Windows Presentation Foundation

Some Basics

Role of XAML

- The XAML describes the view
  - How should it look like
- The XAML also describes the databinding
  - What data is presented, how it is presented
- The XAML also describes event handling
  - OnClick
- The XAML also describes commanding
  - Cut, Copy, Paste and own commands
- The XAML can also describe some behaviors for the view
  - Triggers and animations are a good examples

Describing the UI

- The XAML is used to describe the UI
- We need the root-element
  - And quite a few XML-namespaces declared
- And we describe the UI for the root element
- Quite a few of the controls available inherit ContentControl
  - Very complicated content can be designed for those

MainWindow.xaml

```
<Window x:Class="WpfTests.MainWindow“  . . .
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Calendar HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" />
        <Button x:Name="button" Margin="25,183,0,0" Width="135">
            <StackPanel Orientation="Horizontal">
                <Ellipse Width="20" Height="20" Fill="Red" />
                <Label Content="Press me" />
            </StackPanel>
        </Button>

    </Grid>
</Window>
```

Events

- Event handling is quite straight-forward
    - Event is described in XAML, the handler is implemented to Window-class
- Just remember that events traverse the control-hierarchy, routed events
    - Preview[EVENT] -handlers kind of peek if the event is coming, handled from bottom-most control upwards
    - [EVENT]-handler actually processes the event, handled from top-most control downwards

MainWindow.xaml

```
<Button x:Name="button" . . . Click="button_Click" MouseMove="ButtonMouseMove" >
    <StackPanel Orientation="Horizontal">
        <Ellipse Width="20" Height="20" Fill="Red" />
        <Label x:Name="buttonLabel" Content="Press me" MouseMove="LabelMouseMove"
/>
    </StackPanel>
</Button>
<Label x:Name="labelInfo" Content="Label" . . . />
<Label x:Name="buttonInfo" Content="Label" . . . />
```

```
private void LabelMouseMove(object sender, MouseEventArgs e)
{
    labelInfo.Content = "Move "+e.GetPosition(buttonLabel).X;
    e.Handled = true; // Now ButtonMouseMove will not be called
}
```

Databinding

- Databinding is one of the strongest features of WPF
  - Sadly sometimes ignored and misused
- We have
  - Binding target => Attribute of an element is described with binding
  - Binding source => Property of an object providing the data
  - Type conversion, in many cases automatic
  - Validation, can be declared and implemented
  - Two-way, One-way, One-way to the source, One-time

MainWindow.xaml (Different binding modes between UI-elements)

```
<TextBox x:Name="textBox1" Text="{Binding ElementName=slider1,Path=Value}" . . ./>
<Slider x:Name="slider1"  Minimum="0" Maximum="20" . . ./>

<TextBox x:Name="textBox2"
            Text="{Binding ElementName=slider2,Path=Value,Mode=OneWay}" . . ./>
<Slider x:Name="slider2"  Minimum="0" Maximum="20" . . ./>

<TextBox x:Name="textBox3"
            Text="{Binding ElementName=slider3,Path=Value, Mode=OneWayToSource}" . . ./>
<Slider x:Name="slider3"  Minimum="0" Maximum="20" . . ./>

<TextBox x:Name="textBox4"
            Text="{Binding ElementName=slider4,Path=Value,Mode=OneTime}" . . ./>
<Slider x:Name="slider4" Minimum="0" Maximum="20" Value="10" . . ./>
```

Exercise

- Create a new project
  - Call it MvvmCalculator though not implementing calculator yet
- Just experiment with textbox-slider bindings as demonstrated on previous slide

Setting DataContext to custom object

- Often you should create a new object-type to contain data for your ui (the ViewModel)
    - Instead of adding huge number of properties to you Window-class
- Then you can set the DataContext-property to point to a new object of your class
    - On window or any other container
- Of course the object can also be instantiated and the DataContext be set in the constructor

```
class MyData
{
    public string data { get; set; }

    public MyData()
    {
        data = "Hello";
    }
}
```

```
<!-- local namespace  -->
   <Window.DataContext>
        <local:MyData />
   </Window.DataContext>
```

