HOCHSCHULE HAMM-LIPPSTADT

Praktische Informatik

Vorlesung 10

Listen, Tabellen und Bäume

Zuletzt haben wir gelernt…

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

Inhalt heute

- ItemsControl
- DataTemplate
- ItemsPanel
- ListBox
- ComboBox
- ListView
- TreeView

Listen von Daten

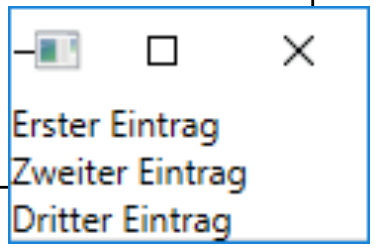
- Bislang haben wir lediglich Steuerelemente kennen gelernt, die einzelne Werte anzeigen können.
  - Sog. `ContentControls`, wie Labels, TextBoxen, Buttons, …
- Die WPF verfügt aber auch über eine Vielzahl von Steuerelementen, um Mengen von Daten darzustellen.
  - Solche Daten werden in grafischen Benutzeroberflächen oft in Form von **Listen**, **Tabellen** oder **Bäumen** dargestellt.
- Wie wir sehen werden, binden wir die Daten an solche Steuerelemente meist mit Hilfe von **Data Binding** an.
  - Über Templates kann die Darstellung der Daten konfiguriert werden.
- Wir werden uns schrittweise die Möglichkeiten der verschiedenen Steuerelemente ansehen.

ItemsControl

- Das einfachste Steuerelement, um eine Menge von Daten darzustellen, ist das `ItemsControl`.
  - Es zeigt standardmäßig seine Elemente in einer vertikalen Liste an.

- Ein Objekt der Klasse `ItemsControl` speichert seine Elemente in der Abhängigkeitseigenschaft `Items`.
  - Dort können im XAML neue Elemente hinzugefügt werden, z.B. einfache Strings:

```
<ItemsControl x:Name="itemsControl">
    <ItemsControl.Items>
        <system:String>Erster Eintrag</system:String>
        <system:String>Zweiter Eintrag</system:String>
        <system:String>Dritter Eintrag</system:String>
    </ItemsControl.Items>
</ItemsControl>
```



Data Binding eines ItemsControls

- Die Elemente können natürlich auch im Code Behind in das `ItemsControl` eingefügt werden:

```
public MainWindow()
{
    InitializeComponent();

    itemsControl.Items.Add("Vierter Eintrag");
}
```

- Meist werden die Daten eines `ItemControl` aber mit Hilfe von **Data Binding** angebunden.

- Dazu sind prinzipiell die folgenden Schritte notwendig:
  - Erstellen einer Eigenschaftsmethode im Code Behind, um die Daten bereitzustellen.
  - Den DataContext des Fensters entsprechend setzen.
  - Erstellen eines Data Bindings im XAML.

Datenbindung eines ItemControls

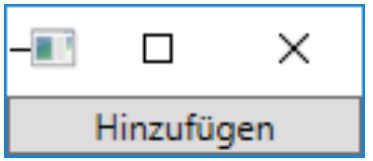
- Zunächst erstellen wir also im Code Behind die Datenquelle und setzen den DataContext:

```
public partial class MainWindow : Window
{
   public List<string> Elements { get; set; }

   public MainWindow()
   {
      InitializeComponent();

      Elements = new List<string>()
      { "Erster Eintrag", "Zweiter Eintrag", "Dritter Eintrag" };

      DataContext = this;
   }
}
```

- Anschließend konfigurieren wir das Data Binding im XAML:

```
<ItemsControl ItemsSource="{Binding Elements}" />
```

Daten hinzufügen



- Wir erweitern unser Beispiel um einen Button, um neue Einträge hinzuzufügen.

```
<StackPanel>
    <Button Click="Button_Click">Hinzufügen</Button>
    <ItemsControl ItemsSource="{Binding Elements}" />
</StackPanel>
```

