

Die wichtigsten Controls

Nach einem ersten Rundflug über das Erstellen von Apps wollen wir nun endlich zu den einzelnen Oberflächen-Controls kommen, die Sie zwar in der einen oder anderen Form bereits von WPF oder den Windows Forms her kennen, die jedoch teilweise ein etwas anderes Verhalten oder Aussehen aufweisen, was nicht zuletzt auch den Touch-Eingabemöglichkeiten unter Windows 8 bzw. 10 geschuldet ist.

HINWEIS: Bitte erwarten Sie an dieser Stelle nicht, dass wir alle Controls in epischer Breite behandeln. Dafür ist die Online-Hilfe viel besser geeignet. Wir setzen an dieser Stelle voraus, dass Sie bereits mit den WPF-Controls in grundlegender Form vertraut sind.

20.1 Einfache WinRT-Controls

Zunächst beschäftigen wir uns mit einigen grundlegenden Controls, auf die wohl keine Anwendung verzichten kann.

HINWEIS: Die einzelnen Controls müssen Sie in eines der Layout-Controls (nur diese können mehrere Controls enthalten) einfügen. Bei den vorliegenden Seiten (*BlankPage*) finden Sie bereits ein zentrales *Grid* vor, das diese Aufgabe übernimmt. Weitere Informationen zu den Layout-Controls finden ab Seite 987.

20.1.1 TextBlock, RichTextBlock

Für die mehr oder weniger statische Textdarstellung eignen sich die Controls *TextBlock* bzw. *RichTextBlock*. Letzteres bietet zusätzlich die Möglichkeit, neben diverse Formatierungen (Schriftgröße, -farbe etc.) auch Grafiken darzustellen. Der *TextBlock* ist das Pendant bzw. der Ersatz für das bekannte WPF-Label-Control.

Einige kleine Beispiele sollen die Verwendung beider Controls demonstrieren.

BEISPIEL 20.1: Einfache Textausgabe mit vordefiniertem Style

```

XAML
...
<TextBlock Text="TextBlock einfach" Style="{StaticResource BasicTextStyle}" />
...

```

BEISPIEL 20.2: *TextBlock* mit komplexerer Formatierung

Dieser *TextBlock* besteht aus einzelnen *Run*-Elementen, die wiederum unterschiedliche Inhalte, Formatierungen etc. aufweisen können. Der Vorteil: auf die einzelnen *Run*-Elemente kann auch per Name und damit zur Laufzeit per Code zugegriffen werden.

```
<TextBlock>
```

Dieses *Run*-Element ist an eine Eigenschaft der aktuellen Page gekoppelt:

```

<Run FontWeight="Bold" FontSize="24" Text="{Binding EineEigenschaft,
  ElementName=pageRoot}" />

```

Auch Zeilenumbrüche sind möglich:

```
<LineBreak/>
```

Unterschiedliche Formatierungen:

```

<Run FontWeight="Bold" FontSize="24"
  Text="TextBlock mit Formatierung " />
<Run FontStyle="Italic" Foreground="Blue" Text="Es geht auch so!" />

```

Auch komplexe Füllpinsel lassen sich definieren:

```

<Run FontWeight="Bold" FontSize="35" Text="LinearGradientBrush">
  <Run.Foreground>
    <LinearGradientBrush>
      <GradientStop Color="Red" Offset="0.25" />
      <GradientStop Color="Green" Offset="0.5" />
    </LinearGradientBrush>
  </Run.Foreground>
</Run>
</TextBlock>

```

Ergebnis Die Laufzeitanzeige:

Wert der Eigenschaft

TextBlock mit Formatierung *Es geht auch so!* **LinearGradientBrush**

Im Gegensatz zum *TextBlock* bestehen beim *RichTextBlock* die einzelnen Absätze aus *Paragraph*-Elementen, diese wiederum können folgende Elemente enthalten: *Inline*, *InlineUIContainer*, *Run*, *Span*, *Bold*, *Italic*, *Underline*, *LineBreak*.

Einige Beispiele zeigen die Verwendung der einzelnen Elemente.

BEISPIEL 20.3: Verwendung *RichTextBlock***XAML**

...

```
<RichTextBlock Name="rtb1" Margin="0,10,0,5" >
  <Paragraph FontSize="10">Absatz 1, Absatz 1, Absatz 1, Absatz 1, Absatz 1,
    Absatz 1, Absatz 1, Absatz 1, Absatz 1</Paragraph>
```

Absatz mit erweiterten Formatierungen:

```
<Paragraph FontSize="24" FontStretch="UltraExpanded" LineHeight="50"
  CharacterSpacing="100" >Absatz 2, Absatz 2, Absatz 2, Absatz 2,
  Absatz 2, Absatz 2, Absatz 2, Absatz 2, Absatz 2, Absatz 2,
  Absatz 2</Paragraph>
```

Zeilenumbruch:

```
<Paragraph><LineBreak/></Paragraph>
```

Schriftauszeichnungen festlegen:

```
<Paragraph>Noch ein <Underline>Absatz</Underline>, aber diesmal in
  <Bold>fett</Bold>
</Paragraph>
<Paragraph>Absatz mit Control
```

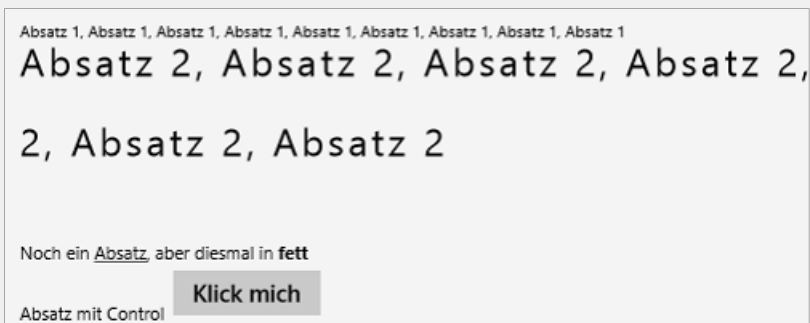
Mittels *InlineUIContainer* wird ein *Button* im Text platziert:

```
<InlineUIContainer>
  <Button Click="Button_Click_1" Name="btn1">
    Klick mich
  </Button>
</InlineUIContainer>
</Paragraph>
</RichTextBlock>
```

...

Ergebnis

Die Laufzeitanzeige:



Die Schaltfläche ist natürlich auch funktionsfähig, auf das Abdrucken der Ereignisprozedur haben wir verzichtet.

Über das Schachteln von *Paragraph*-, *Run*- etc. Elementen können Sie auch zur Laufzeit Texte im *RichTextBlock* erzeugen. Entgegen der Erwartungen müssen Sie diese Elemente jedoch der Blocks-Auflistung übergeben.

BEISPIEL 20.4: Zeilen zur Laufzeit einfügen

```
C# using Windows.UI.Xaml.Documents;
...
    Run run = new Run() { Text = "Text zur Laufzeit! Text zur Laufzeit!" };
    Paragraph para = new Paragraph();
    para.Inlines.Add(run);
    para.Inlines.Add(new LineBreak());
    para.Inlines.Add(new Run() { Text = "Noch mehr Text" });
    rtb2.Blocks.Add(para);
...
```

HINWEIS: Das *Hyperlink*-Element steht Ihnen hier, im Gegensatz zu WPF, nicht zur Verfügung, verwenden Sie stattdessen ein *InlineUIContainer*-Element und fügen Sie darin einen *HyperlinkButton* ein.

20.1.2 Button, HyperlinkButton, RepeatButton

Kommen wir nun zu den etwas "aktiveren" Steuerelementen. An dieser Stelle fassen wir mit *Button*, *HyperLinkButton* und *RepeatButton* drei recht ähnliche Controls zusammen. Allen gemein ist die Aufgabe, als Reaktion auf ein externes Ereignis eine Aktion auszuführen. Die Art des Ereignisses bestimmen Sie mit der Eigenschaft *ClickMode*:

- *Release*,
das Ereignis tritt **nach** dem Klicken (Maustaste, Finger, Leertaste) auf.
- *Press*,
das Ereignis tritt **beim** Klicken (Maustaste, Finger, Leertaste) auf.
- *Hover*,
das Ereignis tritt ein, wenn Maustaste oder Finger über das Control bewegt werden.

BEISPIEL 20.5: Verwendung von Schaltflächen

```
XAML ...
    <Button>Einfacher Button</Button>

Schaltflächen können aus mehreren Controls bestehen:

    <Button>
        <StackPanel>
            <Image Source="Images/mycomputer.png" Stretch="None"/>
            <TextBlock HorizontalAlignment="Center">Button mit Grafik</TextBlock>
        </StackPanel>
    </Button>
```

BEISPIEL 20.5: Verwendung von Schaltflächen

```

<HyperlinkButton Click="HyperlinkButton_Click_1">
    Ein HyperlinkButton
</HyperlinkButton>
<RepeatButton ClickMode="Hover" Click="RepeatButton_Click_1" >
    RepeatButton - Hover</RepeatButton>
<RepeatButton ClickMode="Press" Click="RepeatButton_Click_1" >
    RepeatButton - Press</RepeatButton>
<RepeatButton ClickMode="Release" Click="RepeatButton_Click_1" >
    RepeatButton - Release</RepeatButton>

```

...

C#

...

Die Funktionalität des *HyperlinkButton* muss erst programmiert werden:

```

async private void HyperlinkButton_Click_1(object sender, RoutedEventArgs e)
{
    Uri uri = new Uri(@"http://www.doko-buch.de");
    bool success = await Windows.System.Launcher.LaunchUriAsync(uri);
}

```

Die Reaktion der *RepeatButton*-Controls:

```

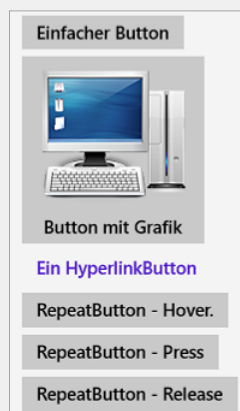
private void RepeatButton_Click_1(object sender, RoutedEventArgs e)
{
    (sender as RepeatButton).Content += ".";
}

```

...

Ergebnis

Die Laufzeitanzeige:



HINWEIS: Es wird Ihnen sicher am Anfang etwas fremd sein, aber die meisten Ereignisse müssen Sie mit dem Schlüsselwort *async* deklarieren, denken Sie dabei an "Fast and Fluid" (siehe Seite 921).

20.1.3 CheckBox, RadioButton, ToggleButton, ToggleSwitch

Neben den beiden bekannten Controls *CheckBox* und *RadioButton*, die es dieser Form schon bei den Windows Forms gab, werden mit WinRT auch zwei neue Vertreter dieser Gattung eingeführt:

- *ToggleButton*
- *ToggleSwitch*

Gemein ist beiden, dass zwischen zwei Zuständen "geschaltet" werden kann. Beim *ToggleButton* wird dadurch die Eigenschaft *IsChecked* beeinflusst, beim *ToggleSwitch* die Eigenschaft *IsOn*.

BEISPIEL 20.6: Verwendung von *CheckBox*, *RadioButton*, *ToggleButton*, *ToggleSwitch*

XAML

```
...
<StackPanel Grid.Row="1" Margin="120,0,120,0">
```

Zunächst ein paar *CheckBox*-Controls definieren:

```
<StackPanel Orientation="Horizontal" Margin="0,50,0,0">
  <CheckBox Margin="0,0,15,0">Option 1</CheckBox>
  <CheckBox IsChecked="True" Margin="0,0,15,0">Option 2</CheckBox>
  <CheckBox Margin="0,0,15,0">Option 3</CheckBox>
  <CheckBox Margin="0,0,15,0">Option 4</CheckBox>
```

Statt einer Beschriftung sind auch andere Inhalte denkbar:

```
<CheckBox Margin="0,0,15,0" >
  <Image Source="Images/cookie.png" Stretch="None" />
</CheckBox>
</StackPanel>
<StackPanel Orientation="Horizontal" Margin="0,50,0,0">
```

Die zwei Zustände des *ToggleButtons* (auch hier könnten Sie Grafiken einblenden):

```
<ToggleButton Margin="0,0,50,0">Das ist ein ToogleButton</ToggleButton>
<ToggleButton IsChecked="True">Das ist ein ToogleButton</ToggleButton>
</StackPanel>
<StackPanel Orientation="Horizontal" Margin="0,50,0,0">
```

Der *ToggleSwitch* in Aktion:

```
<ToggleSwitch Margin="0,0,50,0" IsOn="True">
  Das ist ein ToggleSwitch</ToggleSwitch>
<ToggleSwitch >Das ist ein ToggleSwitch</ToggleSwitch>
```

Alternativ kann den beiden Zuständen des "Schalters" auch unterschiedlicher Content zugewiesen werden (siehe Laufzeitansicht beim Klick auf den *ToggleSwitch*):

```
<ToggleSwitch>
  ToggleSwitch mit wechselnder Grafik
  <ToggleSwitch.OnContent>
    <Image Source="Images/bulbon.png"/>
  </ToggleSwitch.OnContent>
```

BEISPIEL 20.6: Verwendung von *CheckBox*, *RadioButton*, *ToggleButton*, *ToggleSwitch*

```

        <ToggleSwitch.OffContent>
            <Image Source="Images/bulboff.png"/>
        </ToggleSwitch.OffContent>
    </ToggleSwitch>
</StackPanel>

```

Noch ein paar *RadioButton*-Controls, diese müssen zum Gruppieren in einem übergeordneten Control zusammengefasst werden:

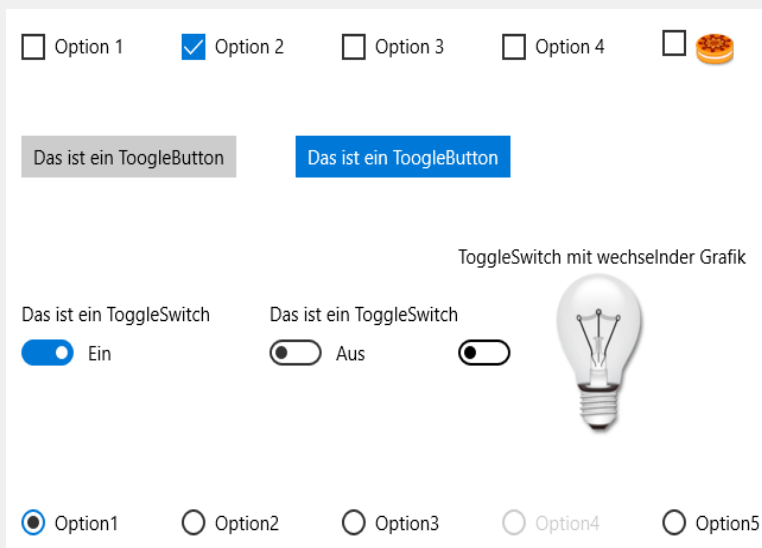
```

<StackPanel Orientation="Horizontal" Margin="0,50,0,0">
    <RadioButton IsChecked="True" Margin="0,0,15,0">Option1</RadioButton>
    <RadioButton Margin="0,0,15,0">Option2</RadioButton>
    <RadioButton Margin="0,0,15,0">Option3</RadioButton>
    <RadioButton IsEnabled="False" Margin="0,0,15,0">Option4</RadioButton>
    <RadioButton Margin="0,0,15,0">Option5</RadioButton>
</StackPanel>
</StackPanel>
...

```

Ergebnis

Die Laufzeitanzeige:

**20.1.4 TextBox, PasswordBox, RichEditBox**

Für die Eingabe von Werten stehen Ihnen neben der bekannten *TextBox* noch die *RichEditBox* und der "Spezialist" *PasswordBox* zur Verfügung. Die Bedeutung der Controls dürfte sich aus dem Namen ergeben, einige Beispiele sollen die Verwendung demonstrieren.

BEISPIEL 20.7: *TextBox*, *PasswordBox*, *RichEditBox***XAML**

```
...
<StackPanel Grid.Row="1" Margin="120,0,120,0">
    <TextBlock Text="TextBox einfach" Style="{StaticResource BasicTextStyle}"
        Margin="0,10,0,5" />

```

Eine simple *TextBox* erzeugen:

```
<TextBox Name="TextBox1" Text="Eingabe" />
<Button Content="Alles Markieren" Click="Button_Click_3"/>

```

Eine mehrzeilige *TextBox* mit automatischem Zeilenumbruch erzeugen:

```
<TextBox Text="Eingabe" Opacity="0.5" Height="100" BorderThickness="0"
    Margin="0,10,0,5" AcceptsReturn="True" TextWrapping="Wrap" />

```

Passwortheingabe leichtgemacht:

```
<TextBlock Text="PasswordBox" Style="{StaticResource BasicTextStyle}"
    Margin="0,10,0,5" />
<PasswordBox Password="abcdefg" PasswordChar="?" />
<TextBlock Text="RichTextBox" Style="{StaticResource BasicTextStyle}"
    Margin="0,10,0,5" />

```

Diese *RichEditBox* füllen wir zur Laufzeit:

```
<RichEditBox Height="200" Name="reb1"/>

```

Einige Schaltflächen, mit denen wir die *RichEditBox* manipulieren:

```
<StackPanel Orientation="Horizontal">
    <Button Content="Laden" Click="Button_Click_1"/>
    <Button Content="Speichern" Click="Button_Click_4"/>
    <Button Content="Text Erzeugen" Click="Button_Click_2"/>
</StackPanel>

```

...

