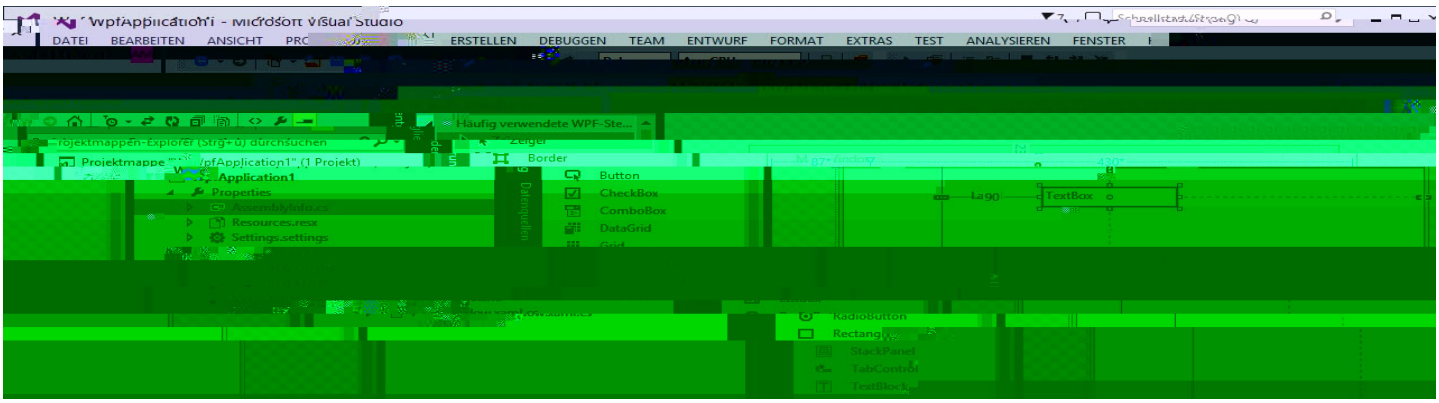
AUSGABE 03/2021

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

André Minhorst Verlag

Das WPF Webbrowser-Steuerelement

Es gibt verschiedene Gründe, warum Sie ein Webbrowser-Steuerelement in Ihrer Anwendung benötigen könnten. Vielleicht möchten Sie einfach die Möglichkeit bieten, bestimmte Webseiten anzuzeigen. Oder Sie wollen es zur Anzeige von HTML-Inhalten verwenden, die Sie aus Dateien lesen oder per Code zusammenstellen. Schließlich bietet HTML einige Möglichkeiten, die anders oder besser zu realisieren sind als unter WPF. Dann können Sie das Webbrowser-Steuerelement verwenden, um die Benutzeroberfläche der WPF-Anwendung zu erweitern. Dieser Artikel erläutert die Grundlagen für die Programmierung des WPF Webbrowser-Steuerelements.

WPF Webbrowser-Steuerelement hinzufügen

Das WPF Webbrowser-Steuerelement fügen Sie wie die übrigen Steuerelemente auf verschiedene Arten zu einem WPF-Fenster hinzu. Sie können es beispielsweise aus dem Bereich **Alle WPF-Steuerelemente** aus der **Toolbox** in das WPF-Fenster ziehen oder dieses anklicken und dann im WPF-Fenster einen Rahmen aufziehen, der direkt die Größe des Steuerelements angibt.

Wer lieber mit Code arbeitet, fügt schlicht das Element **Webbrowser** in das **Grid**-Element des Fensters ein. Wir legen hier mit **wb** direkt den Namen des Steuerelements fest, damit wir es später leichter referenzieren können:

```
<Grid>
    <WebBrowser x:Name="wb"></WebBrowser>
</Grid>
```

Damit ist allerdings noch nichts erreicht, denn das WPF Webbrowser-Steuerelement lädt weder von selbst eine Seite noch bietet es Benutzeroberflächen-Elemente wie ein Textfeld zur Eingabe einer URL oder Schaltflächen zum Navigieren. Also legen wir in den nächsten Abschnitten selbst Hand an und fügen die benötigten Elemente hinzu.

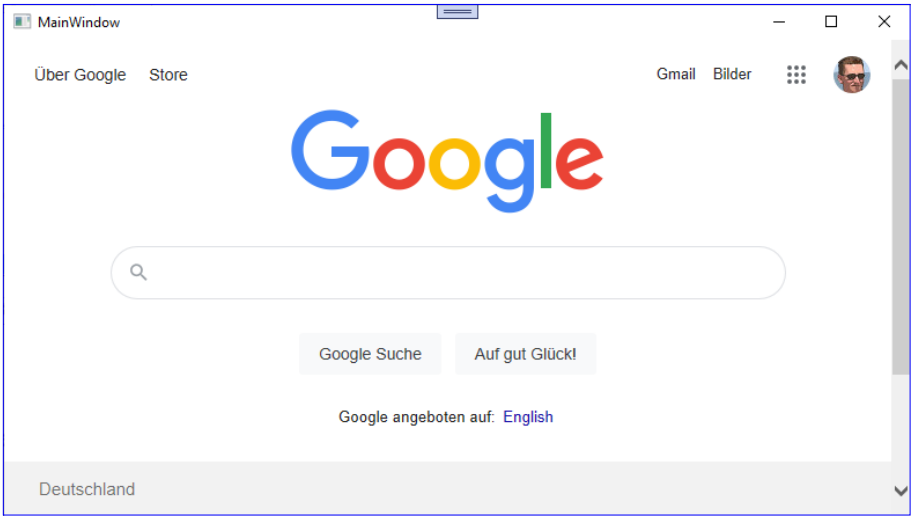


Bild 1: Anzeige einer Webseite im WPF Webbrowser-Steuerelement

Anzeigen einer Webseite beim Starten der Anwendung

Damit das WPF Webbrowser-Steuerelement beim Starten der Anwendung wie in Bild 1 direkt

eine Webseite anzeigt, fügen wir eine Ereignismethode hinzu, die durch das Ereignis **Loaded** des **Window**-Elements ausgelöst wird. Das benötigte Attribut sieht wie folgt aus:

```
<Window x:Class="MainWindow" ...  Loaded="Window_Loaded">
```

Die Ereignismethode **Window_Loaded** implementieren wir wie folgt und fügen als einzige Anweisung einen Aufruf der **Navigate**-Methode des WPF Webbrowser-Steuerelements hinzu, das wir mit **wb** benannt haben. Die **Navigate**-Methode erwartet die URL als Parameter:

```
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
    wb.Navigate("http://www.google.de")
End Sub
```

Navigieren per URL-Textfeld

Normalerweise werden Sie nicht nur eine Seite anzeigen wollen, sondern dem Benutzer die Gelegenheit geben wollen, selbst eine URL einzugeben, die das WPF Webbrowser-Steuerelement anzeigen soll. Dazu fügen wir über dem Webbrowser-Steuerelement ein Textfeld plus Bezeichnungsfeld hinzu sowie eine Schaltfläche zum Anzeigen der gewünschten URL.