- Im Code Behind wird ein Event Handler benötigt.
  - Dort fügen wir in die angebundene List-Collection einen neuen Eintrag ein.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Elements.Add("Neuer Eintrag");
}
```

- Wir stellen allerdings fest, dass der neue Eintrag nicht in der GUI angezeigt wird.
  - Das ItemsControl bekommt nicht mit, dass die angebundenen Daten geändert wurden.

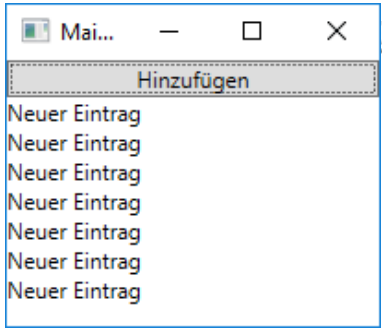
INotifyPropertyChanged

- Ein ähnliches Problem haben wir bereits beim Data Binding besprochen.
  - Damals haben wir das Problem gelöst, indem wir die Schnittstelle `INotifyPropertyChanged` implementiert haben.
  - Somit kann dem Ziel der Datenbindung mitgeteilt werden, dass Änderungen eingetreten sind.

- Auch dieses Mal könnten wir das Problem so lösen.
  - Aber es gibt noch einen anderen Weg.

- Das .Net Framework stellt eine besondere Datenstruktur bereit, die bereits über Änderungen informieren kann.
  - Die Klasse `ObservableCollection`.

ObservableCollection



- Die Klasse `ObservableCollection` unterscheidet sich kaum von der Klasse List.
  - Allerdings kann sie Beobachter über Änderungen an ihren Daten informieren.

- Entsprechend ersetzen wir in unserem Beispiel die Klasse List durch `ObservableCollection`.

```
public ObservableCollection<string> Elements { get; set; }
```

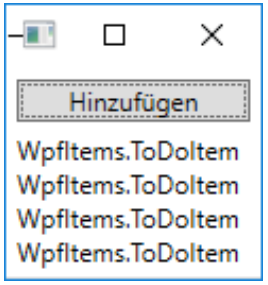
- Anschließend funktioniert der Hinzufügen-Knopf wie gewünscht.
  - Das `ItemsControl` erkennt die Änderungen an der Liste und zeigt den neuen Eintrag an.

ToDo-List

- In unserem Beispiel sind die Elemente der Liste lediglich einfache Strings.
  - Meistens handelt es sich aber um Objekte mit mehreren Eigenschaften.

- Nehmen wir an, wir wollten eine Anwendung zur Verwaltung von ToDo-Einträgen erstellen.
  - Entsprechend besteht die Liste nicht aus Strings, sondern aus Objekten der Klasse ToDoItem.

```
public class ToDoItem
{
    public string Title { get; set; }
    public int Completion { get; set; }
}
```



DataTemplate

- Wir ersetzen an allen Stellen die Klasse String durch `ToDoItem`.
  - Wir stellen nun fest, dass die Elemente in der Oberfläche nicht mehr korrekt dargestellt werden.

- Das `ItemsControl` kann nicht wissen, wie ein `ToDoItem` dargestellt werden soll.
  - Wir müssen dies erst festlegen.
  - Dies geschieht über ein sog. **`DataTemplate`**.

- Ein `DataTemplate` funktioniert ähnlich, wie ein `ControlTemplate` aus der letzten Vorlesung.
  - Mit einem `DataTemplate` legen wir das Aussehen jedes einzelnen Elements in einer Liste fest.

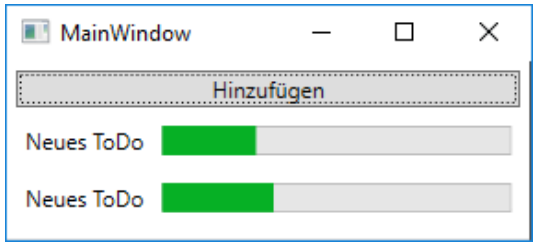
DataTemplate festlegen

- Wir erstellen ein `DataTemplate`, um `ToDoItem-Objekte` vernünftig darzustellen.
  - Dabei nutzen wir auch Data Binding.
  - Dadurch werden die Daten eines einzelnen Objektes an der richtigen Stelle eingeblendet.

- Ein `DataTemplate` kann z.B. als Ressource definiert werden:

```
<DataTemplate x:Key="toDoItemTemplate">
   <Grid Margin="0,0,0,5" >
      <Grid.ColumnDefinitions>
         <ColumnDefinition Width="*" />
         <ColumnDefinition Width="100" />
      </Grid.ColumnDefinitions>
      <TextBlock Text="{Binding Title}" />
      <ProgressBar Grid.Column="1"
          Minimum="0" Maximum="100" Value="{Binding Completion}" />
   </Grid>
</DataTemplate>
```

DataTemplate benutzen



- Im `ItemsControl` können wir das zuvor definierte `DataTemplate` nun benutzen.
  - Dazu setzen wir die Eigenschaft ItemTemplate auf die zuvor definierte Ressource.

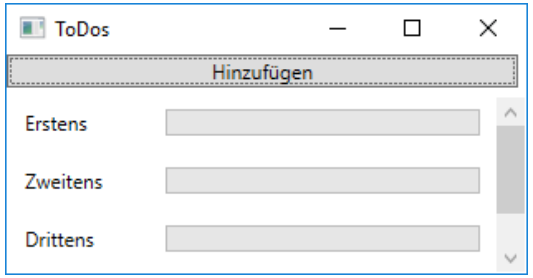
```
<ItemsControl Margin="0,5,0,0" ItemsSource="{Binding Elements}"
   ItemTemplate="{StaticResource toDoItemTemplate}" />
```

- Anschließend werden die Elemente in der Oberfläche gemäß der `DataTemplate` angezeigt.
  - Alle Klassen, die von `ItemsControls` ableiten besitzen diese Eigenschaft.
  - Also z.B. auch die Elemente `ListBox`, `ComboBox`, `Ribbon`, `TabControl`, `ListView`, `DataGrid`, …

ListBox

- Unsere Anwendung funktioniert schon ganz gut.
  - Allerdings fallen uns noch einige Dinge auf.
- Wenn wir das Fenster verkleinern, können wir die unteren Elemente nicht sehen.
  - Es erscheint auch keine `Scrollbar`, so dass wir entsprechend scrollen könnten.
- Wir können auch kein Element auswählen.
  - Dadurch können wir die Anwendung auch nicht weiter entwickeln, um Elemente zu löschen usw.
- Um diese Features zu bekommen, müssen wir ein anderes Interaktionselement benutzen.
  - Die Klasse `ListBox` stellt uns all dies zur Verfügung.

ListBox benutzen



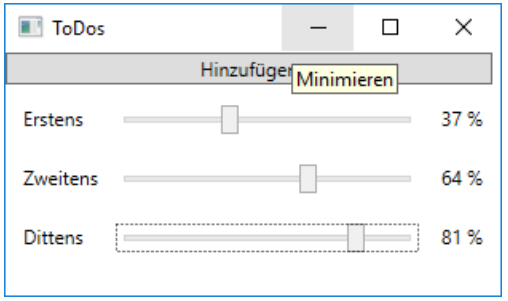
- Im XAML ist es sehr einfach, die Klasse `ItemsControl` durch `ListBox` zu ersetzen.