C#

...

Wir laden eine Text- oder Richtext-Datei in die *RichEditBox*:

```
async private void Button_Click_1(object sender, RoutedEventArgs e)
{

```

Das Öffnen der "Dateidialoge" – jetzt *FileOpenPicker* – erfordert die Vollbildansicht:

```
if (ApplicationView.Value != ApplicationViewState.Snapped)
{
    Windows.Storage.Pickers.FileOpenPicker open = new
        Windows.Storage.Pickers.FileOpenPicker();
    open.SuggestedStartLocation =
        Windows.Storage.Pickers.PickerLocationId.DocumentsLibrary;
    open.FileTypeFilter.Add(".rtf");

```


BEISPIEL 20.8: *TextBox*, *PasswordBox*, *RichEditBox*

C#

```
open.FileTypeFilter.Add(".txt");
Windows.Storage.StorageFile file = await open.PickSingleFileAsync();
```

Wurde eine Datei ausgewählt ...

```
if (file != null)
{
```

... können wir diese in die *RichEditBox* laden:

```
    Windows.Storage.Streams.IRandomAccessStream strm =
        await file.OpenAsync(Windows.Storage.FileAccessMode.Read);
    reb1.Document.LoadFromStream(Windows.UI.Text.TextSetOptions.FormatRtf,
                                strm);
}
}
```

Alternativ können Sie die Inhalte der *RichEditBox* auch per Code erzeugen:

```
async private void Button_Click_2(object sender, RoutedEventArgs e)
{
```

Wie Sie sehen, arbeiten wir quasi mit der Einfügemarke, die wir verschieben:

```
    reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
```

Nachfolgend können wir Texte und Formatierungen zuweisen:

```
    reb1.Document.Selection.SetText(TextSetOptions.None, "Überschrift");
    reb1.Document.Selection.CharacterFormat.BackgroundColor = Colors.White;
    reb1.Document.Selection.CharacterFormat.Size = 18;
```

```
    reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
    reb1.Document.Selection.SetText(TextSetOptions.None, "Hallo in Grün");
    reb1.Document.Selection.CharacterFormat.BackgroundColor = Colors.Green;
    reb1.Document.Selection.CharacterFormat.Size = 10;
```

```
    reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
    reb1.Document.Selection.SetText(TextSetOptions.None, "Hallo in Rot");
    reb1.Document.Selection.CharacterFormat.BackgroundColor = Colors.Red;
```

Einfügen eines Hyperlinks:

```
    reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
```

Zunächst der Beschriftungstext:

```
    reb1.Document.Selection.SetText(TextSetOptions.None, "\nEin Hyperlink");
    reb1.Document.Selection.CharacterFormat.BackgroundColor = Colors.White;
    reb1.Document.Selection.CharacterFormat.ForegroundColor = Colors.Green;
```

BEISPIEL 20.8: *TextBox*, *PasswordBox*, *RichEditBox*

Dann die Sprungadresse:

```
reb1.Document.Selection.Link = "\"http://doko-buch.de\"";
```

Hier fügen wir eine Grafik ein, die der Nutzer per *FileOpenPicker* auf dem System auswählt:

```
Windows.Storage.Pickers.FileOpenPicker open = new
    Windows.Storage.Pickers.FileOpenPicker();
open.SuggestedStartLocation =
    Windows.Storage.Pickers.PickerLocationId.PicturesLibrary;
open.ViewMode = Windows.Storage.Pickers.PickerViewMode.Thumbnail;
open.FileTypeFilter.Clear();
open.FileTypeFilter.Add(".bmp");
open.FileTypeFilter.Add(".png");
open.FileTypeFilter.Add(".jpeg");
open.FileTypeFilter.Add(".jpg");
Windows.Storage.StorageFile file = await open.PickSingleFileAsync();
Windows.Storage.Streams.IRandomAccessStream fileStream = await
    file.OpenAsync(Windows.Storage.FileAccessMode.Read);

reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
reb1.Document.Selection.InsertImage(50, 50, 50,
    VerticalCharacterAlignment.Top, "test", fileStream);
}
```

Die Inhalte der *RichEditBox* speichern:

```
async private void Button_Click_4(object sender, RoutedEventArgs e)
{
    if (ApplicationView.Value != ApplicationViewState.Snapped)
    {
        Windows.Storage.Pickers.FileSavePicker save = new
            Windows.Storage.Pickers.FileSavePicker();
        save.SuggestedStartLocation =
            Windows.Storage.Pickers.PickerLocationId.DocumentsLibrary;
        save.FileTypeChoices.Add("Rich Text Format", new String[] { ".rtf" });
        Windows.Storage.StorageFile file = await save.PickSaveFileAsync();
        if (file != null)
        {
            Windows.Storage.Streams.IRandomAccessStream strm =
                await file.OpenAsync(Windows.Storage.FileAccessMode.ReadWrite);
            reb1.Document.SaveToStream(TextGetOptions.FormatRtf, strm);
            strm.Dispose();
        }
    }
}
```

...

BEISPIEL 20.8: TextBox, PasswordBox, RichEditBox

Ergebnis

RichTextBox

```

<common:LayoutAwarePage
  x:Name="pageRoot"
  x:Class="Steuerelemente.TextBox_RichTextBox"
  DataContext="{Binding DefaultViewModel, RelativeSource={RelativeSource Self}}"
  IsTabStop="false"
  mc:Ignorable="d">
  |
  <Page.Resources>

    <!-- TODO: Delete this line if the key AppName is declared in App.xaml -->
    <x:String x:Key="AppName">TextBox und RichEditBox</x:String>
  </Page.Resources>

```

Laden

Speichern

Text Erzeugen

20.1.5 Image

Die Funktion dieses Controls dürfte sich wohl auf den ersten Blick erschließen, die per *Source*-Eigenschaft zugewiesenen Grafiken können in verschiedenen Ansichtsmodi (Eigenschaft *Stretch*) dargestellt werden.

HINWEIS: In .NET-Projekten wird die Darstellung von SVG-Grafiken **nicht** unterstützt. Sie müssen diese vorher mit einem Konverter in XAML-Grafiken umwandeln!

Das folgende Beispiel zeigt Ihnen neben der trivialen Anzeige von Grafiken auch die Möglichkeit, wie Sie diese per Gestensteuerung zur Laufzeit manipulieren können.

BEISPIEL 20.9: Image-Control und Gestensteuerung

XAML

...

Ein Ressourcen-Image in Originalgröße anzeigen:

```
<Image Source="Images/mycomputer.png" Stretch="None" Margin="0,0,0,20"/>
```

Das Bild wird in diesem Fall proportional auf 50x50 Pixel skaliert:

```
<Image Source="Images/mycomputer.png" Width="50" Height="50" Stretch="Uniform"
  Margin="0,0,0,20"/>
```

Das in den folgenden Canvas eingefügte *Image* wird für die Reaktion auf alle Manipulationsarten (Drehen, Vergrößern, Verschieben) konfiguriert. Die eigentliche Reaktion erfolgt im zugeordneten Eventhandler:

```

<Canvas Height="400" Background="AliceBlue">
  <Image Name="Image1" Source="Images/bulbon.png" ManipulationMode="All"
    ManipulationDelta="Image_ManipulationDelta"
    RenderTransformOrigin="0.5, 0.5">
    <Image.RenderTransform>
      <CompositeTransform/>

```

BEISPIEL 20.9: Image-Control und Gestensteuerung

```

        </Image.RenderTransform>
    </Image>
</Canvas>
...

```

C# Wie versprochen, müssen wir uns noch um die eigentliche Manipulation kümmern, wir hatten lediglich die Reaktion darauf "freigeschaltet":

```
private void Image_ManipulationDelta(object sender, ManipulationDeltaRoutedEventArgs e)
{
```

Aktuelle Einstellungen abrufen:

```
    CompositeTransform comptrans = (CompositeTransform)(sender as Image).RenderTransform;
```

Änderungen einrechnen:

```
    comptrans.ScaleX *= e.Delta.Scale;
    comptrans.ScaleY *= e.Delta.Scale;
    comptrans.TranslateX += e.Delta.Translation.X;
    comptrans.TranslateY += e.Delta.Translation.Y;
    comptrans.Rotation += 180.0 / Math.PI * e.Delta.Rotation;
}
```

```

...

```

Ergebnis

Testen Sie die Reaktion der "Glühlampe" auf die bekannten Gesten.



HINWEIS: Wer nicht über die erforderliche Hardware verfügt, der sei an den Tablet-Simulator verwiesen, hier können Sie die Gesten (Drehen, Verschieben, Vergrößern) über die drei entsprechenden Tasten auswählen und per Mausrad simulieren.

20.1.6 ScrollBar, Slider, ProgressBar, ProgressRing

An dieser Stelle fassen wir einige Controls zusammen, die ähnlich gelagerte Aufgaben übernehmen. Während *Scrollbar* und *Slider* für die Eingabe von Werten zwischen einem vorgegebenen *Minimum* und *Maximum* vorgesehen sind, stellt der *ProgressBar* die optische Anzeige eines Wertes dar. Eine Sonderrolle spielt der *ProgressRing*, dieser stellt einen neutralen Fortschritt oder eine Bearbeitung dar, die mit *IsActive* ein- oder ausgeschaltet wird.

Auf einige spezielle Eigenschaften des *Slider*-Controls geht das folgende Beispiel ein:

BEISPIEL 20.10: *Slider*, *ProgressBar*, *ProgressRing*

XAML

```
...
<StackPanel Grid.Row="1" Margin="120,0,120,0">
    <StackPanel Orientation="Horizontal" Margin="0,50,0,50">
```

Ein horizontaler *Slider* mit einem Wertebereich von 1 bis 100:

```
<Slider Name="Slider1" Width="300" Height="35" Minimum="1" Maximum="100"
    Orientation="Horizontal" Value="25" Margin="0,0,50,0" />
```

Zur Anzeige binden wir einfach einen *TextBlock* an den *Value* des *Sliders*:

```
<TextBlock Text="{Binding ElementName=Slider1, Path=Value}"
    Style="{StaticResource HeaderTextStyle}" Width="75"/>
```

Ebenfalls an den *Slider* gebunden, ein *ProgressBar* zur Wertanzeige:

```
<ProgressBar Maximum="100" Value="{Binding ElementName=Slider1, Path=Value}"
    Margin="50,0,0,0" Height="25" Width="200" Foreground="GreenYellow"/>
</StackPanel>
<StackPanel Orientation="Horizontal" Margin="0,50,0,0">
    <TextBlock Style="{StaticResource TitleTextStyle}" >ProgressRing:</TextBlock>
    <CheckBox Name="Check2" Margin="50,0,50,0">IsActive</CheckBox>
```

Ein *Slider* mit einer Schrittweite von 0,1 und einer Anzeigeschrittweite von 0,5:

```
<StackPanel Orientation="Horizontal" Margin="0,25,0,20">
    <Slider Name="Slider2" Width="300" Height="35" Minimum="1" Maximum="5"
        StepFrequency="0.1" SnapsTo="StepValues" TickFrequency="0.5"
        Orientation="Horizontal" Value="25" Margin="0,0,50,0" />
    <TextBlock Text="{Binding ElementName=Slider2, Path=Value}"
        Style="{StaticResource HeaderTextStyle}" />
```

Der regulär zurückgegebene Wert bestimmt sich aus der Eigenschaft *SnapsTo*. Ist diese auf *StepValue* festgelegt, wie im obigen Beispiel, wird der eingestellte Wert mit einer Genauigkeit von 0,1 bestimmt. Bei *TickValues* würden die Werte nur mit einer Schrittweite von 0,5 zurückgegeben werden.

```
</StackPanel>
```

Ein *ProgressRing* zur Fortschrittsanzeige, das Ein-/Ausschalten übernimmt die *CheckBox*:

```
<ProgressRing Height="60" Width="60" IsActive="{Binding IsChecked,
```

BEISPIEL 20.10: *Slider, ProgressBar, ProgressRing*

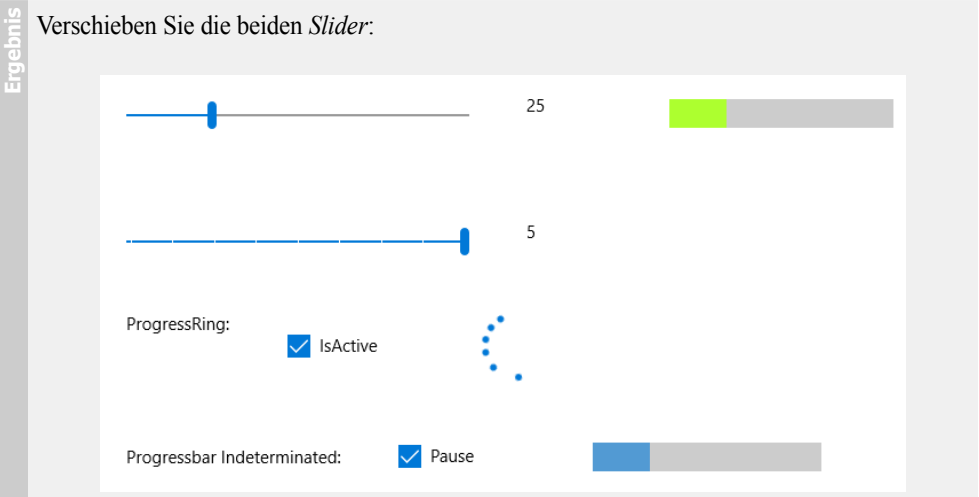
```

        ElementName=Check2}"></ProgressRing>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="0,50,0,0">
        <TextBlock Style="{StaticResource TitleTextStyle}"
            VerticalAlignment="Center">Progressbar Indetermined:</TextBlock>
        <CheckBox Name="Check1" Margin="50,0,50,0">Pause</CheckBox>
    </StackPanel>

    Ein ProgressBar ohne absolute Wertanzeige (nur Animation):

    <ProgressBar Maximum="100" Value="{Binding ElementName=Slider1, Path=Value}"
        IsIndeterminate="True" Height="25" Width="200"
        Foreground="GreenYellow"
        ShowPaused="{Binding IsChecked, ElementName=Check1}"/>
    </StackPanel>
    ...

```



20.1.7 Border, Ellipse, Rectangle

An dieser Stelle erwartet den WPF-Programmierer nichts Neues, die Controls verfügen lediglich über die zusätzlichen Events für die Toucheingabe (*Tapped ...*)

BEISPIEL 20.11: Verwendung *Border*

```

XAML
...
<Border Height="100" Background="AliceBlue" BorderThickness="20,5,10,1"
    BorderBrush="Blue" CornerRadius="0,10,20,50" >
    <TextBlock Foreground="Blue" FontSize="50" HorizontalAlignment="Center"
        VerticalAlignment="Center">Das ist ein Border</TextBlock>
</Border>
...

```

BEISPIEL 20.11: Verwendung *Border*

Ergebnis

Die Laufzeitanzeige:



20.2 Layout-Controls

Wie Sie den bisherigen Beispielen entnehmen konnten, kann in vielen Fällen immer nur ein Control als Content in den Clientbereich, z.B. einer *Page*, eingefügt werden. In diesem Fall können Sie auf den reichen Fundus an Layout-Controls zurückgreifen. Diese bieten nicht nur die Möglichkeit, mehrere Client-Controls aufzunehmen, sondern auch diese geschickt zu platzieren.

20.2.1 Canvas

Der *Canvas* erinnert noch am ehesten an das Erstellen von Windows Forms-Oberflächen, enthaltene Controls werden einfach über Ihre Koordinaten im Clientbereich positioniert.

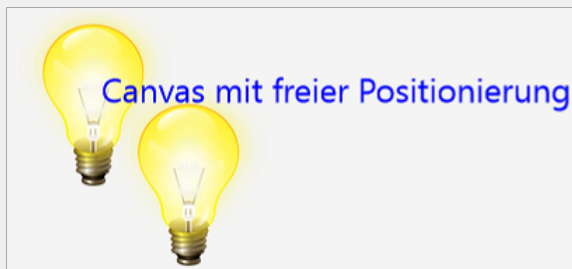
HINWEIS: Beachten Sie, dass die Koordinaten per angehängter Eigenschaft zugewiesen werden (siehe folgende Beispiel).

