to be executed
  - The command source is the object which invokes the command
  - The command target is the object that the command is being executed on
  - The command binding is the object which maps the command logic to the command

Commands

- Commands in WPF are created by implementing the **ICommand** interface
  - This interface contains two methods: **Execute** and **CanExecute**, and an event **CanExecuteChanged**
- The **RoutedCommand** class is the WPF implementation of **ICommand** with support for bubbling and tunneling
- WPF supplies a set of common routed commands spread across 5 classes: **MediaCommands**, **ApplicationCommands**, **NavigationCommands**, **ComponentCommands**, and **EditingCommands**
- Custom commands can be easily created by creating objects of the **RoutedCommand** or **RoutedUICommand** classes
  - It is also possible to create a custom class implementing the **ICommand** interface

Command Sources

- A command source is the object which invokes the command
  - Examples of command sources are **MenuItem**, **Button**, and **KeyGesture**
- Command sources in WPF generally implement the **ICommandSource** interface
  - This interface exposes 3 properties: **Command**, **CommandTarget**, and **CommandParameter**
  - The following classes implement **ICommandSource**: **ButtonBase**, **MenuItem**, **Hyperlink**, and **InputBinding**
    - Two types of input gestures in WPF are the **KeyGesture** and **MouseGesture**
- Typically, a command source will listen to the **CanExecuteChanged** event.

CommandBinding

- A **CommandBinding** associates a command with the event handlers that implement the command
  - It contains a **Command** property, and **PreviewExecuted**, **Executed**, **PreviewCanExecute**, and **CanExecute** events
- A **CommandBinding** is attached to a specific object, such as the root **Window** of the application or a control
- In some situations the **CommandBinding** is attached to the command target itself, such as with the **TextBox** class and the Cut, Copy, and Paste commands
- In most cases it is more convenient to attach the **CommandBinding** to an ancestor of the command target, especially if the same **CommandBinding** can be used for multiple command targets

```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Stop"
                  Executed="Stop_Executed"
                  CanExecute="Stop_CanExecute"/>
</Window.CommandBindings>
<Window.InputBindings>
  <KeyBinding Key="S" Modifiers="Control"
              Command="ApplicationCommands.Stop"/>
</Window.InputBindings>
<StackPanel>
  <Menu>
    <MenuItem Name="stopMenuItem" Header="Stop menu item"
              Command="ApplicationCommands.Stop" />
  </Menu>
  <Button Name="stopButton"
          Command="ApplicationCommands.Stop">Stop button</Button>
  <CheckBox Name="allowStopCheckBox">Allow stop</CheckBox>
</StackPanel>
```

```
private void Stop_Executed(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Stop executed!");
}
private void Stop_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = (allowStopCheckBox != null) ?
        allowStopCheckBox.IsChecked.Value : false;
}
```

Command Target

- The command target is the element on which the command is executed
  - In WPF the **CommandTarget** property on **ICommandSource** is only applicable when the **ICommand** is a **RoutedCommand**
- The command source can explicitly set the command target
- If the command target is not defined, the element with keyboard focus will be used as the command target
  - For example, it allows to work the Paste menu item with currently being edited **TextBox**