```
<ListBox BorderThickness="0"
    x:Name="listbox"
    Margin="0,5,0,0"
    HorizontalContentAlignment="Stretch"
    ItemsSource="{Binding Elements}"
    ItemTemplate="{StaticResource toDoItemTemplate}" />
```

Damit die Elemente den verfügbaren horizontalen Platz benutzen, sollte HorizontalContentAlignment auf Stretch gestellt werden.

- Die ListBox sollte in einem `DockPanel` oder `Grid` untergebracht werden.
  - Dadurch wird automatisch eine Scrollleiste eingeblendet, wenn das Fenster zu klein ist, um alle Elemente anzuzeigen.

Slider im Template



- Wir wollen zudem dafür sorgen, dass der Fertigstellungsgrad eines ToDo-Eintrags in der Oberfläche geändert werden kann.
  - Dies ist sehr einfach möglich, wenn wir im `DataTemplate` die `ProgressBar` durch einen `Slider` ersetzen.
  - Zudem zeigen wir noch den numerischen Wert in einem `TextBlock` an.

```
<DataTemplate x:Key="toDoItemTemplate">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="200" />
      <ColumnDefinition Width="40" />
    </Grid.ColumnDefinitions>
    <TextBlock Text="{Binding Title}" />
    <Slider Grid.Column="1" Minimum="0" Maximum="100" Value="{Binding Completion}" />
    <TextBlock Grid.Column="2"
      Text="{Binding Completion, StringFormat='{}{0} %'}" HorizontalAlignment="Right" />
    </Grid>
</DataTemplate>
```

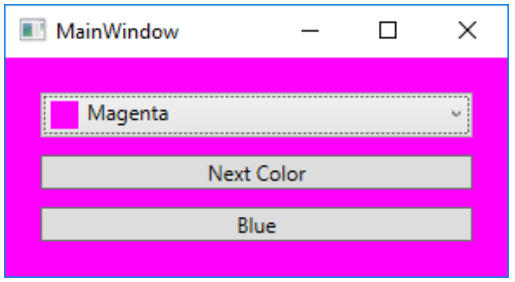
Auswahl

- In der `ListBox` können Einträge ausgewählt werden.
  - Welche Auswahlmöglichkeiten erlaubt sind, ist in der Eigenschaft `SelectionMode` definierbar.
  - Im Standardfall kann immer nur ein Element ausgewählt werden (`SelectionMode=Single`).
- Über das Ereignis `SelectionChanged` kann auf eine geänderte Auswahl reagiert werden.
  - Die `ListBox` stellt zudem viele weitere Methoden und Eigenschaften bzgl. der Auswahl bereit.
- Beispiele:
  - `SelectedItem` → Liefert das aktuell selektierte Objekt, oder setzt dieses.
  - `SelectedIndex` → Liefert den Index des aktuell selektierten Objekts, oder setzt diesen.
  - SelectAll() → Selektiert alle Elemente, wenn dies erlaubt ist.

ComboBox

- Die ComboBox ist in vielen Bereichen ähnlich zur ListBox.
  - Man kann über ein ItemTemplate die Darstellung eines Elements in der Liste beeinflussen.
  - Es kann genau ein Element ausgewählt werden.
- Allerdings wird wesentlich weniger Platz in der Oberfläche benötigt.
  - Die Liste der Elemente ist nicht immer komplett sichtbar, sondern wird aufgeklappt.

Beispiel



- Wir wollen alle vordefinierten Farben in einer `ComboBox` anzeigen.
  - Der Hintergrund des Fenster soll durch die aktuell ausgewählte Farbe bestimmt werden.
  - Auch soll durch einige Buttons die aktuelle Auswahl beeinflusst werden können.

- Im XAML-Code definieren wir zunächst ein `DataTemplate`.
  - Dadurch werden die Einträge in der `ComboBox` ordentlich formatiert.