BEISPIEL 20.12: Verwendung *Canvas*

C#

```
...
<Canvas Height="200" Background="AntiqueWhite">
  <Image Canvas.Left="50" Canvas.Top="5" Source="Images/bulbon.png"/>
  <Image Canvas.Left="118" Canvas.Top="62" Source="Images/bulbon.png"/>
  <TextBlock Foreground="Blue" FontSize="24" Canvas.Left="118" Canvas.Top="51">
    Canvas mit freier Positionierung</TextBlock>
</Canvas>
...
```

Ergebnis



HINWEIS: Möchten Sie ein Control frei über anderen Controls positionieren, fügen Sie es in einen *Canvas* ein, dessen Größe Sie auf 0x0 Pixel reduzieren. Die *Canvas*-Größe hat keine Auswirkungen auf die Positionierbarkeit der enthaltenen Controls.

BEISPIEL 20.13: Frei positionierter *TextBlock*

XAML	<pre> ... <!-- Back button and page title --> <Grid> <Grid.ColumnDefinitions> <ColumnDefinition Width="Auto"/> <ColumnDefinition Width="*/> </Grid.ColumnDefinitions> <Canvas Height="0" Width="0" Grid.Column="0" Grid.Row="0" > <TextBlock Foreground="Blue" FontSize="24" Canvas.Left="0" Canvas.Top="-20">TextBlock mit freier Positionierung</TextBlock> </Canvas> ... </pre>
------	--



20.2.2 StackPanel

Alle enthaltenen Controls werden in der Reihenfolge ihrer Definition "übereinandergestapelt". Standardmäßig erhalten die Controls die Breite des *StackPanels*, Sie müssen also gegebenenfalls die Breite explizit festlegen. Mit der Eigenschaft *Orientation* legen Sie die Stapelrichtung (Horizontal, Vertikal) fest.

20.2.3 ScrollViewer

Kommen Sie mit den verfügbaren Platzverhältnissen nicht klar oder wollen Sie große Grafiken anzeigen, die nicht auf den Bildschirm passen, können Sie einen *ScrollViewer* einsetzen. Kann der enthaltene Content nicht komplett dargestellt werden, blendet der *ScrollViewer* horizontale und/oder vertikale Scrollbars ein, die das Verschieben des Sichtausschnitts ermöglichen.

BEISPIEL 20.14: Verwendung *ScrollViewer*

XAML	<pre> ... <ScrollViewer Height="200" Width="200" HorizontalScrollBarVisibility="Auto" > <Image Source="Images/bulbon.png" Width="400" Height="400" /> </ScrollViewer> ... </pre>
------	--

BEISPIEL 20.14: Verwendung *ScrollView*

Ergebnis

**20.2.4 Grid**

Das wohl wichtigste Layout-Control ist das *Grid*. Ein Blick in den XAML-Code, z.B. einer Standardseite, zeigt recht schnell, dass ohne dieses Control nichts läuft. Wie bei einer Tabelle werden zunächst Zeilen und Spalten definiert. Diese können absolute Angaben (in Pixeln) zu Höhe und Breite enthalten oder auch relative (siehe Beispiel). Alternativ bietet es sich auch an, dass der Inhalt die Höhe bzw. die Breite bestimmt, in diesem Fall wird für *Height* oder *Width* der Wert "Auto" übergeben. Die enthaltenen Controls lassen sich mittels der angehängten Eigenschaften *Grid.Column* und *Grid.Row* im *Grid* positionieren.

BEISPIEL 20.15: Verwendung *Grid*

XAML

...

Zunächst die Maße des *Grids* bestimmen und die Zeilen definieren:

```
<Grid Height="200" Width="500" Background="LightYellow" >
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="3*"/>
    <RowDefinition Height="2*"/>
  </Grid.RowDefinitions>
```

Die Höhe der ersten Zeile wird durch den Inhalt bestimmt, den Rest teilen sich Zeile 2 und 3 im Verhältnis 3 zu 2.

Die Spalten festlegen:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="Auto"/>
  <ColumnDefinition Width="3*"/>
  <ColumnDefinition Width="1*"/>
</Grid.ColumnDefinitions>
```

Die Breite der ersten Spalte wird durch den Inhalt bestimmt, den Rest teilen sich Spalte 2 und 3 im Verhältnis 3 zu 1.

BEISPIEL 20.15: Verwendung *Grid*

Die folgenden drei Rechtecke werden in den Gridzellen positioniert:

```
<Rectangle Grid.Row="0" Grid.Column="0" Width="150" Height="45"
  Fill="BlueViolet" StrokeThickness="5" Stroke="#FF00E01F" >/Rectangle>
<Rectangle Grid.Row="1" Grid.Column="1" Fill="Red" StrokeThickness="5"
  Stroke="#FF00E01F" >/Rectangle>
<Rectangle Grid.Row="1" Grid.Column="2" Grid.RowSpan="2" Fill="Yellow"
  StrokeThickness="5" Stroke="#FF00E01F" >/Rectangle>
```

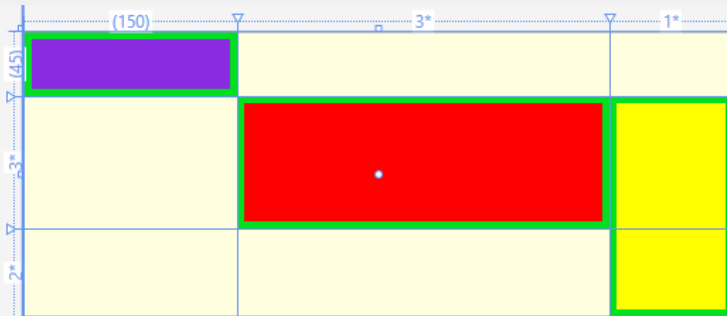
Beachten Sie das dritte Rechteck: Dieses erstreckt sich wegen *Grid.RowSpan="2"* über zwei Zeilen.

```
</Grid>
```

...

Ergebnis

Die Entwurfsansicht:



20.2.5 VariableSizedWrapGrid

Bei diesem erweiterten *WrapGrid* bestimmen Sie zunächst mit *Orientation* die Ausrichtung (Standard *Vertical*) und die Standardhöhe und -breite der enthaltenen Items (*ItemHeight*, *ItemWidth*). Die Eigenschaft *MaximumRowsOrColumns* legt fest, nach welchem Element umgebrochen wird (abhängig von *Orientation*). Die enthaltenen Elemente werden auf das mit *ItemHeight* und *ItemWidth* festgelegte Maß beschnitten, es sei denn, Sie weisen dem Element eine *ColumnSpan* oder *RowSpan* zu (das Element belegt dann mehr als eine Spalte oder Zeile).

BEISPIEL 20.16: Verwendung *VariableSizedWrapGrid*

XAML

VariableSizedWrapGrid mit drei Zeilen und einer Zellgröße von 50x50 Pixeln definieren:

...

```
<VariableSizedWrapGrid Height="160" Background="LavenderBlush" ItemHeight="50"
  ItemWidth="50" MaximumRowsOrColumns="3">
```

Das folgende Rechteck ist kleiner als der "Cell"-Bereich und wird original angezeigt:

```
<Rectangle Fill="PaleGreen" StrokeThickness="5" Stroke="#FF00E01F" >/Rectangle>
```

BEISPIEL 20.16: Verwendung *VariableSizedWrapGrid*

```
<Rectangle Fill="DarkRed" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
```

Ein Rechteck mit doppelter Höhe:

```
<Rectangle Fill="Red" StrokeThickness="5" Stroke="#FF00E01F" Height="100"
  VariableSizedWrapGrid.RowSpan="2" ></Rectangle>
<Rectangle Fill="Green" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
```

Ein Rechteck mit doppelter Breite:

```
<Rectangle Fill="Blue" StrokeThickness="5" Stroke="#FF00E01F" Width="100"
  VariableSizedWrapGrid.ColumnSpan="2" ></Rectangle>
<Rectangle Fill="Yellow" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
<Rectangle Fill="White" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
```

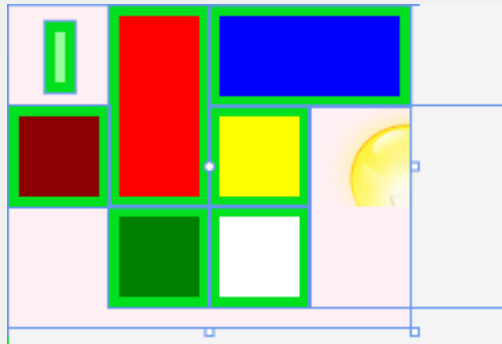
Das Image wird beschnitten, da zu groß:

```
<Image Source="Images/bulbon.png" Height="100" Width="100"/>
</VariableSizedWrapGrid>
```

...

Ergebnis

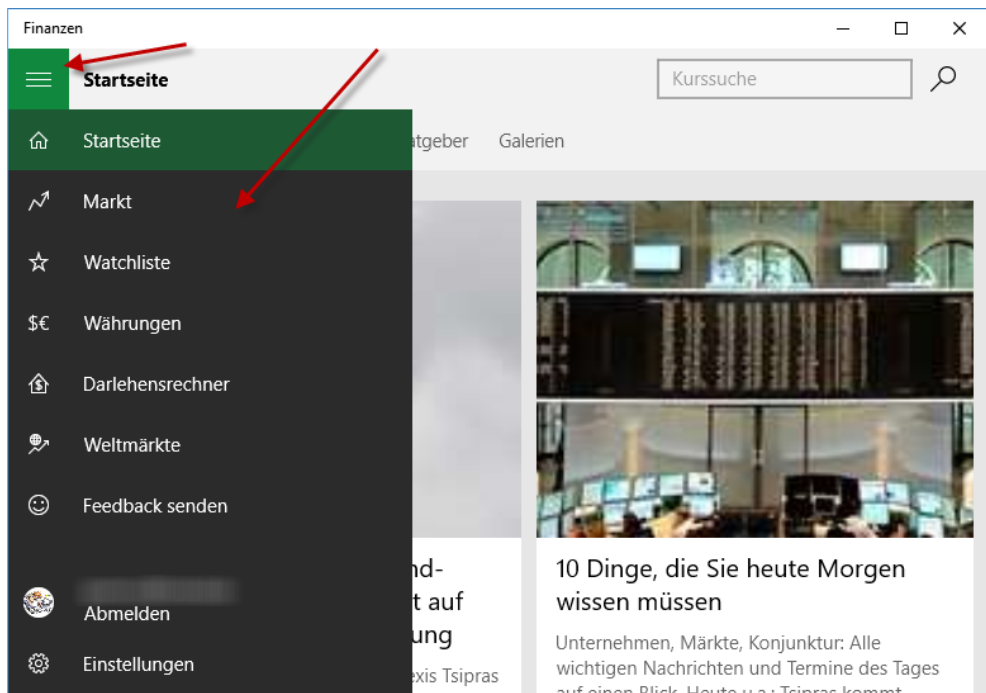
Die Entwurfsansicht:



20.2.6 SplitView

Sicher wird Ihnen eine der neuen Gestaltungsrichtlinien von Windows 10, die bei fast allen mitgelieferten Apps zum Einsatz kommt, nicht verborgen geblieben. Die Rede ist vom aufklappbaren Menü am linken App-Rand, das entweder über dem Content oder neben dem Content der Page eingeblendet wird, bis eine Option gewählt ist. Hierbei handelt es sich um die für Windows-Programmierer neue *SplitView*.

Ein gutes Beispiel ist die Finanz-App von Windows 10, die über eine *SplitView* die einzelnen Aufgaben (*Startseite*, *Markt*, *Watchliste* ...) bereitstellt:



Wir könnten jetzt auch ein komplexes Beispiel entsprechend den Microsoft-Beispielen darstellen, denken aber, dass eine auf das wesentliche reduzierte Lösung besser verständlich ist.

BEISPIEL 20.17: Verwendung *SplitView*

XAML ...

Das umgebende *Grid* aus der *Page*:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Grid.ColumnDefinitions>
```

Wir definieren zwei Spalten (einmal die Spalte mit der Ein-/Ausschaltfläche) und dann der eigentlich sichtbare Bereich:

```
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="*/"/>
  </Grid.ColumnDefinitions>
```

Über den *VisualStateManager* wechseln wir zwischen zwei Modi der *Splitview*: Ist der Bildschirm schmaler als 641 Pixel, wird die Menüauswahl **über** dem Content angezeigt, anderenfalls **links daneben**:

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState x:Name="wideState">
      <VisualState.StateTriggers>
```

BEISPIEL 20.17: Verwendung *SplitView*

```

        <AdaptiveTrigger MinWindowWidth="641" />
    </VisualState.StateTriggers>
    <VisualState.Setters>
        <Setter Target="Splitter.DisplayMode" Value="Inline"/>
    </VisualState.Setters>
</VisualState>
<VisualState x:Name="narrowState">
    <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="0" />
    </VisualState.StateTriggers>
    <VisualState.Setters>
        <Setter Target="Splitter.DisplayMode" Value="Overlay"/>
    </VisualState.Setters>
</VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>

```

Hier die eigentliche *SplitView*:

```
<SplitView x:Name="Splitter" IsPaneOpen="True" Grid.Column="1">
```

Zunächst der optional sichtbare Navigationsbereich:

```

    <SplitView.Pane>
        <RelativePanel Background="{ThemeResource
                                SystemControlBackgroundChromeMediumBrush}">

```

Wir zeigen eine Überschrift und eine *ListBox* an, in der wiederum die einzelnen Menüpunkte dargestellt werden. Der Vorteil dieser Variante: ist der Bildschirm nicht hoch genug, erscheint für die *ListBox* ein *Scrollbar*, die Menüpunkte sind also alle erreichbar.

```

        <TextBlock x:Name="SampleTitle" Text="DOKO-Beispiel" TextWrapping="Wrap"
            Margin="0,10,0,0"/>
        <ListBox x:Name="ListBox1"
            SelectionChanged="ScenarioControl_SelectionChanged"
            SelectionMode="Single" HorizontalAlignment="Left"
            VerticalAlignment="Top"
            RelativePanel.Below="SampleTitle" Margin="0,10,0,0" >
        </ListBox>
    </RelativePanel>
</SplitView.Pane>

```

Kommen wir zum eigentlichen Content-Bereich, in dem später die jeweiligen Ansichten eingeblendet werden sollen:

```
<RelativePanel>
```

Wir nutzen einen *Frame*, so könne wir beliebige Inhalte darstellen (Einzelzeilen):

```

    <Frame x:Name="ScenarioFrame" Margin="0,5,0,0" />
</RelativePanel>

```

BEISPIEL 20.17: Verwendung *SplitView*

```
</SplitView>
```

Was bleibt, ist die Darstellung der Spalte für die Ein-Aus-Schaltfläche:

```
<Border Background="{ThemeResource SystemControlBackgroundChromeMediumBrush}"
        Padding="0,0,5,0">
```

Ein *ToggleButton* genügt uns, Standard ist (derzeit) ein "Hamburger"-Symbol, d.h. drei Linien übereinander:

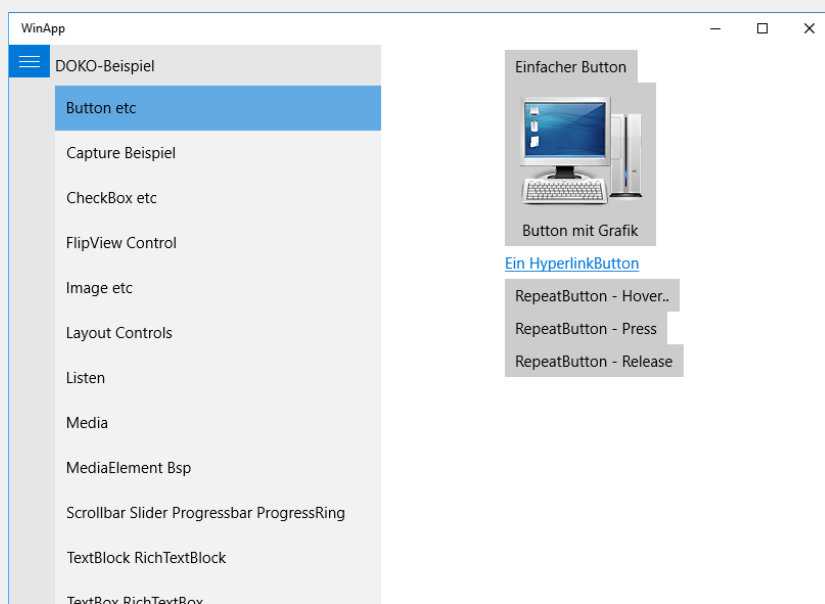
```
<ToggleButton Click="Button_Click" VerticalAlignment="Top"
              Foreground="{ThemeResource ApplicationForegroundThemeBrush}">
  <ToggleButton.Content>
    <FontIcon x:Name="Hamburger" FontFamily="Segoe MDL2 Assets"
              Glyph="⋮" Margin="0,0,0,0"/>
  </ToggleButton.Content>
</ToggleButton>
</Border>
</Grid>
```

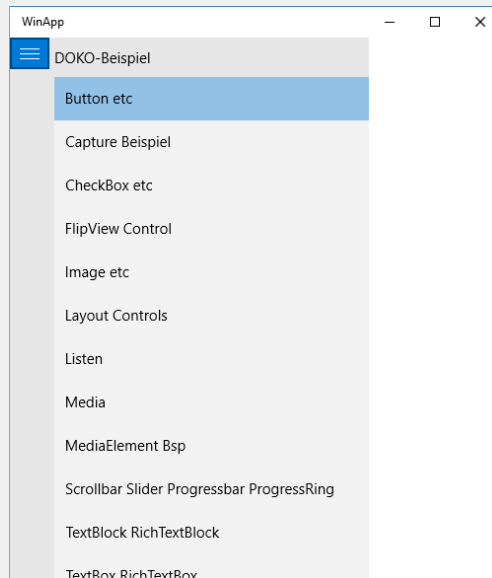
...

Alternativ können Sie die Schaltfläche natürlich auch in der Kopfzeile des Formulars unterbringen, so geht am linken Rand kein Platz verloren, was bei Smartphones etc. sicher von Interesse ist, fehlt es doch immer an Platz.

Ergebnis

Zunächst sehen Sie die Darstellung bei ausreichender Bildschirmbreite, nachfolgend die Overlay-Variante:



BEISPIEL 20.17: Verwendung *SplitView*

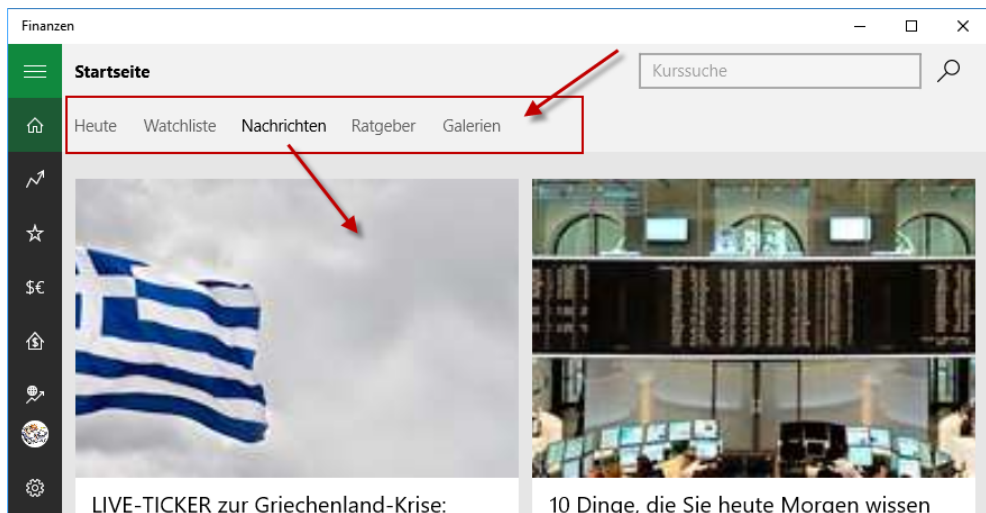
Bei der Overlay-Variante klappt das Menü nach der Auswahl wieder ein, so wird der darunterliegende Content sichtbar.

HINWEIS: Für die Navigation im Content-Bereich nutzen Sie am besten ein *Pivot*-Element.

Wer etwas mehr Aufwand investiert, kann natürlich auch in der Spalte für die Ein-/Aus-Schaltfläche Kurztasten bzw. Symbole einblenden, sodass die App auch dann bedienbar bleibt, wenn der Navigationsbereich eingeklappt ist. Hier nutzen Sie dann am besten ein *StackPanel*, ein Scrollbar würde wohl nur stören.

20.2.7 Pivot

Unter den Layout-Controls wollen wir Ihnen auch das Pivot-Control vorstellen. Dieses dient dazu, innerhalb einer Seite zwischen verschiedenen Optionen/Ansichten zu wechseln. Zum Beispiel werden bei der Finanz-App die einzelnen Rubriken *Heute*, *Watchliste*, *Nachrichten* ... über ein Pivot-Control dargestellt:



Kommen wir zu einem kleinen Beispiel:

BEISPIEL 20.18: Verwendung *Pivot*

XAML

```

...
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  Das übergeordnete Pivot-Control:
  <Pivot>
    Die einzelne Seite mit einem eigenen Header:
    <PivotItem Header="Pivot-Rubrik 1">
      <TextBlock>
        Inhalte der ersten Seite
      </TextBlock>
    </PivotItem>
    Die zweite Seite:
    <PivotItem Header="Pivot-Rubrik 2">
      <TextBlock>
        Inhalte der zweiten Seite
      </TextBlock>
    </PivotItem>
    Weitere Seiten ...
  </Pivot>
  ...

```

HINWEIS: Beachten Sie, dass innerhalb eines *PivotItems* ein Layout-Control zur Anordnung weiterer Steuerelemente benötigt wird.

20.2.8 RelativPanel

Ebenfalls neu für den Windows App-Entwickler ist das *RelativPanel*, das Sie beim Anordnen von Controls unterstützen soll. Wie der Name es schon andeutet, können Sie mit dessen Hilfe entweder ein Control bezüglich des *RelativPanels* ausrichten oder Sie nehmen Bezug auf ein anderes Control innerhalb des *RelativPanels*.

Für das Ausrichten bezüglich des *RelativPanels* stehen Ihnen folgende Eigenschaften zur Verfügung:

Eigenschaft	Standard
<i>RelativePanel.AlignBottomWithPanel</i>	<i>False</i>
<i>RelativePanel.AlignHorizontalCenterWithPanel</i>	<i>False</i>
<i>RelativePanel.AlignLeftWithPanel</i>	<i>True</i>
<i>RelativePanel.AlignRightWithPanel</i>	<i>False</i>
<i>RelativePanel.AlignTopWithPanel</i>	<i>True</i>
<i>RelativePanel.AlignVerticalCenterWithPanel</i>	<i>False</i>

HINWEIS: Die obigen Werte können Sie miteinander kombinieren, sodass sich alle neun Positionen innerhalb des *RelativPanels* zuweisen lassen.

Zusätzlich bieten sich die relativen Angaben bezogen auf ein weiteres Control an:

Eigenschaft	Bemerkung
<i>Above</i>	unter der Unterkante
<i>LeftOf</i>	an der linken Kante
<i>RightOf</i>	an der rechten Kante
<i>Below</i>	auf der Oberkante
<i>AlignBottomWith</i>	gemeinsame Unterkante
<i>AlignHorizontalCenterWith</i>	gemeinsame vertikale Mittelachse
<i>AlignLeftWith</i>	gemeinsame linken Kante
<i>AlignRightWith</i>	gemeinsame rechten Kante
<i>AlignTopWith</i>	gemeinsame Oberkante
<i>AlignVerticalCenterWith</i>	gemeinsame horizontale Mittelachse

HINWEIS: Auch hier können Sie zwei Angaben miteinander kombinieren, um eine eindeutige Ausrichtung in x- und y-Richtung zu realisieren.

BEISPIEL 20.19: Beispiele für die Verwendung des *RelativPanel***XAML**

```

...
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
  Width="640" Height="300">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="319*" />
    <ColumnDefinition Width="321*" />
  </Grid.ColumnDefinitions>

```

Relativ zum Panel ausrichten (siehe untere Abbildung linke Hälfte):

```

<RelativePanel Background="LightCyan">
  <Border Width="40" Height="40" Background="Yellow"
    RelativePanel.AlignBottomWithPanel="True"/>
  <Border Width="40" Height="40" Background="Red"
    RelativePanel.AlignLeftWithPanel="True"/>
  <Border Width="40" Height="40" Background="Blue"
    RelativePanel.AlignVerticalCenterWithPanel="True"
    RelativePanel.AlignRightWithPanel="True"/>
  <Border Width="40" Height="40" Background="Green"
    RelativePanel.AlignHorizontalCenterWithPanel="True"
    RelativePanel.AlignVerticalCenterWithPanel="True" />
</RelativePanel>

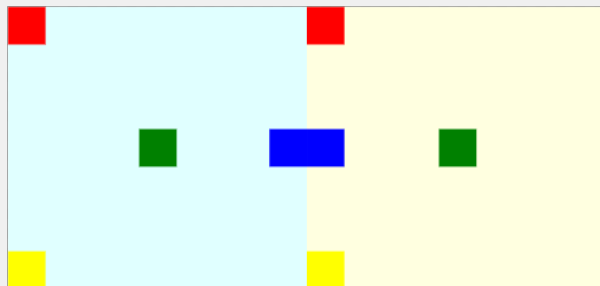
```

Relative Ausrichtung zu anderen Controls (siehe untere Abbildung rechte Hälfte):

```

<RelativePanel Grid.Column="1" Background="LightYellow">
  <Border Name="linksunten" Width="40" Height="40" Background="Yellow"
    RelativePanel.AlignBottomWithPanel="True"/>
  <Border Width="40" Height="40" Background="Red"
    RelativePanel.AlignLeftWith="linksunten"/>
  <Border Name="mitte" Width="40" Height="40" Background="Green"
    RelativePanel.AlignHorizontalCenterWithPanel="True"
    RelativePanel.AlignVerticalCenterWithPanel="True" />
  <Border Width="40" Height="40" Background="Blue"
    RelativePanel.AlignBottomWith="mitte"/>
</RelativePanel>
</Grid>
...

```

Ergebnis

20.3 Listendarstellungen

Im Zusammenhang mit der Anzeige von Datenlisten etc. werden auch entsprechende Controls benötigt, die diese Listen auf sinnvolle Weise visualisieren können.

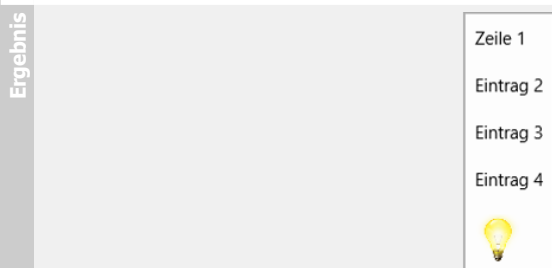
20.3.1 ComboBox, ListBox

Eigentlich sollten Sie sich im Zusammenhang mit WinRT-Anwendungen von der guten alten *ComboBox* verabschieden, ist diese doch für die Touchbedienung nur noch eingeschränkt empfehlenswert. Aber wer trennt sich schon gern von lieb gewonnenen Lösungen.

Beide Controls basieren auf einem *ItemsControl*, können also eine Liste von einzelnen *Items* aufnehmen.

BEISPIEL 20.20: Einfache *ListBox*, die Items werden per XAML definiert

XAML	<pre> ... <ListBox> <ListBoxItem>Zeile 1</ListBoxItem> <x:String>Eintrag 2</x:String> <x:String>Eintrag 3</x:String> <x:String>Eintrag 4</x:String> <Image Source="Images/bulbon.png" Width="40"></Image> </ListBox> ... </pre>
------	---



Wie Sie sehen, kann es sich um recht unterschiedliche Objekte handeln, die Sie sowohl in einer *ListBox* als auch in einer *ComboBox* unterbringen können.

Doch was ist eigentlich mit Daten in der einfachen Form einer *String*-Liste?

BEISPIEL 20.21: Datenbindung *ListBox*

C#	<pre> ... public sealed partial class Listen : Steuerelemente.Common.LayoutAwarePage { Wir definieren die Liste als Eigenschaft der aktuellen Seite: public List<string> Items { get; set; } } </pre>
----	---

BEISPIEL 20.21: Datenbindung *ListBox*

Im Konstruktor weisen wir der Liste einige Werte zu:

```
public Listen()
{
    this.InitializeComponent();
    Items = new List<string>();
    for (int i = 0; i < 100; i++)
        Items.Add("Eintrag " + i.ToString());
}
```

XAML

```
<common:Page
    x:Name="pageRoot"
...
<ListBox ItemsSource="{Binding Items, ElementName=pageRoot}" Width="200"/>
...
```

Wie Sie sehen, genügt jetzt das Binden an die aktuelle Seite (*ElementName=pageRoot*), die eigentlich zu bindende Eigenschaft *Items* können Sie wie oben gezeigt oder per *Binding Path=Items* bestimmen.

Die gleiche Vorgehensweise ist auch bei der *ComboBox* relevant, allerdings ist deren Erscheinung an die Gegebenheiten einer Touchoberfläche angepasst worden:

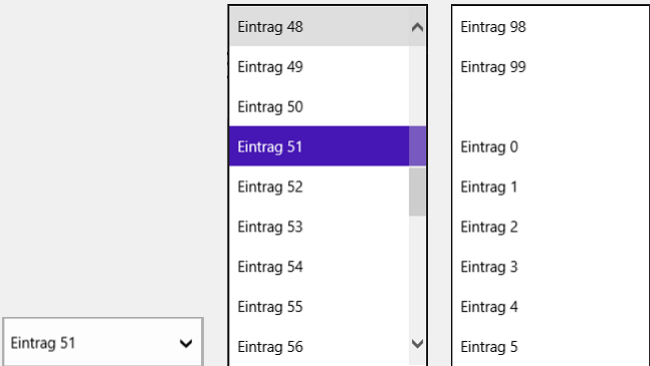
BEISPIEL 20.22: Verwendung *ComboBox*

XAML

```
...
<ComboBox ItemsSource="{Binding Path=Items, ElementName=pageRoot}" Width="200"
    Height="50" VerticalAlignment="Top"/>
...
```

Ergebnis

Die Laufzeitansicht vor und nach dem Klick:



Um eine bessere Touchbedienung zu erreichen, wird die Auswahlliste möglichst zentriert über der *ComboBox* angezeigt. Wie die dritte Abbildung zeigt, kann dabei auch das Listenende "überschritten" werden, die Liste beginnt dann wieder von vorn.

Eine etwas kompliziertere Variante der Datenbindung zeigt das folgende Beispiel. Ausgehend von der Klasse *Artikel* (*Name*, *Preis*) soll eine Liste dieser Daten in einer *ListBox* dargestellt werden.

BEISPIEL 20.23: Binden an eine *ObservableCollection*

Zunächst die Klasse *Artikel* definieren (wir implementieren *INotifyPropertyChanged*, um auf Eigenschaftsänderungen reagieren zu können):

```
...
public class Artikel : INotifyPropertyChanged
{
```

Die beiden Eigenschaften *Name* und *Preis*:

```
    private string _name;
    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            this.OnPropertyChanged("Name");
        }
    }
```

```
    private double _preis;
    public double Preis
    {
        get { return _preis; }
        set
        {
            _preis = value;
            this.OnPropertyChanged("Preis");
        }
    }
```

```
    public Artikel()
    { }
```

Die statische Methode *GetBeispielArtikel* liefert uns eine bereits gefüllte Liste von Artikeln:

```
    public static ObservableCollection<Artikel> GetBeispielArtikel()
    {
        ObservableCollection<Artikel> artikel = new ObservableCollection<Artikel>()
        {
            new Artikel() {Name = "Bier", Preis= 1.20},
            new Artikel() {Name = "Wein", Preis= 3.70},
        }
    }
```

BEISPIEL 20.23: Binden an eine *ObservableCollection*

```
...
    new Artikel() {Name = "Kohlrabi", Preis= 0.50}
};
    return artikel;
}
```

Die Implementierung des *PropertyChanged*-Ereignisses:

```
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
protected virtual void OnPropertyChanged(string propertyName)
{
    if (this.PropertyChanged != null)
    {
        this.PropertyChanged(this,
            new System.ComponentModel.PropertyChangedEventArgs(propertyName));
    }
}
...
}
```

In der eigentlichen Seite definieren wir zunächst eine Eigenschaft vom Typ *ObservableCollection<Artikel>* und weisen dieser die Beispieldaten zu:

```
...
    public ObservableCollection<Artikel> Artikeliste { get; set; }

    public Listen()
    {
        this.InitializeComponent();
        Artikeliste = Artikel.GetBeispielArtikel();
    }
...
}
```

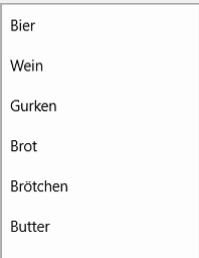
XAML

Die Datenquelle wird in diesem Fall mit der Eigenschaft *Artikelliste* zugewiesen, der anzuzeigende Member ist das Feld *Name*:

```
...
<ListBox ItemsSource="{Binding Artikeliste, ElementName=pageRoot}"
    DisplayMemberPath="Name" Width="200"/>
...

```

Ergebnis



20.3.2 ListView

Soll neben dem Namen des Artikels auch der Preis oder z.B. eine Abbildung etc. angezeigt werden, bietet es sich an, statt der *ListBox* gleich eine *ListView* zu verwenden. Prinzipiell sind beide Controls sehr ähnlich, die *ListView* ist aber die etwas komplexere Variante, die vor allem einige neue Ereignisse (*DragItemsStarting*, *ItemClick*) und Unterstützung für *SemanticZoom* und *IncrementalLoading* bietet.

BEISPIEL 20.24: Verwendung *ListView*

XAML


```
...
<ListView ItemsSource="{Binding Artikeliste, ElementName=pageRoot}" Width="150"
    Height="450" VerticalAlignment="Top" SelectionMode="Extended" Header="Artikel">

In diesem Fall wollen wir allerdings nicht nur den Namen sondern auch den Preis anzeigen
lassen. Dazu nutzen wir ein DataTemplate, das uns freie Gestaltungsmöglichkeiten bietet:

    <ListView.ItemTemplate>
        <DataTemplate>
            <Border Background="#FFFFFFBB1" Padding="5,0,5,0"
                BorderBrush="#FFF93131" BorderThickness="1" Width="100">
                <StackPanel>
                    <TextBlock Text="{Binding Name}" HorizontalAlignment="Left"/>
                    <TextBlock Text="{Binding Preis,
                        ConverterParameter='{0:C}',
                        Converter={StaticResource StringFormatConverter1}}"
                        HorizontalAlignment="Left"/>
                </StackPanel>
            </Border>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
...
```

Ergebnis

Die Laufzeitanzeige (der Scrollbar ist nur sichtbar, wenn die *ListView* den Fokus besitzt):



Der im Beispiel verwendete *StringFormatConverter* wird im Abschnitt 20.5.1 ab Seite 1015 vorgestellt.

HINWEIS: Wer bereits in WPF mit der *ListView* gearbeitet hat, wird sich vielleicht an die Möglichkeit erinnern, mehrere Spalten innerhalb der *ListView* zu generieren und diesen die jeweiligen Member der Collection zuzuordnen. Dies ist bei der WinRT-Variante nicht möglich!

20.3.3 GridView

Möchten Sie die Daten nicht nur in einer Spalte sondern auch mehrspaltig anzeigen, können Sie ein *GridView* verwenden.

BEISPIEL 20.25: Verwendung *GridView*

XAML

Wir binden die *GridView* an die aus dem Beispiel 20.19 bekannte Artikelliste.

...

```
<StackPanel>
```

Mit dem Klick auf einen Item zeigen wir in folgendem *TextBlock* den Inhalt an:

```
<TextBlock Name="txt1" Style="{StaticResource TitleTextStyle}"
    Margin="5,0,0,0"/>
```

Die *GridView*:

```
<GridView ItemsSource="{Binding Artikeliste, ElementName=pageRoot}"
    Width="400" SelectionMode="Extended" VerticalAlignment="Top"
    ItemClick="GridView_ItemClick_1" IsItemClickEnabled="True">
```

Das *DataTemplate* definiert das Aussehen der einzelnen Items:

```
<GridView.ItemTemplate>
    <DataTemplate>
        <Border Background="#FFFFFFBB1" Width="150" Height="80"
            Padding="5,0,5,0" BorderBrush="#FFF93131">
            <StackPanel>
                <TextBlock Text="{Binding Name}"
                    HorizontalAlignment="Left"/>
                <TextBlock Text="{Binding Preis, ConverterParameter='{
                    {0:C}', Converter={StaticResource
                        StringFormatConverter1}}" HorizontalAlignment="Left"/>
                <TextBox Text="{Binding Name, Mode=TwoWay}" />
            </StackPanel>
        </Border>
    </DataTemplate>
</GridView.ItemTemplate>
```


BEISPIEL 20.25: Verwendung *GridView*

Die Anzahl der maximal angezeigten Zeilen festlegen:

```
<GridView.ItemsPanel>
  <ItemsPanelTemplate>
    <WrapGrid MaximumRowsOrColumns="5"
      VerticalChildrenAlignment="Top"
      HorizontalChildrenAlignment="Left"/>
  </ItemsPanelTemplate>
</GridView.ItemsPanel>
</GridView>
</StackPanel>
```

...

C# Über das *ItemClick*-Ereignis (dieses muss mit *IsItemClickEnabled* freigeschaltet werden) können wir den aktuell gewählten Item auswerten (Typisierung nicht vergessen):

...

```
private void GridView_ItemClick_1(object sender, ItemClickEventArgs e)
{
    Artikel item = e.ClickedItem as Artikel;
    txt1.Text = item.Name;
}
```

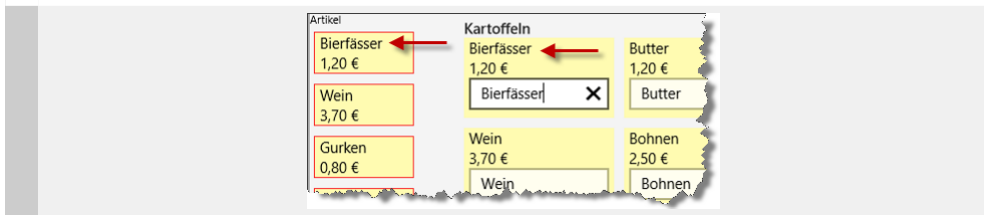
...

Ergebnis

Die Laufzeitanzeige (der Scrollbar ist nur sichtbar, wenn das *GridView* den Fokus besitzt):

Kartoffeln		
Bier 1,20 €	Butter 1,20 €	Zwiebeln 0,50 €
Bier	Butter	Zwiebeln
Wein 3,70 €	Bohnen 2,50 €	Möhren 1,14 €
Wein	Bohnen	Möhren
Gurken 0,80 €	Kartoffeln 1,50 €	Brötchen 0,50 €
Gurken	Kartoffeln	Brötchen
Brot 2,20 €	Ente 10,99 €	Kohlrabi 0,50 €
Brot	Ente	Kohlrabi
Brötchen 0,50 €	Huhn 4,50 €	
Brötchen	Huhn	

Da wir in diesem Fall auch eine *TextBox* im *DataTemplate* eingeblendet haben, können wir die Daten natürlich auch bearbeiten, was wiederum Auswirkungen auf die anderen gebundenen Controls auf der aktuellen Seite hat:

BEISPIEL 20.25: Verwendung *GridView*

HINWEIS: Weitere Anwendungsbeispiele für das *GridView* finden Sie im Kapitelbeispiel.

20.3.4 FlipView

Der eine oder andere wird sich vielleicht wundern, was das *FlipView*-Control an dieser Stelle zu suchen hat, aber es handelt sich (wie bei den vorhergehenden Controls auch) um ein von *Items-Control* abgeleitetes Control. Damit hat es die Fähigkeit, weitere Items aufzunehmen.

Die sicher plausibelste Verwendung ist die Anzeige mehrerer Grafiken, zwischen denen mit Hilfe zweier Schaltflächen gewechselt werden kann.

BEISPIEL 20.26: Verwendung *FlipView* für Bilder

XAML

```

...
<FlipView Name="flv1" MaxHeight="200" MaxWidth="400"
    SelectionChanged="FlipView_SelectionChanged">
    <Image Source="Images/bulboff.png" />
    <Image Source="Images/bulbon.png" />
    <Image Source="Images/mycomputer.png" />
</FlipView>
...

```

Ergebnis Laufzeitansicht (die Schaltflächen sind nur sichtbar, wenn die *FlipView* den Fokus besitzt):



Möchten Sie ein größeres Image (z.B. 1:1-Ansicht) an obige *FlipView* binden, funktioniert dies wie folgt:

```

<Image Source="{Binding ElementName=flv1, Path=SelectedItem.Source}" Stretch="None"
    Margin="0,50,0,50"/>

```

Doch die Anzeige und Auswahl von Bildern ist nicht die einzige Verwendungsmöglichkeit. Es ist ebenfalls problemlos möglich, die *FlipView* an eine Collection zu binden und mittels *DataTemplate* die Daten formatiert anzuzeigen:

BEISPIEL 20.27: Verwendung *ListView* mit Datenbindung

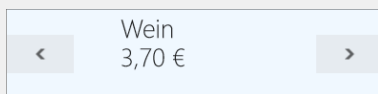
XAML

```

...
<FlipView Width="400" Height="100" ItemsSource="{Binding Artikeliste,
    ElementName=pageRoot}" Background="AliceBlue">
    <FlipView.ItemTemplate>
        <DataTemplate>
            <StackPanel Margin="120,0,120,0">
                <TextBlock Text="{Binding Name}" HorizontalAlignment="Left"
                    Style="{StaticResource SubheaderTextStyle}" />
                <TextBlock Text="{Binding Preis, ConverterParameter='{0:C}'}",
                    Converter={StaticResource StringFormatConverter}"
                    HorizontalAlignment="Left"
                    Style="{StaticResource SubheaderTextStyle}"/>
            </StackPanel>
        </DataTemplate>
    </FlipView.ItemTemplate>
</FlipView>
...

```

Ergebnis Laufzeitansicht (die Schaltflächen sind nur sichtbar, wenn die *FlipView* den Fokus besitzt):



HINWEIS: Ob obige Lösung auch intuitiv ist sei dahingestellt. Im Normalfall sieht der User ja nur die reinen Daten und nicht die Navigationsschaltflächen, ob es sich um eine Auswahl/Navigation handelt, muss man dann erst "ausprobieren".

Möchten Sie die Navigationsschaltflächen am oberen/unteren Rand einblenden, verwenden Sie einfach folgendes *ItemsPanel*:

```

<FlipView Width="400" Height="200" ItemsSource="{Binding Artikeliste,
    ElementName=pageRoot}" Background="AliceBlue">
    <FlipView.ItemTemplate>
        ...
    </FlipView.ItemTemplate>
    <FlipView.ItemsPanel>
        <ItemsPanelTemplate>
            <StackPanel Background="Transparent" Orientation="Vertical"/>
        </ItemsPanelTemplate>
    </FlipView.ItemsPanel>
</FlipView>

```

20.4 Sonstige Controls

An dieser Stelle wollen wir uns einigen mehr oder weniger neuen Controls zuwenden, die aber für das Erstellen von multimedialen Oberflächen unabdingbar sind.

20.4.1 CaptureElement

Geht es um die Anzeige der Daten einer Webcam (Tablets werden meist über eine Front- und eine Rückseiten-Kamera verfügen), bietet sich für die reine Anzeige ein *CaptureElement* an.

Weisen Sie dazu der *Source*-Eigenschaft zur Laufzeit ein initialisiertes *MediaCapture*-Objekt zu. Was sich so einfach anhört, erfordert allerdings einigen Zusatzaufwand, wie es das folgende kleine Beispiel zeigt:

BEISPIEL 20.28: Verwendung *CaptureElement*

XAML

```

...
<StackPanel Grid.Row="1" Margin="120,0,120,0">
    <CaptureElement Width="320" Height="240" Margin="0,0,0,30"
        Name="CaptureElement1"/>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <Button Click="Button_Click_1" Width="100">Start</Button>
        <Button Click="Button_Click_3" Width="100">Stop</Button>
        <Button Click="Button_Click_2" Width="100">Drehen</Button>
    </StackPanel>
</StackPanel>
...

```

C#

```

...
using Windows.Devices.Enumeration;
using Windows.Media.Capture;
...
public sealed partial class Capture_Beispiel : Steuerelemente.Common.LayoutAwarePage
{

```

Eine Instanz der *MediaCapture*-Klasse definieren:

```
MediaCapture medcap;
```

Mit dem Laden der Seite bestimmen wir zunächst die Anzahl der Video-Kameras und wählen dann das erste Gerät in der Liste aus. Alternativ können Sie über die *DeviceInformationCollection* auch Informationen über die einzelnen Geräte auswerten bevor Sie eines zuweisen:

```

private async void pageRoot_Loaded(object sender, RoutedEventArgs e)
{
    DeviceInformationCollection videocams = await
        DeviceInformation.FindAllAsync(DeviceClass.VideoCapture);

```

BEISPIEL 20.28: Verwendung *CaptureElement*

Ist mindestens ein Videogerät vorhanden, wählen wir dieses aus und initialisieren sowohl die *MediaCapture*-Instanz als auch das *CaptureElement*:

```

        if (videocams.Count > 0)
        {
            medcap = new MediaCapture();
            MediaCaptureInitializationSettings mcis = new
                MediaCaptureInitializationSettings();
            mcis.VideoDeviceId = videocams[0].Id;
            await medcap.InitializeAsync( );
            CaptureElement1.Source = medcap;
        }
    }

```

Zu diesem Zeitpunkt wird noch nichts angezeigt, dies wird erst mit dem Aufruf der *StartPreviewAsync*-Methode realisiert:

```

        private async void Button_Click_1(object sender, RoutedEventArgs e)
        {
            await medcap.StartPreviewAsync();
        }

```

Stoppen der Anzeige:

```

        private async void Button_Click_3(object sender, RoutedEventArgs e)
        {
            await medcap.StopPreviewAsync();
        }

```

Für die Art und Weise der Darstellung ist nicht das *CaptureElement* zuständig sondern die Datenquelle, d.h. das *MediaCapture*-Objekt (in diesem Fall drehen wir die Anzeige):

```

        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            medcap.SetPreviewRotation(VideoRotation.Clockwise180Degrees);
        }
    }
}

```

HINWEIS: Wenn Sie die App starten, dürfen Sie nicht vergessen, vorher die erforderlichen Funktionen in der *Package.appxmanifest*-Datei entsprechend zu setzen. Wichtig: Sie müssen Webcam **und** Mikrofon anfordern, andernfalls kommt es zu einer Fehlermeldung, auch wenn Sie nur auf das Video zugreifen wollen. Nach dem Start wird dem Anwender zunächst die obligatorische Sicherheitsabfrage (Zugriff Webcam/Mikrofon) angezeigt.

20.4.2 MediaElement

Wie es sein Name schon vermuten lässt, können Sie das *MediaElement* für die Wiedergabe von Video- oder Musik-Dateien/-Streams nutzen. Verwenden Sie die *Source*-Eigenschaft, um den gewünschten Dateinamen anzugeben, alternativ können Sie auch zur Laufzeit mit der *SetSource*-Methode einen Stream übergeben.

Die wesentlichsten Methoden zur Steuerung der Wiedergabe sind:

- *Play*
- *Stop*
- *Pause*

HINWEIS: Haben Sie die Eigenschaft *AutoPlay* auf *True* gesetzt, startet die Wiedergabe nach dem Zuweisen der *Source*-Eigenschaft (bzw. wenn diese bereits gesetzt ist) mit der Anzeige des Controls.

BEISPIEL 20.29: Laden eines Videos aus dem aktuellen App-Unterverzeichnis *videos*

```

C# ...
private async void Button_Click_4(object sender, RoutedEventArgs e)
{
    StorageFolder folder = await
        Package.Current.InstalledLocation.GetFolderAsync("videos");
    StorageFile file = await folder.GetFilesAsync("video2.mts");
    var stream = await file.OpenAsync(FileAccessMode.Read);
    MediaElement1.SetSource(stream, file.ContentType);
}
...

```

Über die Ereignisse *MediaOpened*, *MediaEnded* und *CurrentStateChanged* können Sie den aktuellen Status der Wiedergabe auswerten und gegebenenfalls eine neue Datei laden bzw. den Status der Steuerungstasten anpassen.

Für die Endloswiedergabe nutzen Sie *IsLooping*, *IsMuted* sorgt für Ruhe. Über *Volume* steuern Sie die Lautstärke.

BEISPIEL 20.30: Lautstärkeregelung mit Datenbindung eines *Sliders*

```

XAML ...
<Slider Maximum="1" Minimum="0" Value="{Binding ElementName=Media1, Path=Volume,
Mode=TwoWay}" Width="200" TickFrequency=".01" StepFrequency="0.01"
SnapsTo="Ticks" Margin="30,0,0,0"/>
...

```

HINWEIS: Die aktuelle Wiedergabeposition können Sie nur per *MediaElement.Position.TotalSeconds*-Eigenschaft und zusätzlichen *Timer* auslesen – ein Ereignis mit entsprechendem Parameter gibt es dafür **nicht**. Alternativ lässt sich jedoch zum Beispiel ein *ProgressBar* direkt an ein *MediaElement* und damit auch an dessen *Position*-Eigenschaft binden.

BEISPIEL 20.31: Fortschrittsanzeige mittels *ProgressBar* realisieren