WPF

Data

Resources

- Every framework-level element (**FrameworkElement** or **FrameworkContentElement**) has a **Resources** property
  - This property contains the resources (as a **ResourceDictionary**) that a resource defines
  - Resources can be defined on any element, however, usually the root element (**Window** or **Page**) is used
  - In general, it is best to define resources in the lowest level of scoping
- Each resource in a resource dictionary must have a unique key
  - In markup, the **x:Key** attribute is used
- Resources are a specialized form of data binding, optimized for a high number of bindings with low update frequency

```
<Window.Resources>
  <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
  <Style TargetType="Border" x:Key="PageBackground">
    <Setter Property="Background" Value="Blue"/>
  </Style>
  <Style TargetType="TextBlock" x:Key="TitleText">
    <Setter Property="DockPanel.Dock" Value="Top"/>
    <Setter Property="FontSize" Value="18"/>
    <Setter Property="Foreground" Value="#4E87D4"/>
    <Setter Property="FontFamily" Value="Trebuchet MS"/>
    <Setter Property="Margin" Value="0,40,10,10"/>
  </Style>
  <Style TargetType="TextBlock" x:Key="Label">
    <Setter Property="DockPanel.Dock" Value="Right"/>
    <Setter Property="FontSize" Value="8"/>
    <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
    <Setter Property="FontFamily" Value="Arial"/>
  </Style>
</Window.Resources>
<StackPanel>
  <Border Style="{StaticResource PageBackground}">
    <DockPanel>
      <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
      <TextBlock Style="{StaticResource Label}">Label</TextBlock>
      <TextBlock DockPanel.Dock="Top" HorizontalAlignment="Left" FontSize="36"
                 Foreground="{StaticResource MyBrush}" Text="Text" Margin="20" />
      <Button DockPanel.Dock="Top" HorizontalAlignment="Left" Height="30"
              Background="{StaticResource MyBrush}" Margin="40">Button</Button>
    </DockPanel>
  </Border>
</StackPanel>
```

Resources cont.

- Resources can also be defined at **Application** level
  - In such case, they are available for all windows, pages, etc.

```
<Application x:Class="WpfApplication1.App"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="Data/Resources.xaml"
   Startup="Application_Startup" Exit="Application_Exit">
    <Application.Resources>
      <SolidColorBrush x:Key="MainBrush">Black</SolidColorBrush>
  </Application.Resources>
</Application>
```

```
<Window [...]>
  <Ellipse DockPanel.Dock="Top" HorizontalAlignment="Left" Width="100"
           Height="100" Fill="{StaticResource MainBrush}" Margin="40" />
</Window>
```

Dynamic Resources

- Dynamic resources can be modified at runtime
  - It can be used to cause updates across all the windows or pages in an application

```
<Ellipse DockPanel.Dock="Top" HorizontalAlignment="Left"
         Width="100" Height="100" Margin="40"
         Fill="{DynamicResource MainBrush}"
         MouseDown="Ellipse_MouseDown" />
```

```
private void Ellipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    Application.Current.Resources["MainBrush"] = new SolidColorBrush(Colors.Blue);
}
```

- To perform a dynamic resource binding in code, use the **SetResourceReference** method

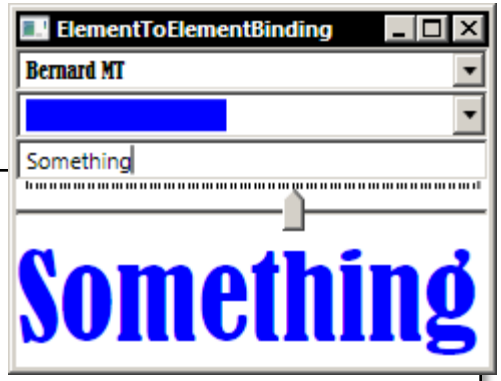
```
button1.SetResourceReference(Button.BackgroundProperty,"MainBrush");
```

- Using dynamic resource references does introduce some performance overhead

Data Binding

- Data binding is a relationship that tells WPF to extract some information from a source object and use it to set a property in a target object
  - The target property is always a dependency property, and it's usually in a WPF element
  - The source object can be just about anything, ranging from another WPF element to an ADO.NET data object or a data-only custom object