```
<DataTemplate x:Key="data_template">
    <StackPanel Orientation="Horizontal">
        <Rectangle Width="16" Height="16" Margin="0,2,5,2" Fill="{Binding}" />
        <TextBlock Text="{Binding}" />
    </StackPanel>
</DataTemplate>
```

XAML

- Der Rest des XAML-Code ist wieder sehr einfach.
  - Wir nutzen Data Binding, um die Combo Box mit den Namen der Farben zu füllen.

- Auch werden mehrere Buttons eingeführt, und mit Event Handlern im Code Behind verknüpft.
  - Ein Button soll dafür sorgen, zur nächsten Farbe zu springen.
  - Ein Button soll die ausgewählt Farbe auf Blau einstellen.

```
<StackPanel Margin="20">
    <ComboBox x:Name="combo" MinWidth="250"
        ItemTemplate="{StaticResource data_template}" />
    <Button Margin="0,10" Click="Button_Next_Click">Next Color</Button>
    <Button Click="Button_Blue_Click">Blue</Button>
</StackPanel>
```

Code Behind

- Im Code Behind weisen wir der `ItemSource` Eigenschaft die Namen der Farben zu.
  - Zudem reagieren wir auf die Click-Ereignisse der Buttons:

```
public MainWindow()
{
   InitializeComponent();
   combo.ItemsSource = from p in typeof(Colors).GetProperties() select p.Name;
   combo.SelectedItem = "White";
}

private void Button_Next_Click(object sender, RoutedEventArgs e)
{
   combo.SelectedIndex++;
}

private void Button_Blue_Click(object sender, RoutedEventArgs e)
{
   combo.SelectedValue = "Blue";
}
```

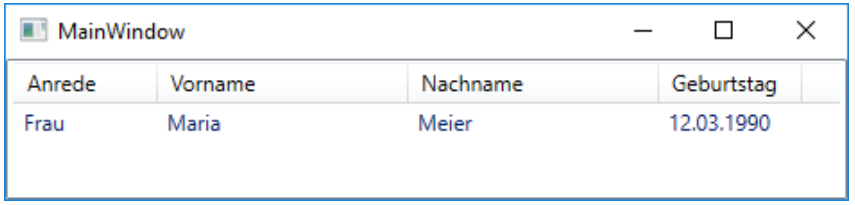
ListView

- Die Listbox kann bereits sehr gut Mengen darstellen.
  - Oft wollen wir jedoch die Daten in Form einer **Tabelle** darstellen.
  - Eine Tabelle besteht aus Spalten und Zeilen und hat eine Überschrift für jede Spalte.

- Um solche Darstellungen zu erzeugen, kann die Klasse **`ListView`** genutzt werden.
  - Die ListView leitet von ListBox ab, besitzt also alle ihre Eigenschaften.
  - ListView stellt die Daten wie die Dateien im Windows Explorer dar.

- Als Besonderheit stellt ListView die Eigenschaft `View` zur Verfügung.
  - Ihr kann ein Objekt vom Typ GridView zugeordnet werden.
  - Ein GridView konfiguriert die Darstellung der Daten in den Spalten.

Beispiel



- Wir erstellen eine Anwendung zur Verwaltung von Mitgliedern in einem Sportverein.
  - Dazu erstellen wir eine eigene Klasse Mitglied.

```
public class Mitglied
{
    public Anrede Anrede { get; set; }
    public string Vorname { get; set; }
    public string Nachname { get; set; }
    public DateTime Geburtstag { get; set; }
}
```

- Im Code Behind erstellen wir eine beobachtbare Liste von Mitglieder und setzen den DataContext darauf:

```
public MainWindow()
{
   InitializeComponent();

   var members = new ObservableCollection<Mitglied>();
   members.Add(new Mitglied() { … });
   DataContext = members;
}
```

XAML

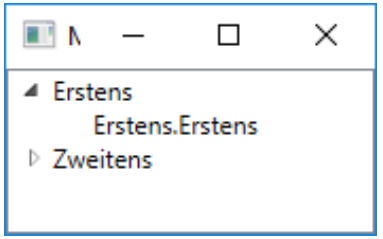
- Im XAML definieren wir ein `GridView` als Ressource.