```
XAML
...
<MediaElement Name="Media1" />
<ProgressBar Width="300" Margin="0,10,0,0"
    Maximum="{Binding ElementName=Media1, Path=NaturalDuration.TimeSpan.TotalSeconds}"
    Value="{Binding ElementName=Media1, Path=Position.TotalSeconds}"/>
...
```

Über die unterstützten Dateiformate können Sie sich unter folgender Adresse informieren:

LINK: <http://msdn.microsoft.com/en-us/library/hh986969.aspx>

20.4.3 Frame

Im Abschnitt 19.1.2 (Die Page, der Frame und das Window) ab Seite 955 haben wir Ihnen bereits das organisatorische Grundkonzept der Windows Apps aufgezeigt. Ausgangspunkt war das einzige Fenster der App, als dessen Inhalt ein zentraler Frame zugewiesen wurde. Alle Seiten Ihrer App werden innerhalb dieses Frames dargestellt, der Frame ist also nur ein Container für die eigentlichen Inhalte. Auch die Navigation zwischen den einzelnen Seiten ist Aufgabe dieses Frames (siehe dazu Abschnitt 19.2 ab Seite 961).

Was an so zentraler Stelle funktioniert, lässt sich aber auch in den einzelnen Seiten nutzen, um zum Beispiel schnell Inhalte innerhalb einer Page zu organisieren (siehe dazu das Kapitelbeispiel).

20.4.4 WebView

Für die **einfache** Anzeige von Webseiten und auch HTML-Strings bietet sich das *WebView*-Control an. Wer jetzt an das WPF *Browser*-Control denkt, liegt teilweise falsch. Auch wenn es die beiden Methoden *Navigate* (Aufruf URL) bzw. *NavigateToString* (Anzeige HTML-Quelltext inklusive Skripts) ebenfalls gibt – der wesentlichste Unterschied ist die mangelnde Unterstützung für Navigations-Ereignisse bzw. die gänzlich fehlende Unterstützung für die Seitennavigation.

BEISPIEL 20.32: Anzeige Webseite in einer *WebView*

```
C#
...
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    Uri uri = new Uri(@"http://www.doko-buch.de");
    WebView1.Navigate(uri);
}
...
```

BEISPIEL 20.33: Anzeige HTML-Code in einer WebView

```

C# ...
    private void Button_Click_2(object sender, RoutedEventArgs e)
    {
        WebView1.NavigateToString("<html><body><b>Hier könnte Ihr HTML-Text stehen  

        ...</b></body></html>");
    }
    ...

```

Einziges verwertbares Feedback des Controls ist neben dem *NavigationFailed* das *LoadCompleted*-Ereignis:

BEISPIEL 20.34: Auswerten des *LoadCompleted*-Ereignisses

```

C# ...
    private void pageRoot_Loaded_1(object sender, RoutedEventArgs e)
    {
        WebView1.LoadCompleted += WebView1_LoadCompleted;
    }

```

Wir zeigen nach dem Laden der Webseite eine kleine Vorschau in einem *Rectangle*-Control an. Dazu nutzen wir einen *WebViewBrush*, der die gerenderte Webseite enthält:

```

    void WebView1_LoadCompleted(object sender, NavigationEventArgs e)
    {
        WebViewBrush b = new WebViewBrush();
        b.SourceName = "WebView1";
        b.Redraw();
        Rectangle1.Fill = b;
    }
    ...

```

20.4.5 Tooltip

Als kleine Hilfe für den Anwender stehen auch in WinRT-Apps die bekannten Tooltips zur Verfügung. Diese können auf reinem Text oder auch auf komplexeren Kombinationen von Controls (Grafiken, Texte, evtl. auch kurzes Video) basieren.

Zugewiesen werden die *ToolTips* über den *ToolTipService* und dessen Eigenschaften:

- *ToolTip* (der gewünschte Text oder auch ein Container-Control mit beliebigen Elementen)
- *ToolTipPlacement* (wo soll der *ToolTip* erscheinen)
- *PlacementTarget* (an welchem Control soll der *ToolTip* ausgerichtet werden)

Ein kleines Beispiel zeigt die Möglichkeiten.

BEISPIEL 20.35: Verwendung von *ToolTips*

XAML

...

```
<StackPanel Grid.Row="1" Margin="120">
  <StackPanel Orientation="Horizontal">
```

Ein Rechteck mit zugehörigem *ToolTip*, der am Mauscursor ausgerichtet wird:

```
<Rectangle Height="100" Width="100" Fill="#FFB000">
  ToolTipService.ToolTip="Ein kleiner ToolTip mit Text"
  ToolTipService.Placement="Mouse" />
```

Ein *ToolTip* mit Grafik und Textausgabe:

```
<Rectangle Height="100" Width="100" Fill="#FF2EFB00">
  ToolTipService.Placement="Bottom" >
  <ToolTipService.ToolTip>
    <StackPanel>
      <Image Source="Images/bulbon.png"/>
      <TextBlock>Hier kann noch mehr Text stehen ...</TextBlock>
      <Button HorizontalAlignment="Center">Klick mich!</Button>
    </StackPanel>
  </ToolTipService.ToolTip>
</Rectangle>
```

Achtung: Obige Schaltfläche ist nur als (schlechtes) Beispiel eingefügt, Sie haben keine Möglichkeit diese zu bedienen!

Ändern der *ToolTip*-Ausrichtung:

```
<Rectangle Height="100" Width="100" Fill="#FF0072FB">
  ToolTipService.ToolTip="Ich gehöre zum blauen Rechteck!"
  ToolTipService.Placement="Right" />
```


Ein *ToolTip* mit Video (als schlechtes Beispiel, aber es geht auch):

```
<Rectangle Height="100" Width="100" Fill="#FFF0F00D">
  ToolTipService.Placement="Right">
  <ToolTipService.ToolTip>
    <StackPanel>
      <MediaElement Width="300" Height="200" Source="video.mts"
        AutoPlay="True" />
    </StackPanel>
  </ToolTipService.ToolTip>
</Rectangle>
</StackPanel>
</StackPanel>
...
```

BEISPIEL 20.35: Verwendung von ToolTips

Ergebnis

Laufzeitansicht:



Hier kann noch mehr Text stehen ...

Klick mich!

20.4.6 CalendarDatePicker

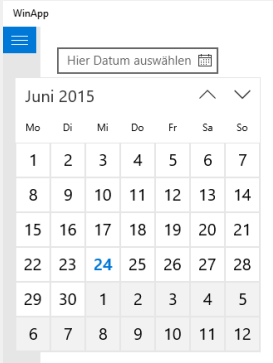
Ein häufig unentbehrliches Eingabe-Control gibt sich jetzt auch für die App-Entwicklung. Die Rede ist von einer Datumsauswahl, die vom *CalendarDatePicker* bereitgestellt wird. Die Verwendung ist trivial, ein kurzes Beispiel genügt:

BEISPIEL 20.36: CalendarDatePicker

C#

```
...
<StackPanel Margin="20">
    <CalendarDatePicker x:Name="Calendar1" Margin="0,0,0,20"
        FirstDayOfWeek="Monday"
        PlaceholderText="Hier Datum auswählen"
        IsTodayHighlighted="True"
        DateChanged="CalendarDatePicker_DateChanged" />
    <TextBlock Name="tb1" Text="Noch kein Datum gewählt!"/>
</StackPanel>
...
```

Ergebnis



20.4.7 DatePicker/TimePicker

Schon seit Windows 8.1 dabei, wollen wir Ihnen die beiden Controls für die einfache Zeit- bzw. Datumsauswahl nicht vorenthalten.

BEISPIEL 20.37: DatePicker und TimePicker

XAML

```

...
<StackPanel Margin="20">
    <TimePicker Header="Zeitauswahl:" Margin="0,20,0,20" />
    <DatePicker Header="Datumsauswahl:" Margin="0,20,0,20" />
</StackPanel>
...

```

Ergebnis

HINWEIS: Beide Controls sind für die Toucheingabe optimiert und zeigen nach der Auswahl entsprechende Auswahllisten für die gültigen Wertebereich an

20.5 Praxisbeispiele

20.5.1 Einen StringFormat-Konverter implementieren

WPF-Programmierer werden bereits eine einfache Möglichkeit vermisst haben, gebundene Felder mittels *StringFormat* an die lokalen Gegebenheiten anzupassen. Da die Eigenschaft *StringFormat* fehlt bleibt nichts anderes übrig, als sich mit einem eigenen Format-Konverter zu behelfen.

Quellcode Klasse StringFormatConverter

Erstellen Sie eine neue Klasse *StringFormatConverter* und implementieren Sie die beiden für einen Converter erforderlichen Methoden *Convert* und *ConvertBack*:

```

public class StringFormatConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        if (parameter == null) return value;
        return String.Format((String)parameter, value);
    }
}

```

```

public object ConvertBack(object value, Type targetType, object parameter, string language)
{
    return value;
}

```

HINWEIS: Wie Sie sehen, nutzen wir den Parameter des Konverters zur Zeichenformatierung.

Verwendung des Konverters

Um den Konverter in Ihrer App nutzen zu können, müssen Sie diesen zunächst in den Ressourcen deklarieren:

```

...
<Page.Resources>
    <common:StringFormatConverter x:Key="StringFormatConverter" />
</Page.Resources>
...

```

Nachfolgend lässt sich der Konverter zuweisen und parametrieren:

Verwendung bei einem Datumswert:

```

<TextBlock Text="{Binding Datum, ElementName=pageRoot, ConverterParameter='{0:dd. MMMM yyyy}',
    Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
    Style="{StaticResource SubheaderTextStyle}"/>
<TextBlock Text="{Binding Datum, ElementName=pageRoot, ConverterParameter='{0:dd.MM.yyyy}',
    Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
    Style="{StaticResource SubheaderTextStyle}"/>

```

Verwendung bei einem Gleitkommawert:

```

<TextBlock Text="{Binding Preis, ElementName=pageRoot, ConverterParameter='{0:C}',
    Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
    Style="{StaticResource SubheaderTextStyle}"/>
<TextBlock Text="{Binding Preis, ElementName=pageRoot, ConverterParameter='{0:E}',
    Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
    Style="{StaticResource SubheaderTextStyle}"/>
<TextBlock Text="{Binding Preis, ElementName=pageRoot, ConverterParameter='{0:G}',
    Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
    Style="{StaticResource SubheaderTextStyle}"/>

```

Zuweisen der Beispielwerte im Page-Konstruktor:

```

...
public sealed partial class BasicPage1 : StringFormatKonverter.Common.LayoutAwarePage
{
    public DateTime Datum { get; set; }
    public Double Preis { get; set; }
}

```

```
public BasicPage1()
{
    this.InitializeComponent();
    Datum = System.DateTime.Now.AddMonths(5);
    Preis = 123456.78;
}
...
```

Test

Die Ausgabe:

```
14. November 2012
14.11.2012
123.456,78 €
1,234568E+005
123456,78
```

Bemerkung

Mit den neuen Windows Apps hält auch eine ganze Reihe von vorgefertigten Konvertern Einzug:

```
Windows.Globalization.NumberFormatting.CurrencyFormatter
Windows.Globalization.NumberFormatting.DecimalFormatter
Windows.Globalization.NumberFormatting.IncrementNumberRouder
Windows.Globalization.NumberFormatting.NumeralSystemTransator
Windows.Globalization.NumberFormatting.PercentFormatter
Windows.Globalization.NumberFormatting.PermilleFormatter
Windows.Globalization.NumberFormatting.RoundingAlgorithm
Windows.Globalization.NumberFormatting.SignificantDigitsNumberRouder
```

Nutzen Sie diese, wenn es um Standardaufgaben geht.

20.5.2 Besonderheiten der TextBox kennen lernen

Für den von Windows Forms kommenden Umsteiger dürften sich bei Verwendung der *TextBox* einige Aha-Erlebnisse einstellen. Auf den ersten Blick nicht ersichtlich, bietet die *TextBox* in Apps einen erweiterten Funktionsumfang, der auf die Touch-Unterstützung zielt.

Virtuelle Tastatur

Entwickeln und nutzen Sie Ihre Apps auf einem normalen Desktop-PC, werden Sie bei der *TextBox* keinen Unterschied zu den aus den .NET-Anwendungen bekannten TextBoxen erkennen. Ganz anders aber ist das Verhalten, wenn Sie ein Tablet mit Toucheingabe oder den Simulator verwenden. Hier wird beim Fokuserhalt eine zusätzliche virtuelle Tastatur eingeblendet, die überhaupt erst eine Texteingabe ermöglicht:



Je nach Einsatzzweck können Sie diese Tastatur auch anpassen. Nutzen Sie dafür die *InputScopeName*-Eigenschaft, die Sie allerdings nur recht umständlich setzen können:

```
<TextBox Text="Eingabetext">
  <TextBox.InputScope>
    <InputScope>
      <InputScope.Names>
        <InputScopeName NameValue="Number"/>
      </InputScope.Names>
    </InputScope>
  </TextBox.InputScope>
</TextBox>
```

Für deutsche Apps sind folgende Werte relevant:

- *Url*
- *EmailSmtAddress*
- *Number*
- *TelephoneNumber*
- *Search*
- *AlphanumericHalfWidth*
- *AlphanumericFullWidth*

Die virtuelle Tastatur für Zahleneingaben:

Tab	!	@	#	\$	€	%	&	1	2	3	⌫
⬅	()	<	>	=	*	+	4	5	6	
➡	\	;	:	"	_	-	/	7	8	9	⬅
&123	Strg	😊	<	>	Leertaste			0	,		⌨

Die virtuelle Tastatur für E-Mail-Adresseingaben:

q	w	e	r	t	z	u	i	o	p	ü	⌫
a	s	d	f	g	h	j	k	l	ö	ä	Eingabe
↑	y	x	c	v	b	n	m	,	.	-	?
&123	Strg	😊	@						.de	<	>

HINWEIS: Das Setzen dieser Eigenschaft hat keinen Einfluss auf die zulässigen Werte in der *TextBox*, es wird lediglich die virtuelle Tastatur angepasst, um das Filtern der Tastatureingaben müssen Sie sich nach wie vor selbst kümmern.

Tasteneingaben validieren/filtern

Im Unterschied zu den guten alten *TextBox*en in Windows Forms oder WPF stehen Ihnen in Apps kaum Möglichkeiten zum Validieren und Filtern von Eingaben zur Verfügung. An dieser Stelle bleibt Ihnen nichts anderes übrig, als sich eine eigene Logik für das *TextBox*-Control zu programmieren.

Der wohl einfachste und naheliegende Ansatzpunkt ist zunächst das Beschränken der per Tastatur einzugebenden Zeichen. Nutzen Sie dazu das *KeyDown*-Ereignis:

```
private void TextBox_KeyDown_1(object sender, KeyRoutedEventArgs e)
{
    e.Handled = (e.Key < Windows.System.VirtualKey.Number0 |
                 e.Key > Windows.System.VirtualKey.Number9);
}
```

Diese Routine filtert alle Zeichen außerhalb des Bereichs 0... 9 heraus.

Ein Test wird auf den ersten Blick die Funktion bestätigen, doch probieren Sie mal einen Text per Zwischenablage in die *TextBox* einzufügen. Hier nützt uns obige Routine nichts, wir müssen zusätzliche Prüfungen im *TextChanged*-Ereignis durchführen:

Grundlage ist zunächst der Ursprungszustand der *TextBox* (dieser sollte den Regeln entsprechen). Diesen speichern wir im *GotFocus*-Ereignis ab:

```
...
    string txtlvalue = "";
...
    private void TextBox_GotFocus_1(object sender, RoutedEventArgs e)
    {
        txtlvalue = (sender as TextBox).Text;
    }
```

Ändert sich der Inhalt der *TextBox* (z.B. Einfügen der Zwischenablage), führen wir folgende Ereignisprozedur aus:

```
private void TextBox_TextChanged_1(object sender, TextChangedEventArgs e)
{
```

Eine Fehlerbehandlung für den Ernstfall:

```
    try
    {
```

Alle Zeichen außer 0...9 herausfiltern:

```
        Regex regex = new Regex("[^0-9]");
        string newvalue = regex.Replace((sender as TextBox).Text, String.Empty);
```

Sollte kein Wert mehr vorhanden sein, setzen wir den Wert auf Null:

```
        if (newvalue == String.Empty) newvalue = "0";
```

Zur Sicherheit testen wir die Umwandlung in *Int32*:

```
        Int32 val = Convert.ToInt32(newvalue);
```

Hat alles geklappt, übernehmen wir den Wert:

```
        txtlvalue = newvalue;
        (sender as TextBox).Text = newvalue;
    }
    catch (Exception)
    {
```

Im Fehlerfall stellen wir den ursprünglichen Wert wieder her¹:

```
        (sender as TextBox).Text = txtlvalue;
    }
}
```

Damit beschränkt sich die Eingabe in der *TextBox* auf gültige *Int32*-Zahlen.

¹ Eine Undo-Methode ist leider nicht vorhanden.

20.5.3 Daten in der GridView gruppieren

Ordnung ist das halbe Leben und so kommt schnell der Wunsch auf, lange Listen nicht nur optisch, sondern auch logisch zu sortieren bzw. zu gruppieren. Kein Problem, das *GridView*-Control besitzt bereits alle Voraussetzungen dafür, Sie müssen "nur noch" die Daten nach Ihren Wünschen gruppieren. Ganz nebenbei benötigen Sie noch eine *CollectionViewSource*, mit deren Hilfe wir die erforderliche Gruppierung vornehmen. Doch der Reihe nach ...

Oberfläche

Erstellen Sie ein neues Windows App-Projekt, fügen Sie der *MainPage* zunächst eine *CollectionViewSource* hinzu und legen Sie deren Eigenschaft *IsSourceGrouped* mit *True* fest:

```
...
<Page.Resources>
  <common:StringFormatConverter x:Key="StringFormatConverter" />
  <CollectionViewSource x:Name="CollectionViewSource1" IsSourceGrouped="True" />
</Page.Resources>
...
```

Die *CollectionViewSource* ist quasi der Mittler zur internen *CollectionView* mit deren Hilfe die Daten gruppiert werden. Statt die Daten direkt an das *GridView*-Control zu binden (*ItemsSource*), schieben wir jetzt die *CollectionViewSource* dazwischen.