Element-To-Element Binding Example




```
<StackPanel>
  <StackPanel.Resources>
    <x:Array x:Key="AvailableColors" x:Type="{x:Type Color}">
      <x:Static Member="Colors.Black"/>
      <x:Static Member="Colors.Red"/>
      <x:Static Member="Colors.Green"/>
      <x:Static Member="Colors.Blue"/>
    </x:Array>
    <CollectionViewSource
           Source="{Binding Source={x:Static Fonts.SystemFontFamilies}}"
           x:Key="AvailableFontFamilies"/>
  </StackPanel.Resources>

  <ComboBox Name="fontFamilyComboBox"
           ItemsSource="{Binding Source={StaticResource AvailableFontFamilies}}"
           SelectedIndex="0" IsReadOnly="True">
    <ComboBox.ItemTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding}" FontFamily="{Binding}"/>
      </DataTemplate>
    </ComboBox.ItemTemplate>
  </ComboBox>
```

```
<ComboBox Name="colorsComboBox"
          ItemsSource="{StaticResource AvailableColors}"
          SelectedIndex="2" IsReadOnly="True">
  <ComboBox.ItemTemplate>
    <DataTemplate>
      <Border x:Name="SelectedItemBorder" MinHeight="16" MinWidth="100" >
        <Border.Background>
          <SolidColorBrush Color="{Binding}"/>
        </Border.Background>
      </Border>
    </DataTemplate>
  </ComboBox.ItemTemplate>
</ComboBox>

<TextBox Name="textBox">Something</TextBox>

<Slider Name="sizeSlider" TickFrequency="1" TickPlacement="TopLeft"
       Minimum="1" Maximum="96" Value="10"/>

<TextBlock Text="{Binding ElementName=textBox, Path=Text}"
           FontSize="{Binding ElementName=sizeSlider, Path=Value}"
           FontFamily=
                 "{Binding ElementName=fontFamilyComboBox, Path=SelectedItem}" >
  <TextBlock.Foreground>
    <SolidColorBrush Color=
                 "{Binding ElementName=colorsComboBox, Path=SelectedItem}" />
  </TextBlock.Foreground>
</TextBlock>

</StackPanel>
```

Binding Errors

- WPF doesn't raise exceptions to notify about data binding problems
  - If the property does not exist, the data will simply fail to appear
- Fortunately, WPF does output trace information that details binding failures
  - This information appears in Visual Studio's Output window when you're debugging the application

```
System.Windows.Data Error: 22 : Cannot convert '#FF008000' from type 'Color' to
type 'System.Windows.Media.Brush' for 'en-US' culture with default conversions;
consider using Converter property of Binding. NotSupportedException:
'System.NotSupportedException: BrushConverter cannot convert from
System.Windows.Media.Color.
   at System.ComponentModel.TypeConverter.GetConvertFromException(Object value)
   at System.ComponentModel.TypeConverter.ConvertFrom(ITypeDescriptorContext
   at System.Windows.Media.BrushConverter.ConvertFrom(ITypeDescriptorContext
   at MS.Internal.Data.DefaultValueConverter.ConvertHelper(Object o, Type
```

- It is possible to change the appearance of the source element to indicate that an error has occurred

Creating Bindings with Code

```
Binding binding = new Binding();
binding.Source = sizeSlider;
binding.Path = new PropertyPath("Value");
previewTextBlock.SetBinding(FontSizeProperty, binding);
```

- Binding can be removed using one of two static method of the **BindingOperations** class: **ClearBinding** or **ClearAllBindings**
  - These methods can remove also bindings defined in XAML
- It is preferred to define bindings in XAML

Binding Direction

- There are 5 available binding modes defined in the **BindingMode** enumeration:
  - **OneWay** – the target is updated according to the source's changes
  - **TwoWay** – the target is updated when the source changes, and the source is updated when the target changes
  - **OneTime** – the target is set initially and all later changes of the source are ignored
  - **OneWayToSource** – the source is updated according the target's changes
  - **Default** – it is either **TwoWay** (for user-settable properties, e.g. TextBox.Text) or **OneWay** (for everything else)

Binding to Objects That Aren't Elements

- Any public property of any object can be used as a source for binding
  - It must be a public property – private information or public fields cannot be used
- One of the following properties can be used for binding to objects that aren't elements:
  - **Source**
  - **RelativeSource**
  - **DataContext**

Binding: Source

- The **Source** property points the object (and its public property)
- The simplest option is to point the **Source** to some static object that's readily available

```
<TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily}}" />
```