```
<GridView x:Key="view">
   <GridViewColumn Header="Anrede" Width="80"
    DisplayMemberBinding="{Binding Anrede}" />
   <GridViewColumn Header="Vorname" Width="140"
    DisplayMemberBinding="{Binding Vorname}" />
   <GridViewColumn Header="Nachname" Width="140"
    DisplayMemberBinding="{Binding Nachname}" />
   <GridViewColumn Header="Geburtstag" Width="80"
    DisplayMemberBinding="{Binding Geburtstag, StringFormat=dd.MM.yyyy}" />
</GridView>
```

- Die Ressource nutzen wir in der `ListView` für die Eigenschaft View:

```
<DockPanel>
    <ListView ItemsSource="{Binding}" View="{StaticResource view}" />
</DockPanel>
```

TreeView



- Ein `TreeView` ist dazu geeignet, hierarchisch organisierte Daten anzuzeigen.
  - Die Klasse `TreeView` leitet von `ItemsControl` ab.
  - Entsprechend sind alle Features des `ItemsControl` auch in der `TreeView` vorhanden.

```
<TreeView>
    <TreeViewItem IsExpanded="True" Header="Erstens">
        <TreeViewItem Header="Erstens.Erstens"/>
    </TreeViewItem>
    <TreeViewItem Header="Zweitens">
        <TreeViewItem Header="Zweitens.Erstens"/>
    </TreeViewItem>
</TreeView>
```

- Im einfachsten Fall zeigt das TreeView Elemente vom Typ `TreeViewItem` an.
  - Über Data Binding können aber auch andere Objekte angezeigt werden.

Beispiel

- Ein bekanntes Beispiel hierarchischer Daten sind die Laufwerke, Verzeichnisse und Dateien auf einem Rechner.
  - Wir wollen daher schrittweise eine Anwendung mit einem `TreeView` konstruieren, um Verzeichnisse und Dateien des lokalen Rechners anzuzeigen.

- Wir wollen von Anfang an mit Data Binding arbeiten.
  - Deshalb erstellen wir zunächst einige Datenklassen, deren Objekte später an das `TreeView` angebunden werden.

- Beim Data Binding kapselt das TreeView die Datenobjekte in einem Objekt der Klasse TreeViewItem.

Datenmodell

- Unser Datenmodell besteht aus mehreren, unterschiedlichen Klassen.
  - Drive, Directory und File.
  - Ein Objekt einer dieser Klassen repräsentiert ein Element auf dem Rechner.

- Alle diese Klassen teilen sich gemeinsame Eigenschaften, z.B. den Namen.
  - Daher ist es sinnvoll, eine gemeinsame Basisklasse einzuführen: Element

- In dieser Basisklasse können wir auch das Nachladen der nächsten Verzeichnisebene realisieren.
  - Das muss ja sowohl bei einem Laufwerk, als auch bei Verzeichnissen geschehen.
  - Bei Dateien allerdings nicht.

Klasse Element

```
public class Element
{
    protected ObservableCollection<Element> elements = null;
    public string Name { get; set; }

    public Element(string name)
    {
        Name = name;
    }

    public ObservableCollection<Element> SubElements
    {
        get
        {
            if (elements == null)
                LoadSubElements();

            return elements;
        }
    }

    protected virtual void LoadSubElements()
    {
        elements = new ObservableCollection<Element>();

        try
        {
            foreach (var d in System.IO.Directory.EnumerateDirectories(Name))
                elements.Add(new Folder(d));

            foreach (var d in System.IO.Directory.EnumerateFiles(Name))
                elements.Add(new File(d));
        }
        catch
        {
        }
    }
```

Die Eigenschaft SubElements stellt alle Kind-Element bereit.

Wenn die Liste noch leer ist, wird sie nachgeladen. Das passiert aber erst, wenn der Nutzer eine Ebene aufklappt.

Drive, Directory, File