```
...
<StackPanel Grid.Row="1" Margin="120,0,20,0" Orientation="Vertical">
  <!-- Bindung an eine CollectionViewSource
```

Die Bindung an die *CollectionViewSource*:

```
<GridView ItemsSource="{Binding Source={StaticResource CollectionViewSource1}}"
  Background="AliceBlue" ScrollViewer.VerticalScrollMode="Enabled"
  ScrollViewer.VerticalScrollBarVisibility="Visible" Height="200"
  Width="800" VerticalAlignment="Top" HorizontalAlignment="Left">
```

Für unser *GridView* haben wir zusätzlich die vertikalen Scrollbars aktiviert, so können die Gruppen später eine beliebige Höhe erreichen (die Gruppen werden vertikal angeordnet).

Das folgende *ItemTemplate* bestimmt das Aussehen des einzelnen Items (Rahmen, Inhalt, Abstand zum nächsten Item):

```
<GridView.ItemTemplate>
  <DataTemplate>
    <Border Background="#FFFFFFB1" Width="150" Height="45" Padding="5,0,5,0"
      BorderBrush="#FFF93131">
      <StackPanel>
```

Der Inhalt besteht aus zwei *TextBlock*-Controls mit den anzuzeigenden Daten¹:

```
<TextBlock Text="{Binding Name}" HorizontalAlignment="Left"/>
<TextBlock Text="{Binding Preis, ConverterParameter='{0:C}'}',
```

¹ Den *StringFormatConverter* kennen Sie bereits aus dem vorhergehenden Beispiel.

```

        Converter={StaticResource StringFormatConverter}}"
        HorizontalAlignment="Left"/>
    </StackPanel>
</Border>
</DataTemplate>
</GridView.ItemTemplate>

```

Mit dem *ItemsPanelTemplate* bestimmen Sie die spätere Anordnung und Gestaltung der einzelnen Gruppen. In diesem Fall verwenden wir ein *StackPanel*, die einzelnen Gruppen werden im vorliegenden Fall horizontal angeordnet:

```

<GridView.ItemsPanel>
    <ItemsPanelTemplate>
        <StackPanel Orientation="Horizontal"/>
    </ItemsPanelTemplate>
</GridView.ItemsPanel>

```

Damit ist zunächst die Gruppe und der Gruppeninhalt bestimmt, was bleibt ist der Gruppenkopf, den Sie mit einem *HeaderTemplate* und einem *ItemsPanelTemplate* formatieren können:

```

<GridView.GroupStyle>
    <GroupStyle>
        <GroupStyle.HeaderTemplate>
            <DataTemplate>

```

Die Daten für den Inhalt des Gruppenkopfs stammen aus der *CollectionViewSource*, wir binden an das Feld *Key*, das beim Gruppieren automatisch erstellt wird:

```

                <TextBlock Text='{Binding Key}' Foreground="Gray" FontSize="25"
                    Margin="5" />
            </DataTemplate>
        </GroupStyle.HeaderTemplate>
    </GroupStyle.Panel>
    <ItemsPanelTemplate>
        <StackPanel Margin="4" Background="Brown" />
    </ItemsPanelTemplate>
</GroupStyle.Panel>
</GroupStyle>
</GridView.GroupStyle>
</GridView>

```

HINWEIS: Unter Bemerkungen finden Sie weitere alternative Gestaltungsmöglichkeiten

Quelltext

Kopieren Sie zunächst die Klasse *Artikel* aus dem Beispiel 20.19 in das Projekt und passen Sie den Namespace an. Nachfolgend müssen wir uns nur noch um das Gruppieren der Daten und die Zuweisung der *Source* für die *CollectionViewSource* kümmern:

```

...
public sealed partial class BasicPagel : GridViewGruppieren.Common.LayoutAwarePage

```

```
{
    public BasicPage1()
    {
        this.InitializeComponent();
    }
}
```

Aus unserer einfachen Collection von *Artikel*-Objekten erstellen wir mit folgender LINQ-Abfrage eine gruppierte Collection, die wir jetzt der *CollectionViewSource* (nicht dem *GridView*) zuweisen:

```
var result = from art in Artikel.GetBeispielArtikel()
              group art by art.Name.Substring(0,1)
              into grp orderby grp.Key select grp;
```

HINWEIS: Als Gruppierungsattribut verwenden wir den ersten Buchstaben des Artikelnamens. Sie könnten auch problemlos nach einem kompletten Feld oder auch nach unterschiedlichen Preisen gruppieren lassen.

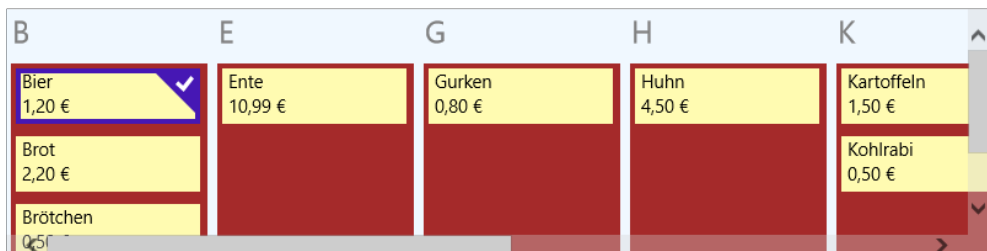
```
CollectionViewSource1.Source = result;
```

...

Damit ist das Beispiel fertig und wir können einen ersten Test wagen.

Test

Erwartungsgemäß erscheinen die Artikel jetzt in Gruppen, die nach dem Anfangsbuchstaben des Artikels gebildet wurden:



HINWEIS: Wir haben das *GridView* bewusst klein gehalten, so erscheinen auch die Scrollbars eher (nur wenn das Control den Fokus besitzt).

Bemerkung

Im vorliegenden Beispiel werden beliebig viele Gruppenmitglieder senkrecht unter dem Gruppenkopf angezeigt. Wer auf einen vertikalen Scrollbar verzichten will, kann die Gruppen auch mit Hilfe eines *VariableSizedWrapGrid* anordnen und die Anzahl der Zeilen beschränken. In diesem Fall werden die Gruppenmitglieder in horizontaler Richtung "umgebrochen", die folgende Gruppe verschiebt sich weiter nach links.

```

<GridView ItemsSource="{Binding Source={StaticResource CollectionViewSource1}}"
    Background="AliceBlue" Height="160" Width="800" VerticalAlignment="Top"
    HorizontalAlignment="Left" Margin="0,50,0,0">
...
    <GridView.GroupStyle>
        <GroupStyle>
            <GroupStyle.HeaderTemplate>
                <DataTemplate>

```

Wir ändern auch gleich noch das Aussehen des Gruppenkopfes mittels *Border*:

```

        <Border Width="50" Height="33" Background="Brown"
            CornerRadius="10,10,0,0" Margin="4,0,0,0" >
            <TextBlock Text='{Binding Key}' Foreground="White"
                FontSize="25" HorizontalAlignment="Center"
                VerticalAlignment="Center" />
        </Border>
    </DataTemplate>
</GroupStyle.HeaderTemplate>
<GroupStyle.Panel>
    <ItemsPanelTemplate>

```

Hier bestimmen wir, wie viele Zeilen pro Gruppe angezeigt werden:

```

        <VariableSizedWrapGrid MaximumRowsOrColumns="2"
            Orientation="Vertical" Margin="4,0,4,0" Background="Brown" />
    </ItemsPanelTemplate>
</GroupStyle.Panel>
</GroupStyle>
</GridView.GroupStyle>
</GridView>

```

Das Ergebnis unserer Bemühungen:



Wenn eine horizontale Anordnung der Gruppen nicht gefällt kann auch dies ändern, wie es das folgende Beispiel zeigt:

```

<GridView ItemsSource="{Binding Source={StaticResource CollectionViewSource1}}"
    Background="AliceBlue" ScrollViewer.VerticalScrollMode="Enabled"
    ScrollViewer.VerticalScrollBarVisibility="Visible" Height="300" Width="500"
    VerticalAlignment="Top" HorizontalAlignment="Left" Margin="0,50,0,0">
...

```

Die Gruppen vertikal anordnen:

```
<GridView.ItemsPanel>
  <ItemsPanelTemplate>
    <StackPanel Orientation="Vertical"/>
  </ItemsPanelTemplate>
</GridView.ItemsPanel>
<GridView.GroupStyle>
  <GroupStyle>
```

...

```
<GroupStyle.Panel>
  <ItemsPanelTemplate>
```

Die Gruppeninhalte horizontal anordnen:

```
<StackPanel Margin="4" Background="Brown"
  Orientation="Horizontal" />
</ItemsPanelTemplate>
</GroupStyle.Panel>
</GroupStyle>
</GridView.GroupStyle>
</GridView>
```

Die umformatierte *GridView* zeigt die nächste Abbildung.

B			
Bier 1,20 €	Brot 2,20 €	Brötchen 0,50 €	
E			
Ente 10,99 €			
G			

Weitere Informationen zur *CollectionViewSource* finden Sie ab Seite 1030 im Praxisbeispiel

► 20.5.5 Die *CollectionViewSource* verwenden.

20.5.4 Das *SemanticZoom-Control* verwenden

Umfangreiche Listendarstellungen im *GridView* haben den Nachteil, dass der Nutzer möglicherweise die Übersicht verliert. Eine erste Lösungsmöglichkeit hat das vorhergehende Praxisbeispiel gezeigt: die Gruppierung innerhalb der *GridView*. Wer jedoch schon einmal mit Touch-Eingabegeräten gearbeitet hat wird sich vielleicht erinnern, dass es auch entsprechende Gesten für das Verkleinern/Vergrößern (*ZoomIn/ZoomOut*) gibt. In der Übersichtsansicht kann der Nutzer dann zum Beispiel zu einer bestimmten Gruppe springen.

Die gleiche Funktionalität bietet auch das so genannte *SemanticZoom*-Control. Sie definieren zwei Zustände, um das Umschalten zwischen diesen per Gestensteuerung und um die Anzeige der Übergangsanimation kümmert sich das Control.

Die grundsätzliche XAML-Struktur:

```
<SemanticZoom>
  <ZoomedInView>
```

Hier folgt die Definition der detaillierten Ansicht:

```
    <GridView></GridView>
  </ZoomedInView>
  <ZoomedOutView>
```

Hier definieren Sie das Aussehen der Übersicht:

```
    <GridView></GridView>
  </ZoomedOutView>
</SemanticZoom>
```

Der Clou: Klicken Sie in der Übersicht auf einen Eintrag, springt die Detailansicht auf eben diese Gruppe, d.h., es bietet sich eine schnelle Navigationsmöglichkeit innerhalb der **gruppierten** Collection. Wie Sie die Collection gruppiert anzeigen, haben wir bereits im vorhergehenden Praxisbeispiel 20.5.3 gezeigt.

HINWEIS: Wir verwenden als Datenquelle die Artikelliste aus dem Beispiel 20.19.

Oberflächen

Erstellen Sie ein neues App-Projekt, fügen Sie der *MainPage* zunächst eine *CollectionViewSource* hinzu und legen Sie deren Eigenschaft *IsSourceGrouped* mit *True* fest:

```
...
  <Page.Resources>
    <common:StringFormatConverter x:Key="StringFormatConverter" />
    <CollectionViewSource x:Name="CollectionViewSource1" IsSourceGrouped="True" />
  </Page.Resources>
...
  <StackPanel Grid.Row="1" Margin="120,0,120,0" Orientation="Vertical">
```

Im *SemanticZoom*-Control realisieren wir zunächst den *ZoomedOutView*-Zustand:

```
    <SemanticZoom x:Name="semanticZoom" VerticalAlignment="Bottom" >
      <SemanticZoom.ZoomedOutView>
```

Die *GridView* wird an die bereits definierte *CollectionViewSource* gebunden, achten Sie darauf, dass bei diesem Grid der *Path* auf *CollectionGroups* festgelegt ist (das ist in unserem Fall die Collection mit den Anfangsbuchstaben):

```
      <GridView Name="GridView1" ItemsSource="{Binding Path=CollectionGroups,
        Source={StaticResource CollectionViewSource1}}" IsSwipeEnabled="True" >
        <GridView.ItemTemplate>
```

Das Aussehen bestimmen Sie wie gewohnt mit einem *DataTemplate*:

```
<DataTemplate>
    <TextBlock Text="{Binding Group.Key}"
               FontFamily="Segoe UI Light" FontSize="22"
               Foreground="Black" />
</DataTemplate>
</GridView.ItemTemplate>
```

Die Größe der einzelnen Items und deren Ausrichtung kann mit dem *ItemsPanelTemplate* festgelegt werden:

```
<GridView.ItemsPanel>
    <ItemsPanelTemplate>
        <WrapGrid ItemWidth="60" ItemHeight="60"
                  MaximumRowsOrColumns="1" VerticalChildrenAlignment="Center"/>
    </ItemsPanelTemplate>
</GridView.ItemsPanel>
```

Über den *ItemContainerStyle* können Sie bequem das Aussehen (nicht den Inhalt) der einzelnen Items steuern:

```
<GridView.ItemContainerStyle>
    <Style TargetType="GridViewItem">
        <Setter Property="Margin" Value="4" />
        <Setter Property="Padding" Value="10" />
        <Setter Property="BorderBrush" Value="Gray" />
        <Setter Property="BorderThickness" Value="1" />
        <Setter Property="HorizontalContentAlignment" Value="Center" />
        <Setter Property="VerticalContentAlignment" Value="Center" />
    </Style>
</GridView.ItemContainerStyle>
</GridView>
</SemanticZoom.ZoomedOutView>
```

Damit ist die Definition der *ZoomOutView* abgeschlossen, wir können uns der *ZoomInView* zuwenden:

```
<SemanticZoom.ZoomedInView>
```

Die *GridView* wird so wie im vorhergehenden Praxisbeispiel konfiguriert, wir gehen an dieser Stelle nicht erneut darauf ein:

```
<GridView ItemsSource="{Binding Source={StaticResource
CollectionViewSource1}}" Background="AliceBlue"
ScrollViewer.VerticalScrollMode="Enabled"
ScrollViewer.VerticalScrollBarVisibility="Visible" Height="200"
VerticalAlignment="Top" HorizontalAlignment="Left">
    <GridView.ItemTemplate>
        <DataTemplate>
            <Border Background="#FFFFFBB1" Width="250" Height="45"
                    Padding="5,0,5,0" BorderBrush="#FFF93131">
                <StackPanel>
```