- Another option is to bind to an object that was previously created as a resource

```
<StackPanel.Resources>
  <FontFamily x:Key="SpecialFont">Trebuchet MS</FontFamily>
</StackPanel.Resources>
<TextBlock Text="{Binding Source={StaticResource SpecialFont}}" />
```

Binding: RelativeSource

- The **RelativeSource** property allows you to point to a source object based on its relation to the target object
  - One option is to use the property-setting syntax instead of the Binding markup extension:

```
<TextBlock>
  <TextBlock.Text>
      <Binding Path="Title">
        <Binding.RelativeSource>
          <RelativeSource Mode="FindAncestor" AncestorType="{x:Type Window}" />
        </Binding.RelativeSource>
      </Binding>
    </TextBlock.Text>
</TextBlock>
```

  - The more common way is to combine the binding into one string using the Binding and RelativeSource markup

```
<TextBlock Text="{Binding Path=Title, RelativeSource=
                  {RelativeSource FindAncestor, AncestorType={x:Type Window}}}">
</TextBlock>
```

Binding: RelativeSourceMode

- There are four options available for **RelativeSources**:
  - **Self** – the expression binds to another property in the same element
  - **FindAncestor** - the expression binds to a parent element; the **AncestorType** property must be used, optionally also the **AncestorLevel** property can be used
  - **PreviousData** – the expression binds to the previous data item in a data-bound list
  - **TemplatedParent** – the expression binds to the element on which the template is applied; it works only if the binding is located inside a control template or data template

Binding: DataContext

- When the source information is missing from a binding expression, WPF checks the **DataContext** property of that element
  - If it's null, WPF searches up the element tree looking for the first data context that isn't null.

```
<StackPanel>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
                            Path=Source}"/>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
                            Path=LineSpacing}"/>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
                            Path=FamilyTypefaces[0].Style}"/>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
                            Path=FamilyTypefaces[0].Weight}"/>
</StackPanel>
```

```
<StackPanel DataContext="{x:Static SystemFonts.IconFontFamily}">
  <TextBlock Text="{Binding Path=Source}"/>
  <TextBlock Text="{Binding Path=LineSpacing}"/>
  <TextBlock Text="{Binding Path=FamilyTypefaces[0].Style}"/>
  <TextBlock Text="{Binding Path=FamilyTypefaces[0].Weight}"/>
</StackPanel>
```

Binding to a Collection of Objects

- To support collection binding, the **ItemsControl** class defines the three key properties:
  - **ItemsSource** – points to the collection of objects
  - **DisplayMemberPath** – identifiers the property that will be used to create the display text for each item
  - **ItemTemplate** – accepts a data template that will be used to create the visual appearance of each item

```
<Window x:Class="WpfApplication1.Data.BindingToCollection"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:src="clr-namespace:WpfApplication1"
   Title="BindingToCollection" Height="300" Width="300">
  <StackPanel>
    <StackPanel.Resources>
      <src:Customers x:Key="customers"/>
    </StackPanel.Resources>
    <ListBox ItemsSource="{StaticResource customers}" DisplayMemberPath="LastName"/>
  </StackPanel>
</Window>
```

ObservableCollection<T>

- WPF provides the **ObservableCollection<T>** class, which is a built-in implementation of a data collection that implements the **INotifyCollectionChanged** interface
  - This interface exposes the **CollectionChanged** event, an event that should be raised whenever the underlying collection changes

Data Conversion

- The value converter is responsible for converting the source data just before it's displayed in the target
  - In the case of a two-way binding, it is also responsible for converting the new target value just before it's applied back to the source
- Value converters can be used to:
  - Format data to a string representation
  - Create a specific type of WPF object
  - Conditionally alter a property in an element based on the bound data

Creating a Value Converter

1. Create a class that implements **IValueConverter**
2. Add the **ValueConversion** attribute to the class declaration, and specify the destination and target data types
3. Implement a **Convert** method that changes data from its original format to its display format
4. Implement a **ConvertBack** method that does the reverse and changes a value from display format to its native format

```
<Window x:Class="WpfApplication1.Data.DateConverterExample"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:src="clr-namespace:WpfApplication1"
   xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <Grid>
    <Grid.Resources>
      <src:DateConverter x:Key="dateConverter"/>
    </Grid.Resources>
    <TextBlock Grid.Row="0" Grid.Column="1"
               Text="{Binding Source={x:Static sys:DateTime.Now},
                              Converter={StaticResource dateConverter}}"/>
  </Grid>
</Window>
```

Value Converter Example

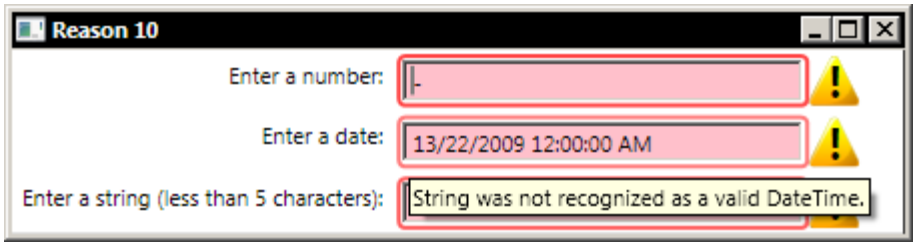
```
[ValueConversion(typeof(DateTime), typeof(String))]
public class DateConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
                          object parameter, CultureInfo culture)
    {
        DateTime date = (DateTime)value;
        return date.ToShortDateString();
    }

    public object ConvertBack(object value, Type targetType,
                              object parameter, CultureInfo culture)
    {
        string strValue = value.ToString();
        DateTime resultDateTime;
        if (DateTime.TryParse(strValue, out resultDateTime))
        {
            return resultDateTime;
        }
        return value;
    }
}
```

Validation

- WPF provides a validation feature that works closely with the data binding system
- There are two ways of handling validation
  - Simply throw an exception from a property set procedure
    - The **ExceptionValidationRule** is a prebuilt validation rule that tells WPF to report all exceptions (by default, WPF ignores any exception thrown when setting a property)
  - Define validation at the binding level
    - This gives the flexibility to use the same validation regardless of the input control

Validation Example




```
<TextBox Grid.Column="1" Grid.Row="1" Width="200" Margin="5,5,50,5"
         Style="{StaticResource highlightValidationError}"
         Validation.ErrorTemplate="{StaticResource validationTemplate}">
    <TextBox.Text>
        <Binding Path="SomeDate" UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <ExceptionValidationRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

http://www.thejoyofcode.com/Reason_10._Validation.aspx

```
<ControlTemplate x:Key="validationTemplate">
  <DockPanel>
    <Border Name="validationBorder" BorderBrush="Red" BorderThickness="2"
            Padding="1" CornerRadius="3">
      <Border.Resources>
        <Storyboard x:Key="_blink">
          <ColorAnimationUsingKeyFrames AutoReverse="True" BeginTime="00:00:00"
                 Storyboard.TargetName="validationBorder"
            Storyboard.TargetProperty="(Border.BorderBrush).(SolidColorBrush.Color)"
                 RepeatBehavior="Forever">
            <SplineColorKeyFrame KeyTime="00:00:00.5" Value="#00FF0000"/>
          </ColorAnimationUsingKeyFrames>
        </Storyboard>
      </Border.Resources>
      <Border.Triggers> [...] </Border.Triggers>
      <AdornedElementPlaceholder/>
    </Border>
    <src:Exclamation Width="25" Height="25" />
  </DockPanel>
</ControlTemplate>
<Style x:Key="highlightValidationError" >
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="TextBox.Background" Value="Pink" />
      <Setter Property="TextBox.ToolTip"
           Value="{Binding RelativeSource={x:Static RelativeSource.Self},
                           Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

Data Template

- A data template is a chunk of XAML markup that defines how a bound data object should be displayed
- Two types of controls support data templates:
  - Content controls support data templates through the **ContentTemplate** property
  - List controls support data templates through the **ItemTemplate** property

```
<ListBox Name="personsListBox">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding Path=FirstName}"></TextBlock>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```