```
public class Drive : Element
{
    public Drive(string name) : base(name)
    {
    }
}
```

```
public class Directory : Element
{
    public Directory(string name) : base(name)
    {
    }
}
```

```
public class File : Element
{
    public File(string name) : base(name)
    {
    }
}
```

Die drei Klassen Drive, Directory und File erben von Element.

Die unterschiedlichen Klassen machen Sinn, da wir über unterschiedliche Templates in der Oberfläche die Darstellung der Objekte steuern können.

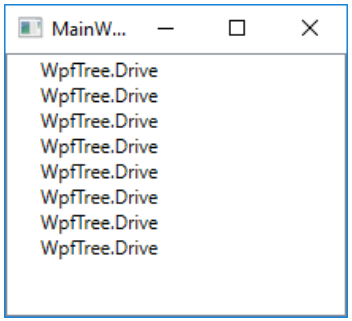
TreeView benutzen

- Im XAML setzen wir Datenquelle auf die Eigenschaft Elements, die sich im Code Behind befinden muss:

```
<TreeView ItemsSource="{Binding Elements}">
</TreeView>
```

- Im Code Behind erzeugen wir für jedes Laufwerk ein Objekt der Klasse Drive:

```
public ObservableCollection<Drive> Elements { get; private set; }

public MainWindow()
{
   InitializeComponent();

   Elements = new ObservableCollection<Drive>();

   foreach (var s in Environment.GetLogicalDrives())
      Elements.Add(new Drive(s);

   DataContext = this;
}
```



Data Template

- Leider zeigt die Anwendung so nur eine Liste von Objektnamen an.
  - Wir müssen dem `TreeView` wieder erklären, wie es Objekte vom Typ Drive vernünftig darstellen kann.
  - Im `ItemsControls` haben wir dazu ein `DataTemplate` genutzt.

- Auch im `TreeView` wird wieder ein `DataTemplate` eingesetzt.
  - Da ein einzelnes Element des `TreeView` aber wieder Sub-Elemente besitzen kann, benötigen wir ein besonderes Template.

- Entsprechend benutzen wir im `TreeView` die Klasse `HierarchicalDateTemplate`.

HierarchicalDataTemplate

- Die Klasse **`HierarchicalDataTemplate`** wird dazu benutzt, um im TreeView die Darstellung eines einzelnen Elements zu definieren.
  - Sie besitzt die Eigenschaft `ItemSource`, um Sub-Elemente anbinden zu können.

- Wir erzeugen ein solches `HierarchicalDataTemplate` und legen es im Bereich Resources des **`TreeView`** ab:
  - Wir definieren zudem über die Eigenschaft DataType die Art der Objekte, für die dieses Template benutzt werden soll.
  - Für jede Datenklasse können wir so separat eine eigenes Template anlegen.

```
<TreeView.Resources>
    <HierarchicalDataTemplate
       DataType="{x:Type local:Drive}" ItemsSource="{Binding SubElements}">
        <StackPanel Orientation="Horizontal">
            <Image Source="003-folder.png" Width="20" />
            <TextBlock Text="{Binding Name}" />
        </StackPanel>
    </HierarchicalDataTemplate>
</TreeView.Resources>
```

Wir haben heute gelernt…

- Wie man mit dem ItemsControl eine Menge von Elementen anzeigt.
- Wie man über ein DataTemplate festlegt, wie ein einzelnen Element dargestellt wird.
- Wie man mit dem ItemsPanel definiert, in welchem LayoutContainer die Elemente abgelegt werden.
- Was die ListBox im Vergleich zum ItemsControl kann.
- Wie man mit der ComboBox umgehen kann.
- Wie man tabellarische Daten mit der ListView anzeigt.
- Wie man hierarchische Daten mit einem TreeView anzeigt.