OR

```
public MainWindow()
{
    DataContext=new MyData();
    InitializeComponent();
}
```

DataContext and ViewModel

- ViewModel(s) are most often declared as resources at application context
  - And the DataContext is then set to the entire Window/Page
- How-evern different controls still may have different DataContexts
- It is still possible to refer to properties of basically any object with data binding

```
<Application.Resources>
  <local:GreetingVM
      x:Key="GreetingVM" />
```

```
<Window
  DataContext="{StaticResource
                 GreetingVM}"
```

Different bindings

```
<!-- Current containers DataContext -->
<TextBox Text="{Binding data}" . . ./>

<!-- Any resource available -->
<TextBox Text="{Binding Source={StaticResource otherData},Path=data}" . . ./>

<!-- Windows properties, two notations basically synonyms -->
<TextBox
   Text="{Binding RelativeSource={RelativeSource
   AncestorType={x:Type Window}},Path=data}" . . ./>
<TextBox Text="{Binding RelativeSource={RelativeSource
       Mode=FindAncestor, AncestorType=Window},Path=data}" . . ./>
```

Notifications

- To reflect changes of data to UI the model objects need to provide a PropertyChanged-event

MyData.cs

```
class MyData : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string _data = "";
    public string data
    {
        get { return _data; }
        set
        {
            _data = value;
            if (PropertyChanged!=null)
                PropertyChanged(this, new PropertyChangedEventArgs("data"));
        }
    }
}
```

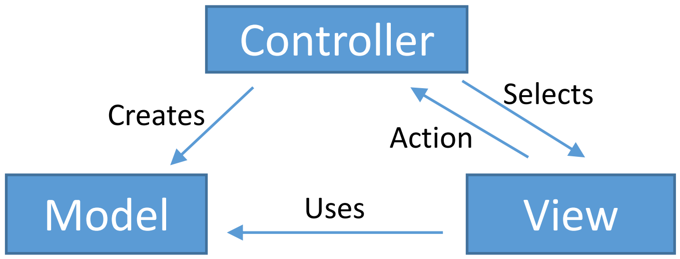
Exercise

- Now you can create calculator
- First create Calculation-class
  - Figure1, Figure2 properties with get and set
  - Calculated property Result
- Publish Calculation as Resource
- Create calculator UI
- Use databinding to update data

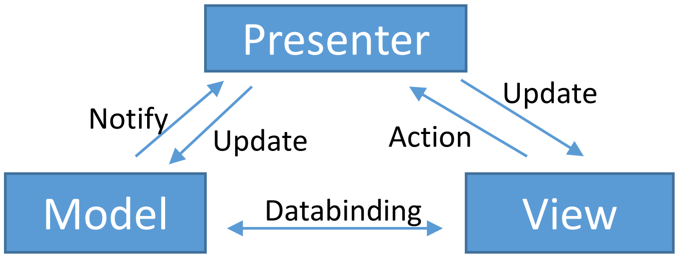
WPF and MVVM

UI-patterns

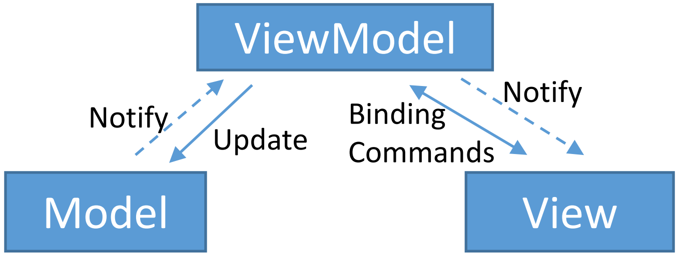
- Model-View-Controller
  - Decouples the user interface from the data model
  - Most suitable for Web development



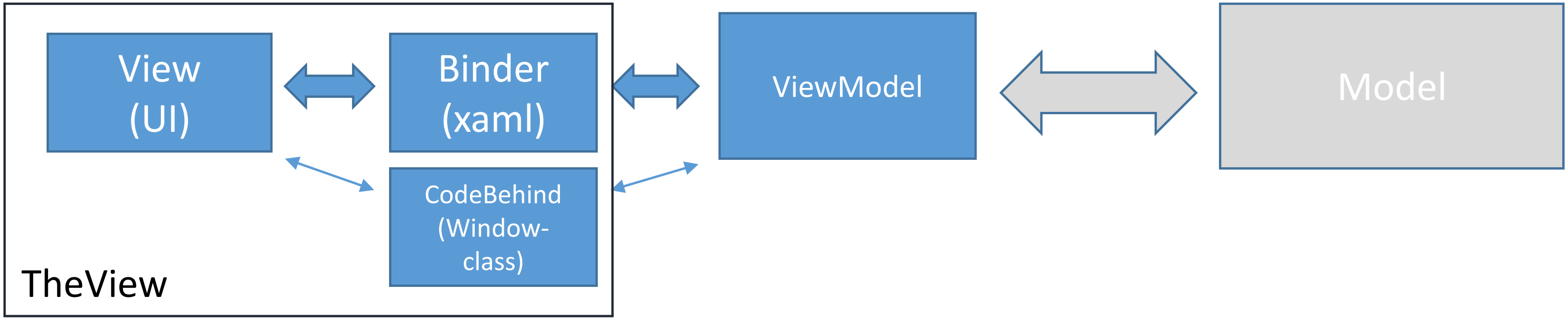
- Model-View-Presenter
  - Evolves the MVC pattern for event-driven applications
  - Most suitable for forms-over-data development
  - Introduces databinding



- Model-View-ViewModel
  - Evolves from the MVP pattern
  - Most suitable for WPF applications
  - More loosely coupled

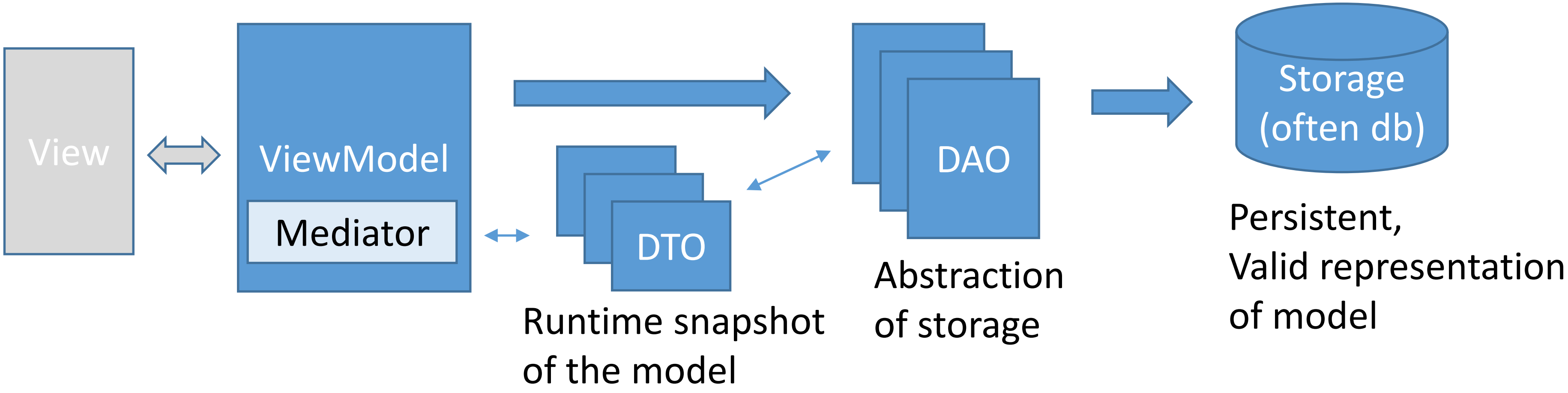


MVVM - View, Binder and ViewModel



- MVVM is a pattern created by Microsoft especially for WPF-applications
  - Now-a-days used on other platforms also
- Like any pattern can actually be implemented in many ways
- The key idea is the abstraction of data- and command-binding to xaml
  - Basis for WPF-programming
  - Deep understanding of xaml and binding mechanisms provided by it are needed
- ViewModel provides the data to the view in format it is easily used in the ui: Mediator, Adapter
    - Objects should/could implement INotifyPropertyChanged
    - Collections should/could implement INotifyCollectionChanged (or inherit ObservableCollection)
- CodeBehind may implement UI-logic associated with the view
  - ViewModel also may contain logic, but it should be view -independent

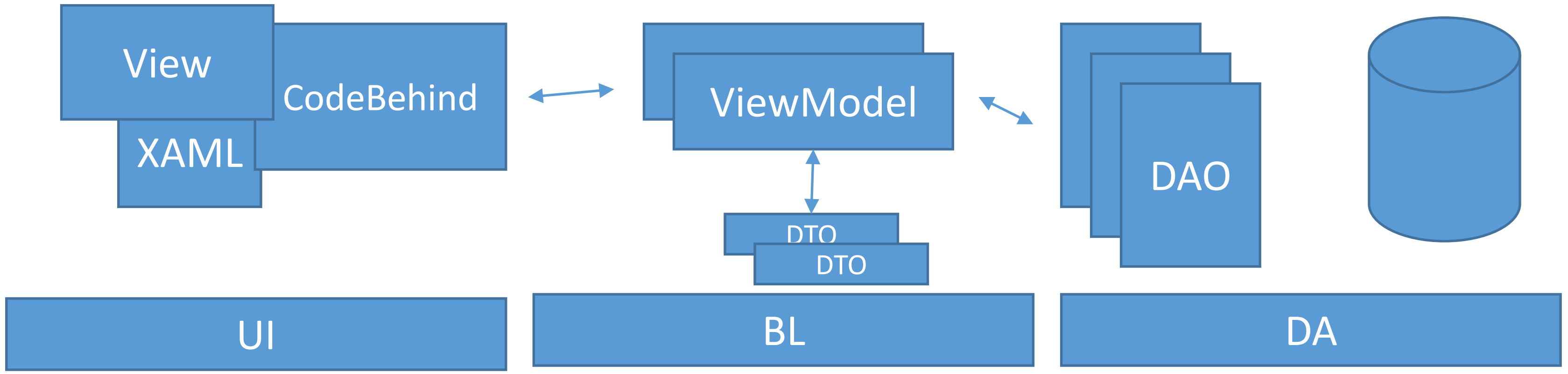
MVVM - ViewModel and Model


View
ViewModel
Mediator
DTO
DAO
Storage (often db)
Runtime snapshot of the model
Abstraction of storage
Persistent, Valid representation of model

THERE IS NOT JUST ONE STRAIGHT-FORWARD, ALWAYS TO USE SOLUTION

- At runtime the model is represented by collection of objects holding the data application is to maintain
- Some abstraction is needed on how the model is generated
  - Factories, Repositories
  - If data is stored into a database or available through web services DAO-pattern is very convenient
- The ViewModel shouldn’t create the model-objects but ask them from a “third-party”
  - And most likely we end up with having Data Transfer Objects (Value Objects) holding the data
- The ViewModel requests data from DAO-objects getting multiple DTOs
  - ViewModel then constructs (or operates as) mediator providing services that combine the manipulation of otherwise unrelated DTOs

Three-tiered design



- No, it is not just for distributed architectures
  - Originally it was developed for desktop applications to separate concerns
- All application
  - Present data : UI, the view and the logic associated with it
  - Manipulate data: BL, manipulation is based on rules
  - Store data: DA, Often storage is DB but it may be a disk file or accessed through web services

So patterns, patterns, patterns…

- Patterns offer solutions for specific problems
  - Abstracting technology specific implementation
  - Abstracting complicated logic
  - Providing flexibility to changes
- But
  - If misused they just complicate the solution
  - Keep simple cases simple
  - If you can rewrite something in an hour don’t spend four hours trying to force a pattern into it (Golden Hammer antipattern)
- If your architecture requires the use of patterns
  - Then use them consistently

Working with collections

And item templates

Exercise

- Create CalculatorVM
  - Holds Current-property that points to Calculation-object
  - List<Calculation> Calculations –property
  - Use databinding in the calculator to CalculatorVM instead of original Calculation
- Add a Listbox to the Calculator that displays the calculations done
  - Might want to initialize Calculations to hold some calculations
  - Set the ItemsSource-property for the listbox
  - What else is needed?
- Add “Add”-button to the calculator, it should add the current calculation to the Calculations-list
  - How do you accomplish this?
  - What do you notice of behaviour?
- When item is clicked on listbox bring the show the selected calculation

Having a collection in the model

- Data binding can be done towards any IEnumerable
  - How ever the UI will not automatically know if items are added, deleted or replaced in the collection
- INotifyCollectionChanged describes the event that should be fired when collection changes
- ObservableCollection is a convenience-class implementing INotyfyCollectionChanged
  - Provides List-like operations, but doesn’t implement IList
  - Can be instantiated from any Ienumerable
  - Easy to use, and misuse…

```
List<Person> personList =. . .;
ObservableCollection<Person> col=new ObservableCollection<Person>(personList);
```

How to use ObservableCollection

- Not all collections in the ViewModel need to be Observable, analyze your need
- Don’t replace the collection, modify the contents
  - Events are subscribed from a specific instance
  - OK, clearing and then adding huge number of items is not a good idea either
- You can use LINQ against ObservableCollection but the returned collection is not Observable
- Don’t modify in background thread
  - Modifications cause events that should be processed in the main thread
- So quite a few design considerations….

Item templates

- Item-template describes how the contents of an object should be shown
- ItemTemplate holds or refers to DataTemplate describing the UI

Sample with ListBox

```
<ListBox
    ItemsSource="{Binding Source={StaticResource personVM}, Path=Persons}“ . . .>
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Label Content="{Binding Path=Name}" />
                <Label Content="{Binding Path=Email}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Some other features

Commanding, converters, validators

Commanding

- Commanding offers more separation on object invoking the command and object executing the command
  - Several sources for single action
  - UI understands if the command can be invoked
- Several components know how to invoke commands
  - Buttons
  - Menus
- Some components have built-in intelligence for processing commands, CommandTarget needs to be specified
  - TextArea
- Some built-in commands exist
  - Printing related: Print, Preview, Cancel..
  - File -related: Open, Close…
  - Edit-related: Cut, Copy, Paste…
  - Movement: MoveUp, MoveLeft….

Using predefined commands

- Window needs to have the command binding
- Code behind needs to have methods describe with Executed and CanExecute attributes

MainWindow.xaml

```
. . .
<Window.CommandBindings>
    <CommandBinding    Command="Close“
                       Executed="ExecuteClose" CanExecute="CanClose"/>
</Window.CommandBindings>

. . .
<CheckBox x:Name="canClose" Content="Can close" . . ./>
<Button Command="Close" Content="Close" . . ./>
. . .
```

```
private void ExecuteClose(object sender, ExecutedRoutedEventArgs e)  {  Close();    }

private void CanClose(object sender, CanExecuteRoutedEventArgs e)  {
    e.CanExecute=(bool)canClose.IsChecked;
}
```

Invoking commands

- Buttons, menus etc can invoke commands
- Keyboard actions can invoke commands
- And of course commands can be invoked from code

MainWindow.xaml (InputBinding for keyboard-events)

```
<Window.InputBindings>
    <KeyBinding Key="C" Modifiers="Control" Command="Close" />
</Window.InputBindings>
```

MainWindow.xaml.cs (Invoking command from code)

```
private void SomeEvent(object sender, RoutedEventArgs e)
{
    // CommandBinding specifies x:Name="MyClose"
    if (MyClose.Command.CanExecute(null))
        MyClose.Command.Execute(null);
}
```

Custom commands

- You need to define the custom command objects
  - static class containing commands as static members for very generic commands
  - ICommand-members in ViewModel for ViewModel specific commands (Save, Reload, Search etc)
- For each member give
  - Text (to be displayed as helper for this commad)
  - Name (Short, descriptive name to be used in XAML)
  - Holding type
  - And possibly also a collection of “InputGestures” that cause this command

CustomCommands.cs

```
public static class CustomCommands
{
    public static readonly RoutedUICommand MyCommand =
                new RoutedUICommand(
                    "My new action",
                    "MyAction",
                    typeof(CustomCommands),
                    new InputGestureCollection {
                        new KeyGesture(Key.A,ModifierKeys.Control)
                    }
                );
}
```

Using custom commands

- No different from predefined commands
- Just make sure you declare the namespace for your custom-command-class in XAML
- InputGestures defined for the command work automatically

MainWindow.xaml

```
<Window x:Class="WpfTests.MainWindow"
    . . .
    xmlns:local="clr-namespace:WpfTests“
    . . .
    Title="MainWindow" Height="350" Width="525">
    <Window.CommandBindings>
        <CommandBinding Command="local:CustomCommands.MyCommand"
                        CanExecute="CanAct" Executed="DoAct" />
    </Window.CommandBindings>


. . .
 <Button Command="local:CustomCommands.MyCommand" Content=“Do action“ . . . />
```

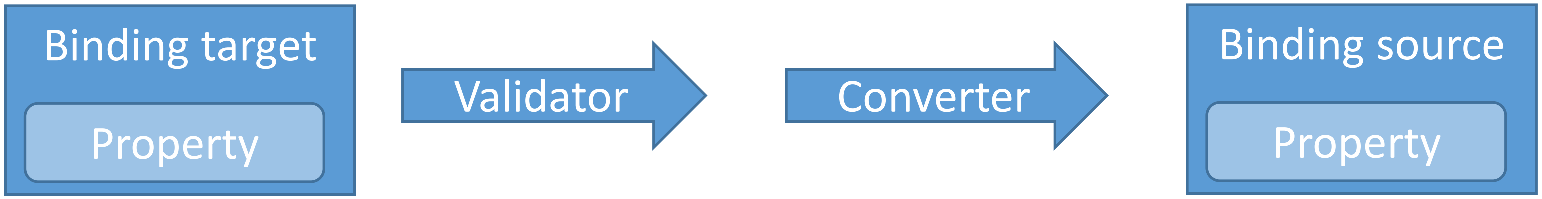
Custom command in ViewModel

- The ViewModel may contain properties that implement ICommand
  - Variations exist on how to implement

```
public ICommand SaveCommand { get; private set; }

public MyViewModel()
{
    SaveCommand = new SaveCommandHandler((obj) => {MessageBox.Show("Executing save");});
}

private class SaveCommandHandler : ICommand
{
    private Action<object> action;

    public SaveCommandHandler(Action<object> act){
        action = act;
    }


    public void Execute(object parameter)
    {
        action(parameter);
    }

. . .
}
```

```
<!--    DataContext must be MyViewModel      -->
<Button Command="{Binding SaveCommand}” . . . />
```

Converters and Validators



Binding target

Property

Validator

Converter

Binding source

Property

- Data binding nearly always passes through a converter
  - Integer property to a contant string
  - Implicit or explicit
- Also a validator can be defined

Custom converter and several validators

```
<TextBox . . .>
    <TextBox.Text>
        <Binding Source="{StaticResource personVM}"
                        Path="BirthDate" Converter="{StaticResource ageConverter}">
            <Binding.ValidationRules>
                <ExceptionValidationRule />
                <DataErrorValidationRule />
                <local:AgeValidationRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Custom validator

- Just inherit ValidationRule and implement Validate-method

Age validation

```
class AgeValidationRule : ValidationRule
{
    public override ValidationResult Validate(object value,
                                      CultureInfo cultureInfo)
    {
        int a = int.Parse(value.ToString());
        if (a < 0) return new ValidationResult(false, "Cannot be");
        if (a < 18) return new ValidationResult(false, "Must be adult");
        if (a > 100) return new ValidationResult(false, "I doubt...");
        return ValidationResult.ValidResult;
    }
}
```

Custom converter

- Often the same result can be achieved by implementing a calculated property to a ViewModel
- But if same conversion is needed in several ViewModel-objects then custom converter can be reused

Converting birthdate to age (not a complete solution)

```
[ValueConversion(typeof(string),typeof(DateTime))]
class DateToAgeConverter : IValueConverter
{
    public object Convert(object value, Type tt, object parameter, CultureInfo culture)
    {
        DateTime dt = (DateTime)value;
        return DateTime.Now.Year - dt.Year; // OK, not exact
    }

    public object ConvertBack(object value, Type tt, object parameter, CultureInfo cult)
    {
        return new DateTime(DateTime.Now.Year - int.Parse(value.ToString()), 1, 1);
    }
}
```

Exercise

- Person should have Birthday
  - Impelement a validator that validates that the birthday is not in futere
- Person should have Gender, just implement boolean property indicating if person is male
  - Implement converter that translates boolean value to String

```
<ControlTemplate x:Key="dateValidationTemplate">
    <DockPanel>
        <TextBlock Foreground="Red" FontSize="20">Note!</TextBlock>
        <AdornedElementPlaceholder/>
    </DockPanel>
</ControlTemplate>

<TextBlock HorizontalAlignment="Left" Margin="44,86,0,0" TextWrapping="Wrap"
         Text="{Binding Path=Gender,Converter={StaticResource genderConv}}" />

 <TextBox Style="{StaticResource dateTextStyleTextBox}"
                 Validation.ErrorTemplate="{StaticResource dateValidationTemplate}" >
            <Binding Path="Birth">
                <Binding.ValidationRules>
                    <local:DateValidationRule />
                </Binding.ValidationRules>
            </Binding>
</TextBox>
```

Some XAML-techniques

Resources

- Resources are just objects reusable on several occasions
- Can be defined at
  - Application context
  - Window (or Page) context
  - Element context
- Resources may be referenced
  - Statically: Resolved when XAML is parsed
  - Dynamically: When resource is actually needed at runtime
- Should the Application declare a resource for the ViewModel-object?
  - Most often at least partially, but wholly can be debated

Resources in App.xaml

- All the resources that are used between views should be described in App.xaml
  - Unless of course a view-specific instance of the resource-type is needed

App.xaml

```
<Application x:Class="WpfTests.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfTests"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <SolidColorBrush x:Key="blueBrush" Color="Blue"/>
        <SolidColorBrush x:Key="whiteBrush" Color="White"/>
        <sys:Double x:Key="myValue">100</sys:Double>
        <local:PersonVM x:Key="personVM" />
        <local:DateToAgeConverter x:Key="ageConverter" />
        <local:AgeValidationRule x:Key="ageRule" />
    </Application.Resources>
</Application>
```

Styles

- Styles are also resources
- Again styles that are used across the views should be defined in App.xaml
- Both of the following are not applied
  - When x:Key is set an element may use the style by binding the Style-property to “{StaticResource [x:Key]}”
  - When x:Key is not set the style is used for all elements of TargetType

Styles in resources

```
<Style TargetType="{x:Type Label}">
    <Setter Property="FontSize" Value="24" />
</Style>
<Style x:Key="warningBkr" TargetType="{x:Type Label}">
    <Setter Property="Background" Value="Red" />
</Style>
```

Style inheritance

- Problem on previous slide
  - All labels have spefic font
  - Except those that choose the “warningBkr” -style
- Styles can inherit definitions of other styles
- BasedOn-attribute

Style inheritance

```
<Style x:Key="labelBase" TargetType="Label">
    <Setter Property="FontSize" Value="24" />
</Style>

<Style TargetType="Label" BasedOn="{StaticResource labelBase}">
    . . . <!- Everything on labelBase applies to all labels
</Style>
<Style x:Key="warningBkr" BasedOn="{StaticResource labelBase}"  TargetType="Label">
    <Setter Property="Background" Value="Red" />
    <!-- labelBase still applies -->
</Style>
```

Templates

- Styles can also be used for describing templates for the controls
  - How they should appear

All buttons should be displayd as blue ellipses

```
<Style TargetType="Button">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="Button">
                <Grid>
                    <Ellipse Fill="Blue"/>
                    <ContentPresenter VerticalAlignment="Center"
                             HorizontalAlignment="Left"/>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```