```

        <TextBlock Text="{Binding Name}"
            HorizontalAlignment="Left"/>
        <TextBlock Text="{Binding Preis,
            ConverterParameter='{0:C}',
            Converter={StaticResource StringFormatConverter}}"
            HorizontalAlignment="Left"/>
    </StackPanel>
</Border>
</DataTemplate>
</GridView.ItemTemplate>
<GridView.ItemsPanel>
    <ItemsPanelTemplate>
        <StackPanel Orientation="Horizontal"/>
    </ItemsPanelTemplate>
</GridView.ItemsPanel>
<GridView.GroupStyle>
    <GroupStyle>
        <GroupStyle.HeaderTemplate>
            <DataTemplate>
                <TextBlock Text='{Binding Key}' Foreground="Gray"
                    FontSize="25" Margin="5" />
            </DataTemplate>
        </GroupStyle.HeaderTemplate>
        <GroupStyle.Panel>
            <ItemsPanelTemplate>
                <VariableSizedWrapGrid Orientation="Vertical"
                    Height="400" />
            </ItemsPanelTemplate>
        </GroupStyle.Panel>
    </GroupStyle>
</GridView.GroupStyle>
</GridView>
</SemanticZoom.ZoomedInView>
</SemanticZoom>
<ToggleSwitch IsOn="{Binding ElementName=semanticZoom, Path=IsZoomedInViewActive,
    Mode=TwoWay}">Detailansicht</ToggleSwitch>

```

Quellcode

Kopieren Sie zunächst die Klasse *Artikel* aus dem Beispiel 20.19 in das Projekt und passen Sie den Namespace an. Nachfolgend müssen wir uns nur noch um das Gruppieren der Daten und die Zuweisung der *Source* für die *CollectionViewSource* kümmern:

```

...
public BasicPage1()
{
    this.InitializeComponent();
    var result = from art in Artikel.GetBeispielArtikel() group art
        by art.Name.Substring(0, 1) into grp orderby grp.Key select grp;

```



```
CollectionViewSource1.Source = result;
}
```

Test

Nach dem Start wird Ihnen zunächst die Detailansicht (*ZoomInView*) angezeigt. Besitzen Sie ein touchfähiges Gerät nutzen Sie die Geste zum Verkleinern (zwei Finger zusammenführen), um in die Übersichtsansicht (*ZoomOutView*) zu wechseln. Alternativ können Sie auch mit dem Mausrad zusammen mit der Strg-Taste diesen Effekt erreichen.

Die *ZoomInView*:

B	E	G
Bier 1,20 €	Ente 10,99 €	Gurken 0,80 €
Brot 2,20 €		
Brötchen 0,50 €		

Die *ZoomOutView*:

B	E	G	H	K	M	R	W	Z
---	---	---	---	---	---	---	---	---

Klicken Sie in der *ZoomOutView* auf den Buchstaben "Z", so sollte nach Rückkehr in die *ZoomInView* die Gruppe mit dem Anfangsbuchstaben "Z" in den sichtbaren Bereich gerückt sein.

HINWEIS: Alternativ kann die Ansicht auch über die Eigenschaft *IsZoomedInViewActive* gewechselt werden, wie wir es zum Beispiel mit dem *ToggleSwitch* realisiert haben.

Haben Sie beim *ItemsPanelTemplate* für die *ZoomOutView* den Wert von *MaximumRowsOrColumns* zum Beispiel auf 3 festgelegt, dürfte die *ZoomOutView* wie folgt aussehen:

B	H	R
E	K	W
G	M	Z

20.5.5 Die CollectionViewSource verwenden

Wer sich durch die Praxisbeispiele dieses Kapitels gekämpft hat, wird bereits mehrfach in Kontakt mit der *CollectionViewSource* getreten sein. Dieses Control fungiert quasi als Mittler zur automatisch erstellten View (Satzzeigerverwaltung) bei einer Datenbindung. Da die aus WPF bekannte statische Methode *CollectionViewSource.GetDefaultView* für den Zugriff auf die View nicht zur Verfügung steht, stellt das Control den einfachsten Weg dar, um mit dem *CollectionView*-Objekt zu arbeiten (Eigenschaft *View*).

HINWEIS: Ein Sortieren und/oder Filtern ist bei dieser Version der *CollectionViewSource* nicht möglich. Diese Funktionalität müssen Sie mittels LINQ realisieren. Damit beschränkt sich die Funktionalität in diesem Bereich auf das Gruppieren von Daten, wie es auch im Praxisbeispiel 20.5.3 ab Seite 1021 beschrieben ist.

Was bleibt, ist die Verwaltung des Satzzeigers, die wir im folgenden kleinen Beispiel demonstrieren wollen.

Oberfläche

Erstellen Sie ein neues Windows App-Projekt und fügen Sie der *MainPage* zunächst eine *CollectionViewSource* hinzu:

```
...
<Page.Resources>
    <CollectionViewSource x:Name="CollectionViewSource1" />
</Page.Resources>
...
```

Unsere kleine App-Oberfläche enthält eine *ListBox*, die wir an die *CollectionViewSource* binden:

```
<StackPanel Grid.Row="1" Margin="120,0,20,0" Orientation="Horizontal">
    <ListBox ItemsSource="{Binding Source={StaticResource CollectionViewSource1}}"
        DisplayMemberPath="Name" Width="200" Height="400" VerticalAlignment="Top"/>
    <StackPanel Orientation="Vertical" Margin="40,0,0,0">
        <TextBlock>Navigation:</TextBlock>
    </StackPanel>
</StackPanel>
```

Die folgenden Schaltflächen dienen als Navigationsschaltflächen (siehe auch Sourcecode):

```
<StackPanel Orientation="Horizontal">
    <Button Content="⬅️" Click="Button_Click_2"
        FontFamily="Segoe UI Symbol"/>
    <Button Content="." Click="Button_Click_3" FontFamily="Segoe UI Symbol"/>
    <Button Content="." Click="Button_Click_4" FontFamily="Segoe UI Symbol"/>
    <Button Content="➡️" Click="Button_Click_5"
        FontFamily="Segoe UI Symbol"/>
    <Button Content="5" Click="Button_Click_6" FontFamily="Segoe UI Symbol"/>
</StackPanel>
```

HINWEIS: Beachten Sie die Verwendung der Schriftart "Segoe UI Symbol", die mit Windows 8 eingeführt wurde. Diese erspart Ihnen in vielen Fällen die Verwendung von Grafiken.

Die aktuelle Position des Satzzeigers können wir über die Eigenschaft *CurrentPosition* der *View* abrufen:

```
<TextBlock Margin="0,30,0,0">Position:</TextBlock>
<TextBlock Text="{Binding Path=View.CurrentPosition,
    ElementName=CollectionViewSource1}"
    Style="{StaticResource SubheaderTextStyle}"/>
<TextBlock Margin="0,30,0,0">Auswertung Event:</TextBlock>
```

Diesen Text setzen wir per *CurrentChanged*-Ereignis:

```
<TextBlock Name="txt1" Text="" Style="{StaticResource SubheaderTextStyle}"/>
</StackPanel>
</StackPanel>
...
```

Quelltext

Kopieren Sie zunächst die Klasse *Artikel* aus dem Beispiel 20.19 in das Projekt und passen Sie den Namespace an. Nachfolgend können wir uns um den Konstruktor der Seite kümmern:

```
...
public BasicPage1()
{
    this.InitializeComponent();
}
```

Daten abrufen und der *CollectionViewSource* zuweisen:

```
CollectionViewSource1.Source = Artikel.GetBeispielArtikel();
```

Wir nutzen die beiden Ereignisse *CurrentChanged* (nach dem Satzzeigerwechsel) und *CurrentChanging* (vor dem Satzzeigerwechsel):

```
CollectionViewSource1.View.CurrentChanged += View_CurrentChanged;
CollectionViewSource1.View.CurrentChanging += View_CurrentChanging;
}
```

Wenn der aktuelle Artikel-Datensatz "Ente" ist, soll sich die Satzzeigerposition nicht mehr verändern lassen (die Änderung könnten Sie von weiteren Bedingungen abhängig machen):

```
void View_CurrentChanging(object sender, CurrentChangingEventArgs e)
{
    if ((CollectionViewSource1.View.CurrentItem as Artikel).Name == "Ente")
    {
        if (e.IsCancelable)
            e.Cancel = true;
    }
}
```

Hat sich die Satzzeigerposition erfolgreich geändert, zeigen wir den Namen des aktuell gewählten Artikels an:

```
void View_CurrentChanged(object sender, object e)
{
    txt1.Text = (CollectionViewSource1.View.CurrentItem as Artikel).Name;
}
```

Wechsel zum ersten Datensatz:

```
private void Button_Click_2(object sender, RoutedEventArgs e)
{
    CollectionViewSource1.View.MoveCurrentToFirst();
}
```

Wechsel zum vorhergehenden Datensatz:

```
private void Button_Click_3(object sender, RoutedEventArgs e)
{
    CollectionViewSource1.View.MoveCurrentToPrevious();
    if (CollectionViewSource1.View.IsCurrentBeforeFirst)
        CollectionViewSource1.View.MoveCurrentToFirst();
}
```

Wechsel zum nächsten Datensatz:

```
private void Button_Click_4(object sender, RoutedEventArgs e)
{
    CollectionViewSource1.View.MoveCurrentToNext();
    if (CollectionViewSource1.View.IsCurrentAfterLast)
        CollectionViewSource1.View.MoveCurrentToLast();
}
```

Wechsel zum letzten Datensatz:

```
private void Button_Click_5(object sender, RoutedEventArgs e)
{
    CollectionViewSource1.View.MoveCurrentToLast();
}
```

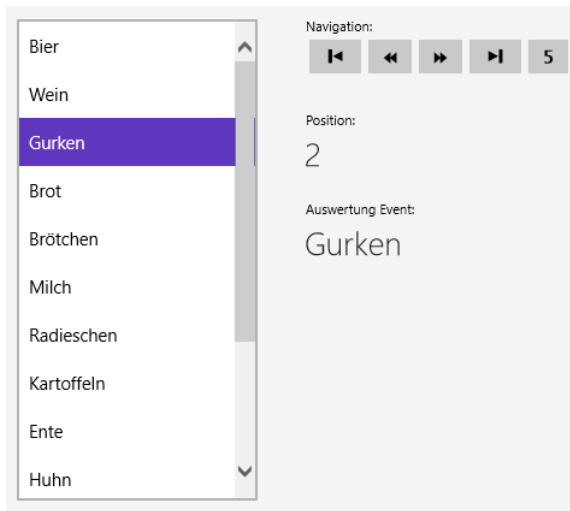
Sprung zum fünften Datensatz:

```
private void Button_Click_6(object sender, RoutedEventArgs e)
{
    CollectionViewSource1.View.MoveCurrentToPosition(5);
}
```

...

Test

Nach dem Start der App können Sie mittels *ListBox* oder mit den Navigationstasten zwischen den Datensätzen hin- und herspringen. Haben Sie einmal den Artikel "Ente" ausgewählt, ist eine Änderung der Satzzeigerposition nicht mehr möglich.



20.5.6 Zusammenspiel ListBox/AppBar

Es wird immer wieder propagiert: Windows App-Oberflächen sollen standardmäßig nur die notwendigsten Informationen anzeigen, weitere Optionen, Befehle sollen auf den AppBars am oberen oder unteren Bildschirmrand eingeblendet werden, wenn dies erforderlich ist. Leicht gesagt, doch wie sieht dies in der Praxis aus?

Unser Beispiel zeigt Ihnen wie Sie ein AppBar einblenden, wenn Einträge in einer *ListBox* (dies könnte auch jedes andere Listen-Control sein) ausgewählt werden bzw. wie Sie die *AppBar* ausblenden, wenn kein Eintrag selektiert ist.

Oberfläche

Erstellen Sie ein neues Windows App-Projekt und fügen Sie der *MainPage* eine *CollectionViewSource* hinzu:

```
...
<Page.Resources>
  <CollectionViewSource x:Name="CollectionViewSource1" />
</Page.Resources>
```

Wir erstellen zunächst den *AppBar* für den unteren Bildschirmrand:

```
<Page.BottomAppBar>
  <AppBar x:Name="BottomAppBar1" Padding="10,0,10,0">
```

Ein Grid hilft uns bei der Ausrichtung (die Schaltflächen sollen wegen der Erreichbarkeit mit den Fingern möglichst am linken und rechten Rand liegen):

```
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="50%"/>
        <ColumnDefinition Width="50%"/>
```

```
</Grid.ColumnDefinitions>
```

Diese Schaltflächen werden am linken Rand ausgerichtet:

```
<StackPanel Orientation="Horizontal" Grid.Column="0" HorizontalAlignment="Left">
  <AppBarButton x:Name="Edit" Label="Edit" Click="Edit_Click_1">
    <AppBarButton.Icon>
      <FontIcon FontFamily="Segoe UI Symbol" Glyph="✎"/>
    </AppBarButton.Icon>
  </AppBarButton>
  <AppBarButton x:Name="Delete" Label="Delete" Click="Delete_Click_1">
    <AppBarButton.Icon>
      <FontIcon FontFamily="Segoe UI Symbol" Glyph="✖"/>
    </AppBarButton.Icon>
  </AppBarButton>
</StackPanel>
```

Diese Schaltfläche wird rechts unten angezeigt:

```
<StackPanel Orientation="Horizontal" Grid.Column="1"
  HorizontalAlignment="Right">
  <AppBarButton x:Name="Help" Label="Help" Click="Help_Click_1">
    <AppBarButton.Icon>
      <FontIcon FontFamily="Candara" Glyph="?" />
    </AppBarButton.Icon>
  </AppBarButton>
</StackPanel>
</Grid>
</AppBar>
</Page.BottomAppBar>
```

...

Als Beispiel für eine Datenanzeige dient uns eine *ListBox*, welche wir an die *CollectionViewSource* binden:

```
<StackPanel Grid.Row="1" Margin="120,0,20,0" Orientation="Horizontal">
  <ListBox Name="ListBox1" ItemsSource="{Binding Source={StaticResource
    CollectionViewSource1}}" DisplayMemberPath="Name" Width="200" Height="400"
    VerticalAlignment="Top" SelectionMode="Multiple"
    SelectionChanged="ListView_SelectionChanged" />
</StackPanel>
```

...

HINWEIS: Mit *SelectionChanged* reagieren wir zur Laufzeit auf Änderungen der Auswahl in der *ListBox*.

Quelltext

Kopieren Sie zunächst die Klasse *Artikel* aus dem Beispiel 20.19 in das Projekt und passen Sie den Namespace an. Nachfolgend kümmern wir uns um den Konstruktor der Seite:

```
...
public BasicPage1()
{
    this.InitializeComponent();

```

Daten laden:

```
        CollectionViewSource1.Source = Artikel.GetBeispielArtikel();
    }

```

Ändert sich die Auswahl in der *ListBox* (mehr als ein ausgewählter Item), blenden wir die *AppBar* ein (*IsOpen*) und sorgen dafür, dass diese auch geöffnet bleibt (*IsSticky*):

```
private void ListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (this.ListBox1.SelectedItems.Count > 0)
    {
        this.BottomAppBar1.IsSticky = true;
        this.BottomAppBar1.IsOpen = true;
    }

```

Keine Auswahl, keine Anzeige:

```
    else
    {
        this.BottomAppBar1.IsOpen = false;
        this.BottomAppBar1.IsSticky = false;
    }
}

```

Klickt der Nutzer auf die *Delete*-Schaltfläche, zeigen wir eine Sicherheitsabfrage an:

```
async private void Delete_Click_1(object sender, RoutedEventArgs e)
{
    string msg = "Datensatz wirklich löschen?";
    if (ListBox1.SelectedItems.Count > 1)
        msg = "Datensätze wirklich löschen?";

```

MessageDialog erzeugen:

```
var messageDialog = new MessageDialog(msg, "Sicherheitsabfrage");

```

Zwei Schaltflächen einblenden:

```
messageDialog.Commands.Add(new UICommand("Ja",
    new UICommandInvokedHandler(this.CommandInvokedHandler), true));
messageDialog.Commands.Add(new UICommand("Nein",
    new UICommandInvokedHandler(this.CommandInvokedHandler), false));
messageDialog.DefaultCommandIndex = 1;
messageDialog.CancelCommandIndex = 1;

```

Anzeige des Dialogs:

```
await messageDialog.ShowAsync();
}

```

Das Ergebnis des Dialogs können wir in folgendem Ereignis auswerten:

```
private void CommandInvokedHandler(IUICommand command)
{
```

Wurde die *Ja*-Schaltfläche gedrückt, löschen wir alle markierten Einträge:

```
    if ((bool)command.Id)
    {
        while (ListBox1.SelectedItems.Count > 0)
        {
            (CollectionViewSource1.Source as
                ObservableCollection<Artikel>).Remove(
                ListBox1.SelectedItems[0] as Artikel);
        }
    }
}
```

Die Reaktion auf die *Edit*-Schaltfläche ist nur eine Statusmitteilung:

```
async private void Edit_Click_1(object sender, RoutedEventArgs e)
{
    var messageDialog = new MessageDialog("Funktion nicht implementiert!", "Hinweis");
    await messageDialog.ShowAsync();
}
...

```

Test

Öffnen Sie die App, markieren Sie einige Einträge in der *ListBox* und versuchen Sie diese per *AppBar* zu löschen:

