HOCHSCHULE HAMM-LIPPSTADT

Praktische Informatik

Vorlesung 09

Styling

Zuletzt haben wir gelernt…

- Das man häufig Daten zwischen Teilen von Anwendungen synchronisieren muss.
- Wie in WPF mit Element-Bindung synchronisiert werden kann.
- Welche Eigenschaften die Klasse Binding besitzt.
- Welche Bindungsmodi existieren und wie Converter verwendet werden können.
- Wie man Datenbindung einsetzt.
- Warum in Datenmodellen, die angebunden werden sollen die Schnittstelle INotifyPropertyChanged benötigt wird.
- Was der Unterschied zwischen MVC und MVVM ist.

Inhalt heute

- Logische Ressourcen
- Styles
- Trigger
- ControlTemplate
- Drag-and-Drop mit Thumb

Logische Ressourcen

- In der letzten Vorlesung haben wir für die Datenbindung im Temperaturumrechner eine Converter-Klasse erstellt.
  - Diese Converter-Klasse war dafür zuständig, die Temperatur in Celsius in Fahrenheit umzurechnen.

- Das Objekt dieser Klasse mussten wir als sog. **logische Ressource** in XAML einbinden.
  - Erst dann konnten wir den Converter auch nutzen.

- Logische Ressourcen können in XAML für unterschiedliche Zwecke genutzt werden.
  - Wir sollten uns daher etwas näher mit ihnen befassen.

Logische Ressourcen

- Logische Ressourcen sind Objekte, die an unterschiedlichen Stellen in Projekten benutzt werden können.
  - z.B. Converter, Pinsel, Styles, Objekte eigener Klassen, …
- Meist werden logische Ressourcen in XAML definiert.
  - Jede Klasse, die von `FrameworkElement` ableitet, kann logische Ressourcen definieren.
- Die Ressourcen sind für alle Kindeelemente desjenigen Elements benutzbar, in dem es definiert wurde.
  - In der App.xaml definierte Ressourcen sind entsprechend für die gesamte Anwendung global verfügbar.

Logische Ressourcen definieren

- Wir können z.B. einen Pinsel als logische Ressource global für das ganze Fenster definieren.

```
<Window.Resources>
   <SolidColorBrush x:Key="green" Color="green" />
</Window.Resources>
```

- Dabei müssen wir über die Eigenschaft x:Key einen eindeutigen Schlüssel definieren.
  - Über den Schlüssel können wir später die Ressource wieder auffinden und benutzen.

- Genauso können wir auch andere Objekte als Ressourcen definieren.
  - z.B. Strings, oder ein Objekt vom Typ Bruch.

```
<Window.Resources>
    <local:Bruch x:Key="b" Zaehler="1" Nenner="3" />
    <clr:String x:Key="s">Dies ist ein Text</clr:String>
</Window.Resources>
```

Achtung! Die Präfixe **local** und **clr** müssen im Kopf der XAML-Datei mit entsprechenden Namespaces verknüpft sein!

Ressourcen in einer eigenen Datei

- In der Datei App.xml können logische Ressourcen global für die ganze Anwendung definiert werden.
  - Dort kann aber auch eine externe Datei hinzugefügt werden, die alle Ressourcen zentral aufnimmt.
  - Ein sog. **Resource Dictionary**.

```
<Application.Resources>
   <ResourceDictionary Source="Dictionary1.xaml" />
</Application.Resources>
```

- Ein neues Resource Dictionary kann dem Projekt leicht hinzugefügt werden.
  - Kontextmenü Hinzufügen → Ressourcenwörterbuch

- In der neuen Datei können alle Ressourcen global hinterlegt werden.

Ressourcen benutzen

- Wurde eine Ressource mit einem eindeutigen Schlüssel definiert, kann diese leicht benutzt werden.
- Um im XAML einer Eigenschaft eine Ressource zuzuweisen, werden wie beim Data Binding auch die geschweiften Klammern benutzt (markup extension).

```
<Label Content="Ein grünes Label" Background="{StaticResource green}"/>
```

- In diesem Fall wird die Ressource als statische Ressource benutzt.
  - Ändert sich die Ressource zur Laufzeit, wird das Label dies nicht bemerken.
- Im Gegensatz dazu kann auch die Markup Erweiterung DynamicResource benutzt werden.

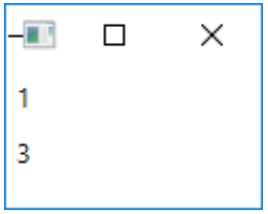
Data Binding mit Ressourcen

- Ressourcen können auch am Data Binding beteiligt werden.
  - Nehmen wir an, wir hätten ein Bruch-Objekt als Ressource erzeugt:

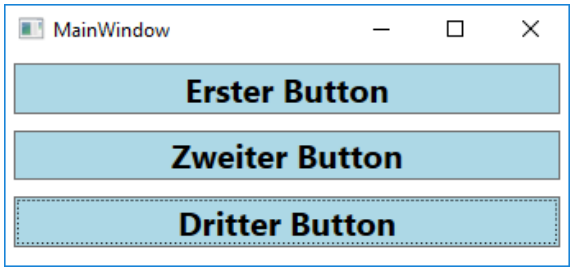
```
<local:Bruch x:Key="bruch" Zaehler="1" Nenner="3" />
```

- Wir können nun zwei Label an Zähler und Nenner des Bruchs binden.
  - Dazu nutzen wir die Eigenschaft Source des Binding-Objektes.

```
<Label Content="{Binding Source={StaticResource bruch}, Path=Zaehler}" />
<Label Content="{Binding Source={StaticResource bruch}, Path=Nenner}" />
```



Styles



- Mit Hilfe von Attributen können wir das Aussehen der Elemente im XAML beeinflussen.
  - Oft wollen wir mehreren Elementen ein ähnliches Aussehen geben.
  - Dabei stellen wir oft fest, dass sich die Angaben für viele Elemente wiederholen.

- Im folgenden Beispiel werden drei Buttons exakt gleich konfiguriert.

```
<Button FontSize="20" Margin="5" Background="LightBlue" FontWeight="Bold">Erster Button</Button>
<Button FontSize="20" Margin="5" Background="LightBlue" FontWeight="Bold">Zweiter Button</Button>
<Button FontSize="20" Margin="5" Background="LightBlue" FontWeight="Bold">Dritter Button</Button>
```

- Bei diesem Vorgehen entsteht viel redundanter XAML-Code.
  - Um dies zu verhindern, können sog. Styles definiert werden.

Styles definieren

- In WPF sind Styles ähnlich zu CSS in Html.
  - Mit Hilfe von Styles kann das Aussehen von beliebig vielen Elementen zentral und gemeinsam definiert werden.

- Jedes Objekt der Klasse `Style` besitzt einen Satz von Settern.
  - Jeder Setter legt genau einen Eigenschaftswert fest.
  - Ein Style wird als Ressource definiert.

- Im folgenden Beispiel wird der Style `btnStyle` als logische Ressource definiert.
  - Dieser Style legt für einige Eigenschaften aus der Klasse Control Werte fest.

```
<Style x:Key="btnStyle">
   <Setter Property="Control.FontSize" Value="20" />
   <Setter Property="Control.Margin" Value="5" />
   <Setter Property="Control.Background" Value="LightBlue" />
</Style>
```

Styles benutzen

- Soll ein Element einen zuvor definierten Style nutzen, muss dies ausdrücklich mitgeteilt werden.
  - Elemente, die von `FrameworkElement` ableiten, besitzen dazu die Abhängigkeitseigenschaft `Style`.

- Im folgenden Beispiel weisen wir den drei Buttons den zuvor definierten Style zu.
  - Auf den drei Buttons werden dann alle im Style definierten Eigenschaftswerte entsprechend gesetzt.

```
<Button Style="{StaticResource btnStyle}">Erster Button</Button>
<Button Style="{StaticResource btnStyle}">Zweiter Button</Button>
<Button Style="{StaticResource btnStyle}">Dritter Button</Button>
```

Typisierte Styles

- Ein Style kann festlegen, für welchen Typ von Elementen er Eigenschaften definiert.
  - Dazu wird der sog. `TargetType` entsprechend gesetzt.

- Ist der `TargetType` gesetzt, müssen die Namen der zu setzenden Eigenschaften in den Settern nicht mehr voll qualifiziert werden.

```
<Style x:Key="btnStyle" TargetType="Button">
    <Setter Property="FontSize" Value="20" />
    <Setter Property="Margin" Value="5" />
    <Setter Property="Background" Value="LightBlue" />
</Style>
```

- Wird bei solchen typisierten Styles zudem noch der Schlüsselname der Ressource weggelassen, wird der Style automatisch auf alle Objekte der Klasse `TargetType` angewandt.

Trigger

- Styles sind rein statisch.
  - Die Attributwerte eines Styles werden einmal festgelegt und ändern sich zur Laufzeit nicht.

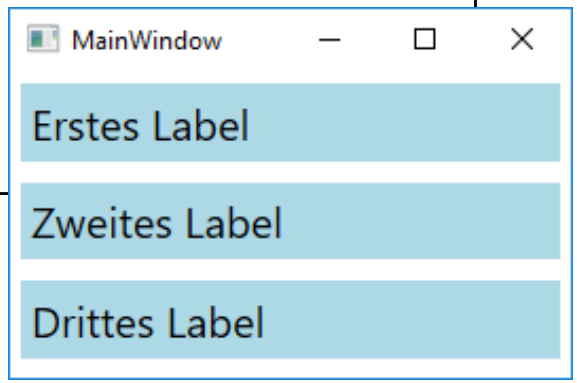
- Häufig sollen aber zur Laufzeit Aktionen ausgeführt werden, wenn bestimmte Bedingungen eintreten.
  - Objekte vom Typ **`Trigger`** können auf Ereignisse, Eigenschaftsänderungen usw. reagieren und dann dynamisch Eigenschaften ändern.

- Es existieren unterschiedliche Arten von Triggern.
  - Trigger → Reagiert auf geänderte Eigenschaften.
  - MultiTrigger → Reagiert auf mehrere geänderte Eigenschaften.
  - EventTrigger → Reagiert auf Routed Events.

Beispiel

- Wir wollen uns ein Beispiel ansehen.
  - Wir erzeugen drei Labels und definieren dazu einen Style, der das Aussehen festlegt.

```
<Window.Resources>
   <Style TargetType="Label">
      <Setter Property="FontSize" Value="20" />
      <Setter Property="Margin" Value="5" />
      <Setter Property="Background" Value="LightBlue" />
   </Style>
</Window.Resources>
<StackPanel>
   <Label>Erstes Label</Label>
   <Label>Zweites Label</Label>
   <Label>Drittes Label</Label>
</StackPanel>
```

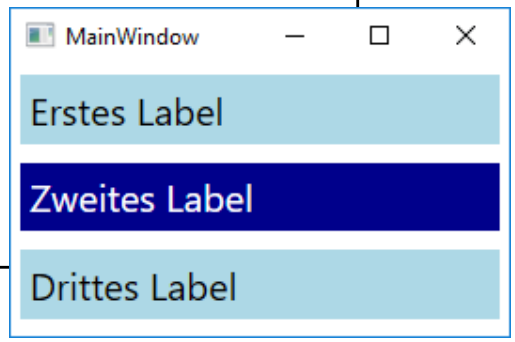


Trigger

- Wir wollen nun einen Trigger einsetzen, der auf eine geänderte Eigenschaft reagiert.
  - Die Farben eines Labels soll sich genau dann ändern, wenn die Maus über diesem Label steht.

- In dem Style wird dafür der Abschnitt Style.Triggers eingeführt.
  - Ein Label verfügt über die Eigenschaft IsMouseOver.
  - Erhält diese den Wert True, sollen die Farben geändert werden.

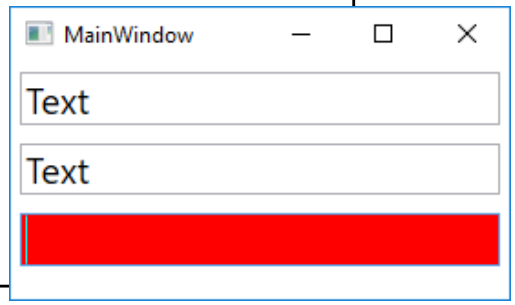
```
<Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
        <Trigger.Setters>
            <Setter Property="Background" Value="DarkBlue" />
            <Setter Property="Foreground" Value="White" />
        </Trigger.Setters>
    </Trigger>
</Style.Triggers>
```



MultiTrigger

- Mit Hilfe von Multitriggern kann sogar auf mehrere Eigenschaftsänderungen gleichzeitig reagiert werden.
  - Man kann z.B. die Hintergrundfarbe einer Textbox auf Rot einstellen, wenn der Inhalt leer ist und diese den Eingabefocus besitzt.

```
<Window.Resources>
    <Style TargetType="TextBox">
        <Setter Property="FontSize" Value="20" />
        <Setter Property="Margin" Value="5" />
        <Style.Triggers>
            <MultiTrigger>
                <MultiTrigger.Conditions>
                    <Condition Property="IsFocused" Value="True" />
                    <Condition Property="Text" Value="" />
                </MultiTrigger.Conditions>
                <MultiTrigger.Setters>
                    <Setter Property="Background" Value="Red" />
                </MultiTrigger.Setters>
            </MultiTrigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
```



Templates

- Mit Hilfe von Styles kann mehreren Elementen ein identisches Layout verliehen werden.
  - Dabei wird auch Code eingespart.
- Die grundlegende optische Darstellung lässt sich dabei allerdings nicht ändern.
  - Ein Button wird z.B. immer als Rechteck dargestellt.
  - Durch einen Style lässt sich dies nicht ändern.
- **Templates (engl. für Vorlagen)** gehen hier noch einen Schritt weiter.
  - Template gestatten die individuelle Gestaltung eines Controls mit Hilfe von eigenem XAML-Code.
  - Die elementare Funktionsweise des Controls wird dabei jedoch nicht verändert.
- Templates können aber noch mehr.
  - Auch die Darstellung von einzelnen Daten kann dadurch angepasst werden.
  - Hierzu dienen die sog. ItemsPanelTemplates, bzw DataTemplates.
  - Diese werden wir uns allerdings erst in der nächsten Vorlesung ansehen.

Control Template

- Alle Steuerelemente, die von der Klasse `Control` ableiten, können ihr Layout über ein sog. `ControlTemplate` anpassen.
  - Ein `ControlTemplate` wird ebenfalls als Ressource definiert.

- Über die Eigenschaft Template kann einem Steuerelement ein zuvor definiertes Template zugewiesen werden.
  - Das Template wird wieder dann wieder als `StaticResource` herangezogen.

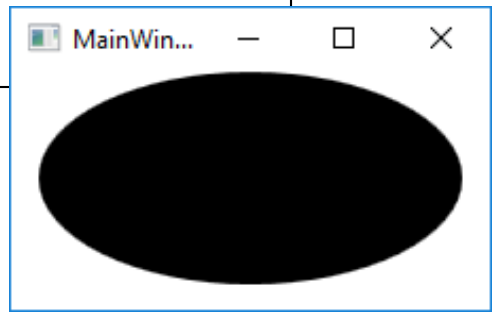
```
<Window.Resources>
    <ControlTemplate x:Key="btnTemplate" TargetType="Button">
        <!-- Hie kommt das Template -->
    </ControlTemplate>
</Window.Resources>
<StackPanel>
    <Button Template="{StaticResource btnTemplate}">Ein Button</Button>
</StackPanel>
```

Beispiel

- Im Folgenden wollen wir ein neues ControlTemplate für einen Button definieren.
  - Der Button soll durch das neue Template rund erscheinen.

- Wir beginnen damit, dass wir in einem Grid eine Ellipse definieren.
  - Dadurch wird die Ellipse mittig an der Stelle des Buttons zentriert.

```
<ControlTemplate x:Key="btnTemplate" TargetType="Button">
   <Grid>
      <Ellipse Width="200" Height="100" Fill="Black" />
   </Grid>
</ControlTemplate>
```

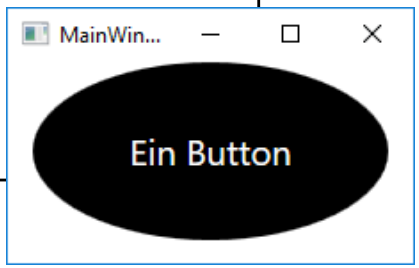


Content Presenter

- Der Button erscheint nun schon als Ellipse.
  - Allerdings fehlt die Beschriftung des Buttons.

- Die Eigenschaft Content des Buttons kann mit Hilfe des Elements `ContentPresenter` in das Template eingefügt werden.
  - Wir benutzen dazu einen `TextBlock`, der den Text mittig und in weißer Farbe darstellt.

```
<ControlTemplate x:Key="btnTemplate" TargetType="Button">
   <Grid>
     <Ellipse Width="200" Height="100" Fill="Black" />
     <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
        FontSize="20" Foreground="White">
        <ContentPresenter Content="{TemplateBinding Content}" />
     </TextBlock>
   </Grid>
</ControlTemplate>
```

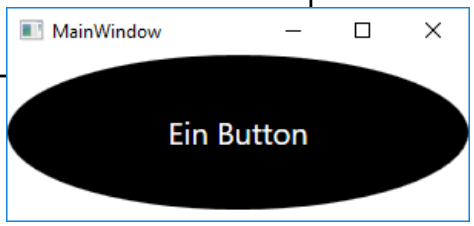


Template Binding

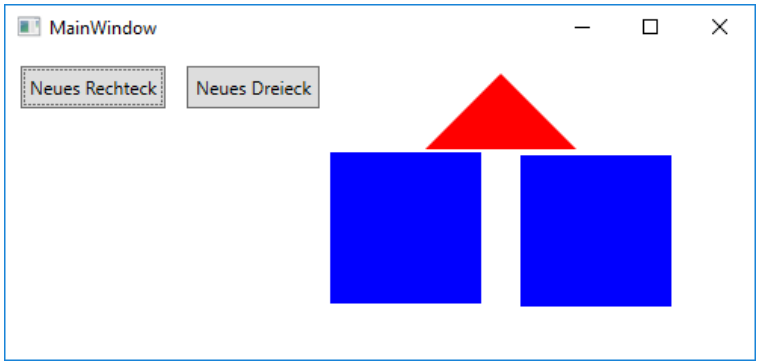
- Zu guter Letzt wollen wir noch dafür sorgen, dass der Button auch die selbe Breite bekommt, wie das Original.
  - Wir können dies mit Hilfe von DataBinding erreichen.

- Mit Hilfe der Markup Erweiterung `TemplateBinding` können wir einen Wert des Templates an die Eigenschaft des Buttons binden.

```
<ControlTemplate x:Key="btnTemplate" TargetType="Button">
   <Grid>
     <Ellipse Width="{TemplateBinding Width}" Height="100" Fill="Black" />
     <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
        FontSize="20" Foreground="White">
        <ContentPresenter Content="{TemplateBinding Content}" />
     </TextBlock>
   </Grid>
</ControlTemplate>
```



Drag und Drop



- Mit Hilfe von ControlTemplates lassen sich viele weitere interessante Ideen umsetzen.
  - Wir wollen ein kleines Zeichenprogramm erstellen.

- Über Buttons sollen Rechtecke oder Dreiecke erzeugt werden.
  - Dazu benötigen wir Event-Handler im Code Behind.
  - Diese erzeugten Elemente soll man mit Hilfe von **Drag-and-Drop** verschieben können.

- Dabei hilft und die Klasse **Thumb**.
  - Objekte der Klasse stellen diejenigen Ereignisse bereit, um auf Drag-und-Drop zu reagieren.
  - Jedem Thumb kann über ein ControlTemplate dabei ein individuelles Aussehen verpasst werden.

Elemente erzeugen

- Wir legen zwei ControlTemplates als Ressourcen an.
  - Diese repräsentieren ein blaues Rechteck, bzw. ein rotes Dreieck:

```
<ControlTemplate x:Key="r"><Rectangle Width="100" Height="100" Fill="Blue" />
</ControlTemplate>
<ControlTemplate x:Key="t"><Polygon Points="0,100 50,50 100,100" Fill="Red" />
</ControlTemplate>
```

- Im Code Behind erstellen wir eine Methode, um neue Thumb-Objekte zu erzeugen.

```
private void CreateNew(string template_name)
{
    var b = new Thumb();
    b.Template = (ControlTemplate)Resources[template_name];
    Canvas.SetTop(b, 100);
    Canvas.SetLeft(b, 100);
    b.DragDelta += Thumb_DragDelta;
    canvas.Children.Add(b);
}
```

Buttons verknüpfen

- Zu guter Letzt müssen noch zwei Buttons mit EventHandlern verknüpft werden.

```
<Canvas Width="640" Height="480" x:Name="canvas">
    <Button Click="Neues_Rechteck">Neues Rechteck</Button>
    <Button Click="Neues_Dreieck">Neues Dreieck</Button>
</Canvas>
```

- In den Event Handlern wird lediglich die Methode zur Erzeugung der Elemente aufgerufen.

```
private void Neues_Rechteck(object sender, RoutedEventArgs e)
{
    CreateNew("r");
}

private void Neues_Dreieck(object sender, RoutedEventArgs e)
{
    CreateNew("t");
}
```

Wir haben heute gelernt…

- Wie logische Ressourcen definiert werden.
- Was Styles sind und wie man diese anwendet.
- Wie man mit Hilfe von Triggern Dynamik in Styles erzeugt.
- Wie man mit Hilfe von ControlTemplates das Aussehen von Controls verändern kann.
- Wie man Drag-and-Drop mit Hilfe der Klasse Thumb umsetzt.