Im XAML-Code legen wir dafür ein **DockPanel**-Element an, das oben ein **Grid**-Element mit **Label**, **TextBox** und **Button** enthält sowie unten das **Webbrowser**-Steuerelement:

```
<DockPanel>
    <Grid DockPanel.Dock="Top"><
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Label Grid.Column="0">URL:</Label>
        <TextBox x:Name="txtURL" Grid.Column="1"></TextBox>
        <Button x:Name="btnNavigate" Grid.Column="2" Click="btnNavigate_Click">Navigate</Button>
    </Grid>
    <WebBrowser x:Name="wb"></WebBrowser>
</DockPanel>
```

Navigieren per Klick auslösen

Um den Navigationsvorgang auszulösen, reicht eigentlich die folgende Methode:

```
Private Sub btnNavigate_Click(sender As Object, e As RoutedEventArgs)
    Dim strURL As String
```

```
    strURL = txtURL.Text
    wb.Navigate(strURL)
End Sub
```

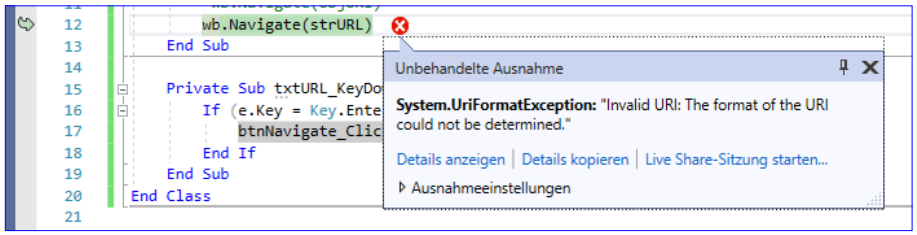


Bild 2: Fehler bei Angabe einer URL ohne **http://**

Diese setzt allerdings voraus, dass der Benutzer eine vollständige URL liefert, also beispielsweise inklusive **http://** oder **https://**. Dann können wir beim Anklicken der Schaltfläche den Text des Textfeldes **txtURL** in eine **String**-Variable einlesen und deren Inhalt der Methode **Navigate** als Parameter übergeben. Gibt der Benutzer allerdings beispielsweise nur **www.google.de** an, erhalten wir den Fehler aus Bild 2.

Wenn wir diesen Fehler ausschließen wollen, können wir eine leicht abgewandelte Version der zuvor vorgestellten Methode nutzen. Hier führen wir das Objekt **objUri** mit dem Typ **Uri** ein. Wir schreiben den Inhalt von **txtURL** wieder in **strURL**. Dann erstellen wir ein neues Objekt des Typs **UriBuilder** und übergeben diesem den Inhalt von **strURL**. Die Eigenschaft **Uri** dieses Objekts liefert das erstellte **Uri**-Objekt und wir speichern es in **objURI**. Dieses übergeben wir schließlich als Parameter an die **Navigate**-Methode:

```
Private Sub btnNavigate_Click(sender As Object, e As RoutedEventArgs)
    Dim strURL As String
    Dim objURI As Uri
    strURL = txtURL.Text
    objURI = New UriBuilder(strURL).Uri
    wb.Navigate(objURI)
End Sub
```

Auf diese Weise kann der Benutzer auch Internetadressen ohne Angabe von **http://** oder **https://** in das Textfeld eingeben.

Seite direkt beim Betätigen der Eingabetaste öffnen

Benutzerfreundlicher wird die Lösung, wenn wir dem Benutzer die Betätigung der Eingabetaste zum Aufrufen der angegebenen Seite anbieten. Dazu fügen wir dem **TextBox**-Element **txtURL** noch das **KeyDown**-Attribut hinzu, für das wir die folgende Ereignismethode hinterlegen:

```
Private Sub txtURL_KeyDown(sender As Object, e As KeyEventArgs)
    If (e.Key = Key.Enter) Then
        btnNavigate_Click(Me, New RoutedEventArgs)
    End If
End Sub
```

Die Methode prüft, ob das mit den **KeyEventArgs** übergebene Zeichen dem Wert **Key.Enter** entspricht, als der Eingabetaste. In diesem Fall ruft die Methode die zuvor definierte Methode **btnNavigate_Click** auf.

Javascript-Fehler im WPF Webbrowser-Steuerelement

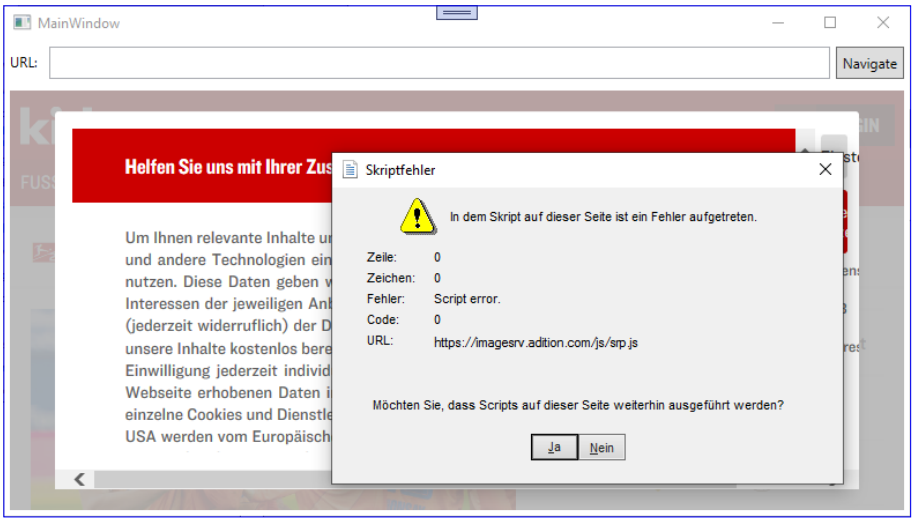


Bild 3: Javascript-Fehler im WPF Webbrowser-Steuerelement

Beim Laden verschiedener Webseiten sind in unserem WPF Webbrowser-Steuerelement durch Javascript ausgelöste Fehler aufgetreten (siehe Bild 3). Warum treten diese hier auf, obwohl sie in anderen Browsern nicht in Erscheinung treten? Das liegt daran, dass das WPF Webbrowser-Steuerelement schlicht nicht so umfangreiche Funktionen wie eigenständige Browser hat. Er ist somit nicht zum Anzeigen aller möglichen Seiten geeignet, vor allem aber nicht von Seiten, die mehr oder weniger Gebrauch von Javascript-Code machen. Sie sollten das WPF Webbrowser-Steuerelement also eher für solche Einsatzzwecke nutzen, wo Sie selbst den anzuzeigenden Inhalt per Code zusammenstellen, der entweder keinen Skript-Code enthält oder wo Sie zumindest die Kontrolle über eventuell auftretende Fehler haben.

Navigationsschaltflächen programmieren

Wenn schon das WPF Webbrowser-Steuerelement verwendet wird, soll der Benutzer auch die Möglichkeit haben, zur vorherigen und zur nachfolgend aufgerufenen Seite zu navigieren. Dazu fügen wir dem Fenster zwei Schaltflächen hinzu, die wir im XAML-Code wie folgt zusammen mit der Schaltfläche **btnNavigate** in einem **StackPanel**-Element definieren:

```
<StackPanel Grid.Column="2" Orientation="Horizontal">
    <Button x:Name="btnNavigate" Grid.Column="2" Click="btnNavigate_Click">Navigate</Button>
    <Button Command="NavigationCommands.BrowseBack">
        <Image Source="images/nav_left.png" Width="16" Height="16" />
    </Button>
    <Button Command="NavigationCommands.BrowseForward">
        <Image Source="images/nav_right.png" Width="16" Height="16" />
    </Button>
</StackPanel>
```

Für die beiden **Button**-Elemente haben wir jeweils ein **Image**-Element angegeben. Die beiden dazu benötigten und im **Source**-Attribut angegebenen Bilddateien speichern wir im Projektmappen-Explorer im Verzeichnis **images**. Die Bilddateien ziehen Sie dazu einfach in das zuvor im Projekt angelegte Verzeichnis hinein. Danach stellen Sie noch sicher, dass die Eigenschaften für die Bilddateien korrekt eingestellt sind. Dazu markieren Sie die

Navigieren im Frame-Steuerelement

Wenn Sie auf einfache Weise verschiedene Seiten in einer WPF-Benutzeroberfläche anzeigen wollen, bietet sich die Nutzung des Frame-Steuerelements zur Anzeige von Page-Elementen an. Das Frame-Steuerelement ist dabei der Rahmen, die Page-Elemente sind im Prinzip Fenster ohne Fensterrahmen. Dieser Artikel zeigt, welche Möglichkeiten das Frame-Element bietet, wie Sie Page-Elemente darin anzeigen und wie die Steuerung der Anzeige verschiedener Page-Elemente funktioniert.

Wenn Sie eine einfache WPF-Anwendung bauen, die ein Ribbon zur Steuerung der Anwendung enthält und einen Bereich, der die zu bearbeitenden Daten anzeigen soll, ist das **Frame**-Steuerelement in Kombination mit **Page**-Elementen für die einzelnen Seiten eine praktische Lösung. Das Ribbon würden Sie dabei in der oberen Zeile eines **Grid**-Elements darstellen, das **Frame**-Element in der zweiten Zeile. Wir wollen im Beispiel nicht direkt mit einem Ribbon arbeiten, sondern mit einfachen Schaltflächen, also fügen wir diese in der ersten Zeile eines Grids ein und das Frame-Element direkt darunter (siehe Bild 1).

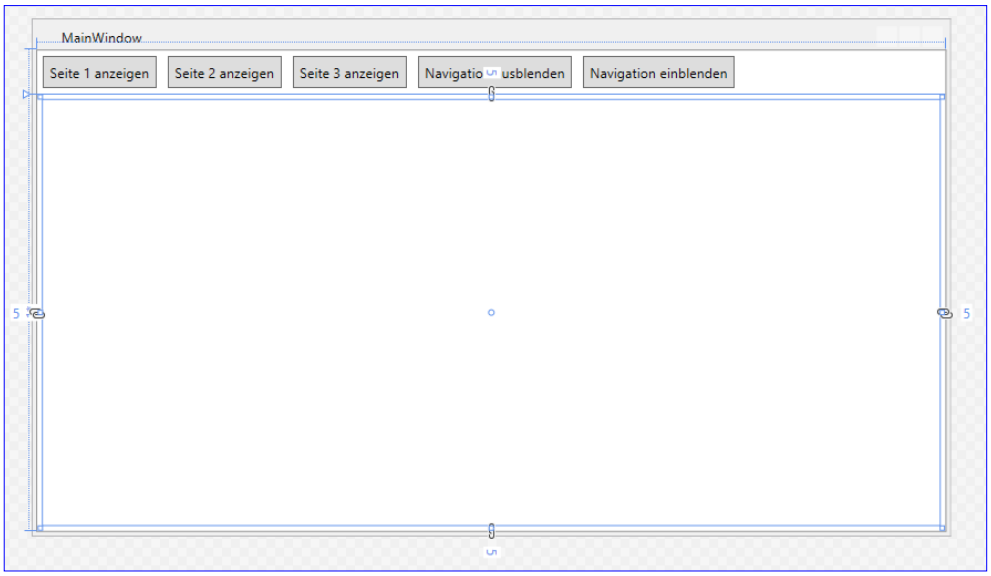


Bild 1: **Frame**-Element mit Schaltflächen

Der Code für die folgenden Beispiele ist wie folgt aufgebaut:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
   <StackPanel Grid.Row="0" Orientation="Horizontal">
        <Button x:Name="btnSeite1" Content="Seite 1 anzeigen" Click="btnSeite1_Click"/>
        <Button x:Name="btnSeite2" Content="Seite 2 anzeigen" Click="btnSeite2_Click"/>
        <Button x:Name="btnSeite3" Content="Seite 3 anzeigen" Click="btnSeite3_Click"/>
        ... weitere Steuerelemente
    </StackPanel>
```

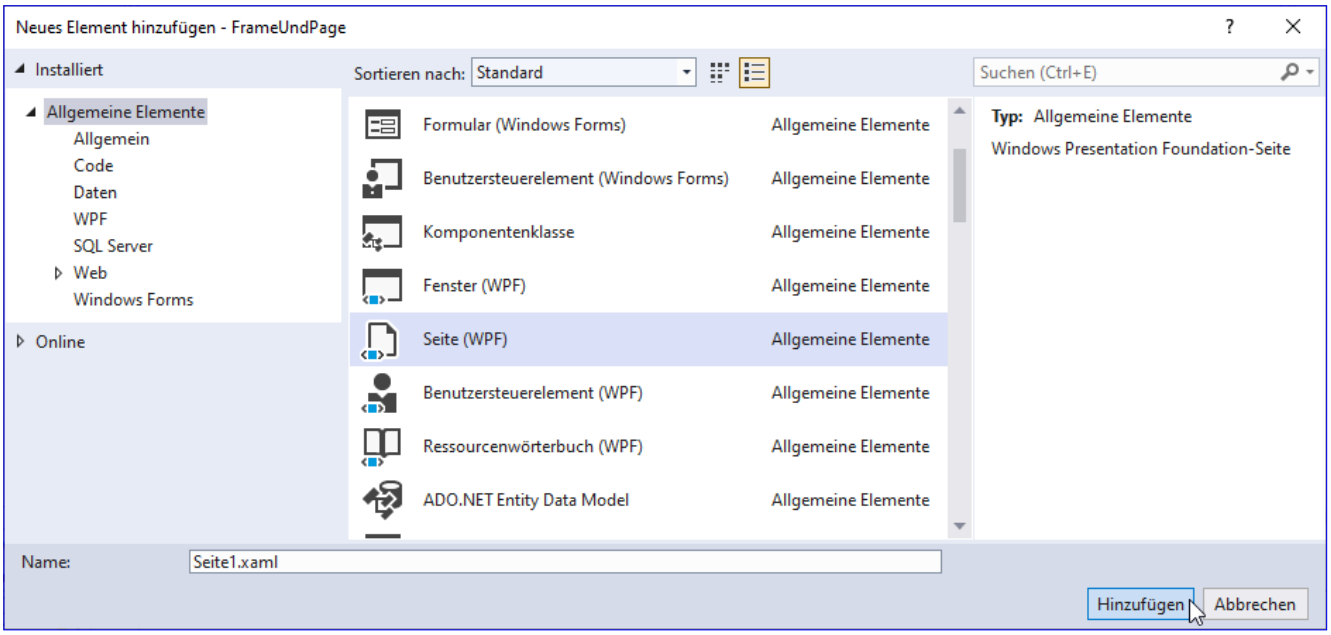


Bild 2: Hinzufügen eines **Page**-Elements

```
    <Frame x:Name="fra" Grid.Row="1"></Frame>
</Grid>
```

Wir definieren hier also ein **Grid** mit zwei Zeilen, von denen die erste die in einem **StackPanel** angeordneten Schaltflächen enthält und die zweite das **Frame**-Steuerelement. Wie kommen nun die Pages ins Spiel? Diese fügen wir zunächst als **Page**-Elemente zum Projekt hinzu – zunächst einmal drei Stück.

Dazu nutzen wir den Befehl **Hinzufügen|Seite (WPF)...** aus dem Kontextmenü des **Projekt**-Elements im Projektmappen-Explorer und geben im nun erscheinenden Dialog **Neues Element hinzufügen** den Namen des hinzuzufügenden **Page**-Elements an – hier **Seite1.xaml** (siehe Bild 2).

Damit wir in der Anwendung erkennen können, welches **Page**-Element wir jeweils anzeigen, fügen wir jedem **Page**-Element ein Label hinzu – beispielsweise mit dem Text **Seite 1**:

```
<Page x:Class="Seite1" ... Title="Seite1">
    <Grid>
        <Label>Seite 1</Label>
    </Grid>
</Page>
```

Im Gegensatz zur Darstellung eines Fensters (**Window**-Elements) mit dem Fensterrahmen und dem weißen Hintergrund erscheint das **Page**-Element ohne ausgeprägten Rahmen und hat auch keinen Hintergrund, das heißt, es ist transparent (siehe Bild 3). Darüber hinaus enthält es jedoch in der XAML-Definition die gleichen Eigenschaften wie ein **Window**-Element. Wenn Sie beispielsweise Ressourcen definieren wollen, um seitenweit

gültige Eigenschaften für Steuerelemente zu definieren, nutzen Sie nicht das Element **Window.Resources**, sondern **Page.Resources**:

```
<Page x:Class="Seite2"
      ...
      Title="Seite2">
    <Page.Resources>
        <Style TargetType="Label">
            <Setter Property="Margin" Value="5"></Setter>
        </Style>
    </Page.Resources>
    ...
```

Page-Element im Frame-Element anzeigen

Wie sorgen wir nun für die Anzeige von **Page**-Elementen im **Frame**-Element?

Dazu können wir verschiedene Techniken nutzen:

- **Content**-Eigenschaft
- **Navigate**-Methode
- Hyperlinks
- Navigationsjournal

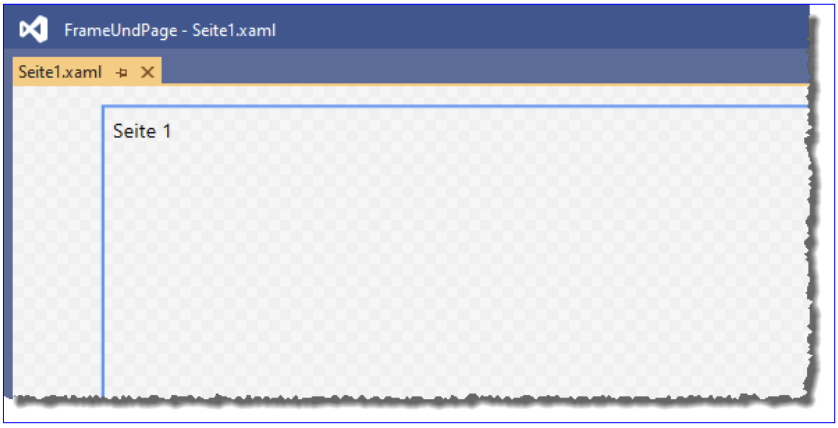


Bild 3: Darstellung eines **Page**-Elements im Entwurf

Page-Element mit Content anzeigen

Die erste Möglichkeit der Anzeige eines **Page**-Elements ist die **Content**-Eigenschaft des **Frame**-Elements.

Bild 4: Anzeigen eines **Page**-Elements im **Frame**-Element

Wir hinterlegen die folgende Methode für die Schaltfläche **btnSeite1**:

```
Private Sub btnSeite1_Click(sender As Object, e As RoutedEventArgs)
    Dim pgeSeite1 As New Seite1
    fra.Content = pgeSeite1
End Sub
```

Diese erstellt eine neue Instanz des **Page**-Elements **Seite1** und weist diese der Eigenschaft **Content** des Frame-Elements namens **fra** zu. Das Ergebnis sehen Sie in Bild 4.

Auf die gleiche Weise können wir auch die übrigen Seiten anzeigen. Eigentlich reicht diese Technik schon aus, wenn Sie einfach nur über ein Ribbon die verschiedenen Elemente der Benutzeroberfläche einer Anwendung einblenden möchten.

Seiten ausblenden beziehungsweise schließen

Vielleicht möchten Sie einen geöffneten Bereich wieder schließen – beispielsweise, indem Sie eine dafür vorgesehene Schaltfläche mit dem Text **Schließen** oder **OK** betätigen.

Eine Möglichkeit ist das Einstellen der **Content**-Eigenschaft des **Frame**-Elements auf den Wert **Nothing**. Damit wird das **Frame**-Element geleert. Fügen wir dem Hauptfenster eine Schaltfläche namens **btnLeeren** hinzu, können wir für ihr **Click**-Ereignis die folgende Methode hinterlegen:

```
Private Sub btnLeeren_Click(sender As Object, e As RoutedEventArgs)
    fra.Content = Nothing
End Sub
```

Wenn wir nun eines der **Page**-Elemente geladen haben und diese Schaltfläche betätigen, wird das **Frame**-Element geleert.

Wenn Sie das **Page**-Element jedoch vom geöffneten **Page**-Element aus schließen wollen, beispielsweise per Schaltfläche, dann haben Sie ebenfalls mehrere Möglichkeiten – zum Beispiel die folgenden:

- Durch Übergabe eines Verweises auf das **Frame**-Steuerelement an die aufgerufene Seite und Leeren der Content-Eigenschaft (siehe nächster Abschnitt).
- Durch Verwenden der **NavigationService**-Klasse (siehe weiter unten unter den Methoden der **NavigationService**-Klasse).

Seite schließen per übergebenem Frame-Verweis

Dieses Beispiel schauen wir uns anhand der Schaltfläche **btnSeite2** des Fensters **MainWindow** und dem **Page**-Element **Seite2** an. Für dieses **Page**-Element legen wir eine Variable namens **m_Frame** an, die einen Verweis auf das übergeordnete **Frame**-Element aufnehmen soll. Außerdem erstellen wir eine Konstruktor-Methode, die einen Verweis auf das **Frame**-Element entgegennimmt und diesen in der Variablen **m_Frame** speichert:

```
Class Seite2
    Private m_Frame As Frame

    Public Sub New(fra As Frame)
```

Nicht-generische Auflistungen unter VB.NET

Unter VB6/VBA waren die bekanntesten und sofort einsetzbaren Auflistungsklassen das Array und die Collection. Sie boten einfache Möglichkeiten zum Hinzufügen von Objektverweisen oder Werten, zum Löschen derselben oder auch zum Bearbeiten. Wer mehr brauchte, konnte sich noch des Dictionary-Objekts bedienen, was aber schon das Einbinden eines weiteren Verweises erforderte. Unter .NET gibt es bei Auflistungsklassen einen wichtigen Unterschied: nicht-generische Auflistungsklassen, die nicht stark typisiert sind und generische Auflistungsklassen, die stark typisiert sind. Der vorliegende Artikel betrachtet die wichtigsten nicht-generischen Auflistungsklassen und ihre Eigenschaften.

Hinweis zu den Beispielen

Die Beispiele dieses Artikels haben wir in LINQPad5 erstellt und ausprobiert. Das ist eine schnellere Möglichkeit als ein Visual Studio Projekt zu erstellen, das Sie für jeden Test neu kompilieren und starten müssen. Weitere Infos zu LINQPad5 finden Sie beispielsweise im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** (**www.datenbankentwickler.net/100**).

Wir verwenden hier unter **Language** den Wert **VB Program** und rufen unsere Beispielprozeduren jeweils von der Hauptprozedur **Sub Main** auf.

Was sind Auflistungsklassen?

Auflistungsklassen sind Klassen, mit denen Sie Objekte erstellen können, die das Speichern einer Reihe anderer Objekte erlauben. .NET bietet einige Auflistungsklassen für verschiedene Anwendungszwecke an. Sie sind aber nicht einfach nur Sammlungen von Daten, sondern bieten auch die Infrastruktur an, um diese Daten an verschiedenen Stellen auf spezielle Art zu nutzen – zum Beispiel als Datenquelle von Steuerelementen unter WPF.

Die Auflistungsklassen implementieren verschiedene Interfaces, die Eigenschaften, Methoden und Ereignisse für die unterschiedlichen Zwecke vorsehen. Welche Interfaces eine Auflistungsklasse implementiert, können Sie im Objektkatalog von Visual Studio nachsehen (**Strg + Alt**

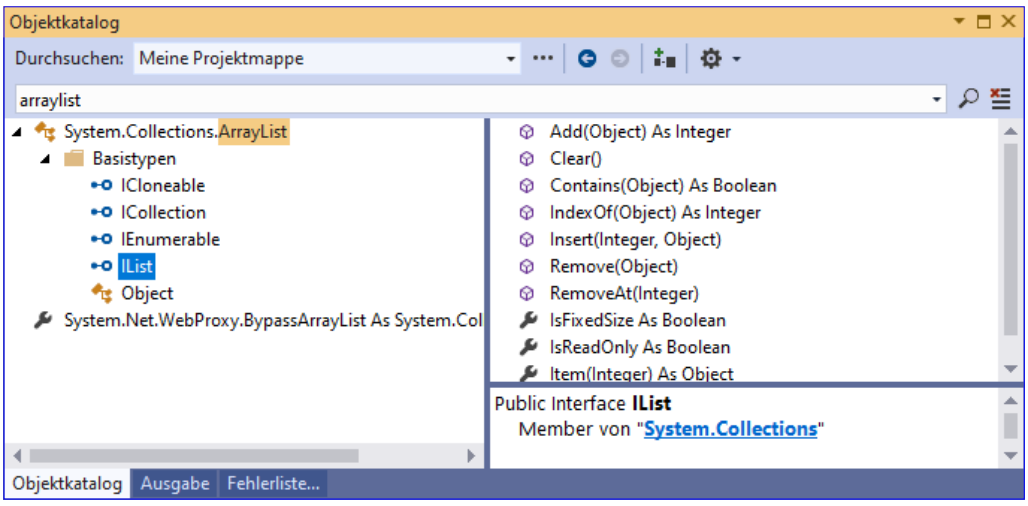


Bild 1: Eigenschaften, Methoden und Ereignisse der **IList**-Schnittstelle

+ J). Bild 1 zeigt die Basistypen beziehungsweise Schnittstellen der Auflistung **ArrayList**. Hier finden Sie auch die Eigenschaften, Methoden und Ereignisse der jeweiligen Auflistungsklasse.

Die **ArrayList**-Auflistungsklasse implementiert beispielsweise die folgenden Interfaces:

- **IEnumerable**: Stellt die Eigenschaft **GetEnumerator** bereit, die allerdings nicht explizit verwendet wird, sondern das Durchlaufen der enthaltenen Elemente in einer **For Each**-Schleife erlaubt
- **ICollection**: Liefert vor allem die Eigenschaft **Count**.
- **IList**: Enthält Methoden, um die Liste zu verwalten – also um Elemente hinzuzufügen, zu entfernen und abzufragen, die Liste zu leeren und weitere Eigenschaften.
- **ICloneable**: Liefert die Methode **Clone**.

Anhand der Schnittstellen, die eine Auflistungsklasse implementiert, können Sie erkennen, welche Möglichkeiten diese bietet. Andere Auflistungsklassen verwenden noch weitere Schnittstellen wie beispielsweise **IDeserializationCallback**, **IDictionary** oder **ISerializable**.

Dadurch, dass die verschiedenen Auflistungsklassen die Schnittstellen implementieren, haben diese einige Gemeinsamkeiten, auf die Sie zählen können – zum Beispiel die Möglichkeit, die Menge der enthaltenen Elemente mit **Count** zu ermitteln oder diese mit **For...Each** zu durchlaufen.

Namespaces mit Auflistungsklassen

Die Auflistungsklassen sind in eigenen Namespaces organisiert. Der Namespace **System.Collections** enthält beispielsweise die Auflistungsklassen **ArrayList, HashTable**, **SortedList** oder **Stack**. Die generischen Auflistungsklassen, über die wir in einem weiteren Artikel namens **Generische Auflistungen unter VB.NET** berichten (**www.datenbankentwickler.net/268**), finden Sie im Namespace **System.Collections.Generic**. Die **ObservableCollection**, die oft als Datenquelle für WPF-Steuerelemente dient, stammt aus dem Namespace **System.Collections.Objectmodel**. Die jeweiligen Schnittstellen sind übrigens auch in den jeweiligen Namespaces zu finden.

Nicht-generische versus generische Auflistungsklassen

Nicht-generische Auflistungsklassen unter VB.Net sind, wie oben bereits angedeutet, nicht »stark typisiert«. Was heißt das überhaupt? Das bedeutet, dass wir nicht explizit den Typ der in der Auflistungsklasse enthaltenen Elemente angeben. Deshalb wird hier der Typ **Object** verwendet. Im Gegensatz dazu geben wir bei generischen Auflistungsklassen den Typ der in der Auflistungsklasse enthaltenen Elemente explizit an, zum Beispiel **String** oder auch komplexere Datentypen. Ein Beispiel, das wir schon oft genutzt haben, sind die Auflistungsklassen mit Entitäten aus einer Datenbank, die wir als Datenquelle für die Steuerelemente unter WPF nutzen – diese haben beispielsweise den Datentyp **Kunde** oder **Produkt**. Nicht generische Auflistungsklassen können Sie hingegen auch Elemente unterschiedlicher Datentypen hinzufügen.

Beispiele für nicht-generische Auflistungsklassen

Es gibt beispielsweise die folgenden nicht-generischen Auflistungsklassen:

- **ArrayList**: eine sortierte Liste von Elementen. Die Sortierung erfolgt nach der Reihenfolge des Hinzufügens der Elemente.
- **Queue**: Die **Queue**-Auflistungsklasse ermöglicht das Verwalten von Einträgen nach dem First-In, First-Out-Prinzip.
- **Stack**: Die **Stack**-Auflistungsklasse arbeitet ähnlich wie Queue, aber andersherum: Sie stellt die zuletzt hinzugefügten Elemente als Erste wieder bereit.
- **HashTable**: Die **HashTable**-Auflistungsklasse speichert Schlüssel-Wert-Paare und bietet umfangreiche Möglichkeiten, diese abzufragen.

Die ArrayList-Auflistung

Wir schauen uns einige Beispiele für den Umgang mit Auflistungsklassen anhand der **ArrayList**-Auflistungsklasse an. Später stellen wir weitere nicht-generische Auflistungsklassen vor und beschreiben deren Besonderheiten.

Die wichtigsten Methoden und Eigenschaften einer **ArrayList** sind die folgenden:

- **Count**: Liefert die Menge der enthaltenen Elemente.
- **CopyTo**: Kopiert den Inhalt der **ArrayList**-Auflistungsklasse in ein Array.
- **Contains**: Erwartet den Wert eines Elements als Parameter und gibt an, ob dieses enthalten ist.
- **ToArray**: Kopiert den Inhalt der **ArrayList**-Auflistungsklasse in ein Array des Typs **Object**.
- **IndexOf**: Liefert den Index des als Parameter angegebenen Elements. Der Index ist immer nullbasiert.
- **GetRange**: Erstellt eine neue **ArrayList**-Auflistungsklasse nur mit den Elementen, die durch den Startindex und die Menge als Parameter angegeben wurden.
- **Sort**: Sortiert die Elemente der ArrayList.
- **Reverse**: Kehrt die Sortierung der Elemente der ArrayList um.
- **Item**: Liest das Element mit dem angegebenen Index aus oder setzt diesen.
- **Clear**: Leert die Auflistungsklasse.

Elemente zu einer nicht-generischen Auflistungsklasse hinzufügen

Die **ArrayList**-Auflistungsklasse ist eine einfache, nicht-generische Auflistungsklasse, daher verwenden wir diese zunächst für die Beispiele. Sie können der **ArrayList** einfach mit der **Add**-Methode neue Elemente hinzufügen, die verschiedenen Typs sein können. Sie können sogar eine neue **ArrayList** als Element der **ArrayList** hinzufügen:

```
Sub ArrayListErstellen
    Dim Gemischt As New ArrayList
    Gemischt.Add(10)
    Gemischt.Add("10")
    Gemischt.Add("André")
    Gemischt.Add(New ArrayList)
End Sub
```

Außerdem sehen Sie, dass wir die Zahl **10** einmal als Zahl und einmal als Zeichenkette hinzugefügt haben. Das weisen die folgenden beiden Anweisungen nach, die Sie der Prozedur hinzufügen:

```
Debug.Print(Gemischt.Item(0).GetType.ToString)
Debug.Print(Gemischt.Item(1).GetType.ToString)
```

Das Ergebnis lautet:

```
System.Int32
System.String
```

Dass Sie nicht wissen, welcher Objekttyp Sie in einem Element einer nicht-generischen Auflistungsklasse erwartet, ist übrigens einer der Gründe, warum Sie eine generische Auflistungsklasse verwenden sollten – außer, Sie benötigen explizit eine nicht-generische Liste.

Sie können auch die Elemente eines Array zu einer **ArrayList**-Auflistungsklasse hinzufügen. Im folgenden Beispiel fügen wir zunächst zwei Elemente wie zuvor per **Add** hinzu. Dann erstellen wir ein Array namens **arr** mit zwei weiteren Elementen. Die Elemente aus diesem Array fügen wir dann mit der **AddRange**-Methode hinzu:

```
Dim Produkte As ArrayList
Dim arr() As String
Produkte = New ArrayList
Produkte.Add("Datenbankentwickler")
Produkte.Add("Access im Unternehmen")
arr = {"Access [basics]", "Access und Formulare"}
Produkte.AddRange(arr)
```

Elemente einer Auflistungsklasse durchlaufen

Die einfachste Methode, die Elemente einer Auflistungsklasse zu durchlaufen, ist die **For Each**-Schleife. Sie verwenden darin eine Variable, welche das jeweilige Element aufnehmen soll und die zu durchlaufende Auflistungsklasse. Die Elemente der soeben angelegten Auflistungsklasse können Sie wie folgt durchlaufen und beispielsweise im Ausgabe-Bereich ausgeben:

```
Dim Produkt As String
For Each Produkt In Produkte
    Debug.Print(Produkt)
Next Produkt
```

Auf Elemente einer Auflistungsklasse per Index zugreifen

Weiter oben haben wir gleich gezeigt, wie Sie über den Index auf die Elemente der Auflistungsklassen zugreifen können. Sie verwenden dazu die **Item**-Eigenschaft und geben den Index des gesuchten Elements an, der immer 0-basiert ist:

```
Debug.Print (Produkte(0))
```

Damit können wir auf einzelne Elemente innerhalb einer weiteren Schleife zugreifen, nämlich **For...Next**:

```
Dim i As Integer
For i = 0 To Produkte.Count - 1
    Debug.Print(Produkte(i))
Next i
```

Elemente an bestimmten Stellen einfügen

Mit der **Add**- oder der **AddRange**-Methode fügen wir Elemente immer hinten an die Liste an. Sie können aber auch Elemente mitten in der **ArrayList**-Auflistungsklasse anfügen. Dazu nutzen Sie die **Insert**- oder die **InsertRange**-Methode. Im folgenden Beispiel fügen wir ein Element mit **Add** an, dann eines mit **Insert** vor diesem Element an der mit dem ersten Parameter angegebenen Position **0**.

Die Elemente des Arrays **arr** fügen wir dann zwischen diese beiden Elemente an Position **1** ein:

```
Dim Produkte As ArrayList
Dim arr() As String
Produkte = New ArrayList
Produkte.Add("Datenbankentwickler")
Produkte.Insert(0, "Access im Unternehmen")
arr = {"Access [basics]", "Access und Formulare"}
Produkte.InsertRange(1, arr)
Produkte.Dump
```

Generische Auflistungen unter VB.NET

Der Unterschied zwischen generischen Auflistungen und nicht-generischen Auflistungen ist, dass Sie für die Elemente einer generischen Auflistung einen konkreten Datentyp vorgeben – das nennt man auch stark typisiert. Bei nicht-generischen Auflistungen nutzen Sie den Objekttyp Object und können somit Werte verschiedener Typen hinzufügen. Dieser Artikel stellt die Grundlagen zu generischen Auflistungsklassen vor und erläutert ihre Funktionsweise.

Hinweis zu den Beispielen

Die Beispiele dieses Artikels haben wir in LINQPad5 erstellt und ausprobiert. Das ist eine schnellere Möglichkeit als ein Visual Studio Projekt zu erstellen, das Sie für jeden Test neu kompilieren und starten müssen. Weitere Infos zu LINQPad5 finden Sie beispielsweise im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** (**www.datenbankentwickler.net/100**). Wir verwenden hier unter **Language** den Wert **VB Program** und rufen unsere Beispielprozeduren jeweils von der Hauptprozedur **Sub Main** auf.

Generische Auflistungen

Eine generische Auflistung erkennen Sie primär daran, dass Sie diese mit der Angabe des Datentyps für die enthaltenen Werte deklarieren und initialisieren. Dazu verwenden wir das Schlüsselwort **Of** zusammen mit dem gewünschten Datentyp. Die gängigen generischen Auflistungen werden in den Namespaces **System.Collections.Generic** und **System.Collections.ObjectModel** aufgeführt.

Zu den nicht-generischen Auflistungen gibt es in den meisten Fällen ein Pendant unter den generischen Auflistungen. Diese sind leider nicht entsprechend benannt. So ist das generische Gegenstück zu der nicht-generischen Auflistungsklasse **HashTable** die **Dictionary**-Klasse und **List** ist das Pendant zu **ArrayList**.

Es gibt aber auch generische Auflistungsklassen ohne nicht-generische Entsprechung.

Was sind generische Auflistungsklassen?

Generische Auflistungen sind auch Klassen, allerdings solche, die das Speichern einer Reihe anderer Objekte erlauben. .NET bietet einige Auflistungsklassen für verschiedene Anwendungszwecke an. Sie sind aber nicht einfach nur Sammlungen von Daten, sondern bieten auch die Infrastruktur an, um diese Daten an verschiedenen Stellen auf spezielle Art zu nutzen – zum Beispiel als Datenquelle von Steuerelementen unter WPF. Die Auflistungsklassen implementieren verschiedene Interfaces, die Eigenschaften, Methoden und Ereignisse für

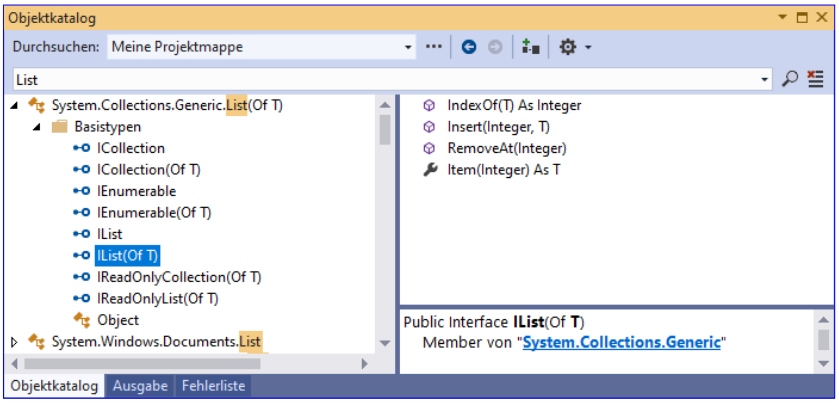


Bild 1: Die **List(Of T)**-Auflistungsklasse im Objektkatalog

die unterschiedlichen Zwecke vorsehen. Welche Interfaces eine Collection implementiert, können Sie im Objektkatalog von Visual Studio nachsehen (**Strg + Alt + J**).

Bild 1 zeigt die Basistypen beziehungsweise Schnittstellen der Auflistungsklasse **List(Of T)**, wobei **T** dem Datentyp der Elemente der Auflistung entspricht. Im Objektkatalog finden Sie auch die Eigenschaften, Methoden und Ereignisse der jeweiligen Auflistungen.

Die **List(Of T)**-Auflistungsklasse implementiert beispielsweise die folgenden Interfaces:

- **IEnumerable**: Stellt die Eigenschaft **GetEnumerator** bereit, die allerdings nicht explizit verwendet wird, sondern das Durchlaufen der enthaltenen Elemente in einer **For Each**-Schleife erlaubt
- **ICollection**: Liefert vor allem die Eigenschaft **Count**.
- **IList**: Enthält Methoden, um die Liste zu verwalten – also um Elemente hinzuzufügen, zu entfernen und abzufragen, die Liste zu leeren und weitere Eigenschaften.

Beispiele für generische Auflistungsklassen

In diesem Artikel stellen wir verschiedene generische Auflistungsklassen vor:

- **List(Of T)**: Entspricht der nicht-generischen Auflistungsklasse **ArrayList**.
- **Dictionary(Of TKey, TValue)**: Entspricht der nicht-generischen Auflistungsklasse **HashTable**.

Die List(Of T)-Auflistungsklasse

Die **List(Of T)**-Auflistungsklasse ist das generische Pendant zur **ArrayList**-Auflistungsklasse, das heißt, dass für Ihre Elemente exakt festgelegt wird, welchen Typ diese aufweisen. Das nennt sich auch stark typisiert.

In den folgenden Beispielen wollen wir mit einer kleinen Beispielklasse namens Produkt arbeiten, die wir wie folgt deklarieren:

```
Public Class Produkt
    Public Produktname As String
    Public Einzelpreis As Decimal
End Class
```

Ein paar Beispielobjekte auf Basis dieser Klasse erstellen wir mit den folgenden Anweisungen:

```
Dim Produkt1 As New Produkt With {.Produktname = "Access im Unternehmen", .Einzelpreis = 124}
Dim Produkt2 As New Produkt With {.Produktname = "Access [Basics]", .Einzelpreis = 69}
Dim Produkt3 As New Produkt With {.Produktname = "Datenbankentwickler", .Einzelpreis = 129}
```

Generische Auflistungsklasse deklarieren und initialisieren

Bei der Deklaration einer generischen Auflistungsklasse geben Sie als Parameter das Schlüsselwort **Of** und den Typ des enthaltenen Objekttyps an. Für unser Beispiel verwenden wir eine Auflistung namens **Produkte** mit dem Typ **Produkt**. Die Deklarationszeile sieht so aus:

```
Dim Produkte As List(Of Produkt)
```

Wenn Sie die Initialisierung separat gestalten wollen, gelingt dies so:

```
Produkte = New List(Of Produkt)
```

Sie können natürlich auch beides in einer Zeile unterbringen:

```
Dim Produkte As New List(Of Produkt)
```

Elemente zu einer generischen Auflistungsklasse hinzufügen

Die oben erstellten Objekte fügen wir mit der **Add**-Methode zur Auflistungsklasse **Produkte** hinzu:

```
With Produkte
    .Add(Produkt1)
    .Add(Produkt2)
    .Add(Produkt3)
End With
```

Generische Auflistungsklasse schnell füllen per Collection Initializer

Wenn Sie die Liste nicht etwa per Entity Framework aus einer **DbSet**-Auflistung füllen wollen, können Sie den sogenannten **Collection Initializer** nutzen. Dabei geben Sie hinter der eigentlichen Initialisierung das **From**-Schlüsselwort und dann in geschweiften Klammern eine Liste der einzufügenden Elemente an:

```
Dim Produkt1 As New Produkt With {.Produktname = "Access im Unternehmen", .Einzelpreis = 124}
Dim Produkt2 As New Produkt With {.Produktname = "Access [Basics]", .Einzelpreis = 69}
Dim Produkt3 As New Produkt With {.Produktname = "Datenbankentwickler", .Einzelpreis = 129}
Dim Produkte As New List(Of Produkt) From {Produkt1, Produkt2, Produkt3}
Debug.Print("Anzahl Produkte: " + Produkte.Count.ToString)
```

Elemente einer generischen Auflistungsklasse mit For Each durchlaufen

Die einfachste Methode, die Elemente einer generischen Auflistungsklasse zu durchlaufen, ist die **For Each**-Schleife. Sie verwenden darin eine Variable, welche das jeweilige Element aufnehmen soll und die zu durchlaufende Auflistungsklasse. Die Elemente der soeben angelegten Auflistungsklasse können Sie wie folgt durchlaufen und beispielsweise im Ausgabe-Bereich ausgeben – ausgehend von einer wie oben beschrieben erstellten Auflistung namens **Produkte**:

```
For Each Produkt In Produkte
    Debug.Print(Produkt.Produktname + " " + Produkt.Einzelpreis.ToString)
Next Produkt
```

Das Ergebnis lautet:

```
Access im Unternehmen 124
Access [Basics] 69
Datenbankentwickler 129
```

Auf Elemente einer generischen Auflistungsklasse per Index zugreifen

Die Elemente einer **List(Of T)**-Auflistungsklasse können Sie genau wie bei der **ArrayList**-Auflistungsklasse auch über die **Item**-Eigenschaft referenzieren. Dazu können Sie beispielsweise den Index als Parameter verwenden. Wichtig ist, dass Sie damit ein Objekt des jeweiligen Typs, hier **Produkt**, zurückbekommen:

```
Dim Produkt4 As Produkt
Produkt4 = Produkte.Item(0)
Debug.Print(Produkt4.Produktname)
```

Da **Item** die Standardeigenschaft ist, können Sie sie auch weglassen und direkt den Index angeben:

```
Produkt4 = Produkte(0)
```

Sie können auch direkt über das **Item**-Element auf die Eigenschaften zugreifen:

```
Dim strProduktname As String
strProduktname = Produkte(0).Produktname
Debug.Print(strProduktname)
```

Elemente einer generischen Auflistungsklasse mit For...Next durchlaufen

Damit können wir sowohl auf einzelne Elemente innerhalb einer Schleife zugreifen, nämlich **For...Next**:

```
Dim i As Integer
For i = 0 To Produkte.Count - 1
    Debug.Print(Produkte(i).Produktname)
Next i
```

Elemente an bestimmter Stelle einfügen

Mit der **Add**-Methode fügen wir Elemente immer hinten an die Liste an. Sie können aber auch Elemente mitten in der **ArrayList**-Auflistungsklasse anfügen. Dazu nutzen Sie die **Insert**- oder die **InsertRange**-Methode. Im

folgenden Beispiel fügen wir ein Element mit **Add** an, dann eines mit **Insert** vor diesem Element an der mit dem ersten Parameter angegebenen Position **0**.

Die Elemente des Arrays **arr** fügen wir dann zwischen diese beiden Elemente an Position **1** ein:

```
Produkte.Insert(1, Produkt3)
```

Mehrere Elemente an bestimmter Stelle einfügen

Mit der **AddRange**-Methode können Sie auch mehrerer Elemente gleichzeitig hinzufügen. Voraussetzung dafür ist, dass die hinzuzufügenden Elemente sich ebenfalls in einer generischen Auflistung befinden. Im folgenden Beispiel erstellen wir eine Zielauflistung und eine, aus der die Elemente zur Zielauflistung hinzugefügt werden – in diesem Fall auch eine des Typs **List(Of T)** namens **ProdukteZumEinfuegen**. Dieser fügen wir zwei Elemente hinzu. Dann fügen wir mit der **InsertRange**-Methode die Elemente aus **ProdukteZumEinfuegen** zur Auflistung Produkte hinzu – und zwar an der **Index**-Position **0**:

```
Dim Produkte As New List(Of Produkt)
Dim ProdukteZumEinfuegen As New List(Of Produkt)
With Produkte
        .Add(Produkt1)
End With
With ProdukteZumEinfuegen
        .Add(Produkt2)
        .Add(Produkt3)
End With
Produkte.InsertRange(0, ProdukteZumEinfuegen)
```

Elemente aus generischen Auflistungsklassen entfernen

Um einzelne oder mehrere Elemente wieder zu entfernen, können Sie drei Möglichkeiten nutzen:

- **Remove**: Erwartet das zu entfernende Element als Parameter, im Fall eines Strings also beispielsweise **"Datenbankentwickler"**.
- **RemoveAt**: Erwartet den Index des zu entfernenden Elements.
- **RemoveRange**: Erwartet als ersten Parameter den Index des ersten zu entfernenden Elements und als zweiten Parameter die Menge der zu entfernenden Elemente.

Im folgenden Beispiel löschen wir nacheinander alle zuvor hinzugefügten Elemente wieder mit den drei Methoden **Remove**, **RemoveAt** und **RemoveRange**.

RemoveAt entfernt das Element mit dem als Parameter angegebenen Index:

```
Produkte.RemoveAt(0)
```

Remove entfernt ein Element, das mit dem zu entfernenden Produkt übereinstimmt – zu ermitteln beispielsweise über die **Produkte**-Auflistung:

```
Produkte.Remove(Produkte(0))
```

RemoveRange entfernt vom als ersten Parameter angegebenen Index aus die mit dem zweiten Parameter angegebene Menge aus der Auflistung:

```
Produkte.RemoveRange(0,2)
```

Gegenüber der **ArrayList** liefert **List(Of T)** noch eine weitere Methode zum Entfernen, nämlich **RemoveAll**. Mit dieser können Sie etwa wie folgt Elemente nach bestimmten Kriterien entfernen:

```
Produkte.RemoveAll(Function(Produkt) Produkt.Produktname.StartsWith("A"))
```

Generische Auflistungsklassen leeren

Fehlt noch der Befehl, um eine generische Auflistungsklasse zu leeren. Das erledigen wir mit der **Clear**-Methode:

```
... 'Produkt1 bis Produkt3 erstellen
Dim Produkte As New List(Of Produkt)
With Produkte
    .Add(Produkt1)
    .Add(Produkt2)
    .Add(Produkt3)
End With
Produkte.Clear
```

Elemente der generischen Auflistungsklasse durchsuchen

Da die generische Auflistungsklasse **List(Of T)** auch die Schnittstelle **IEnumerable(Of T)** implementiert, können Sie auch die dort bereitgestellten Funktionen zum Abfragen von Daten nutzen. Hier ein kleines Beispiel, mit dem wir ein einzelnes Element ermitteln – das erste, dessen Eigenschaft Produktname mit **D** beginnt. Dazu nutzen wir die Funktion **First** und übergeben einen Lambda-Ausdruck zum Ermitteln dieses Elements:

```
... 'Liste Produkte mit Elementen füllen
Dim Produkt4 As Produkt
Produkt4 = Produkte.First(Function(p) p.Produktname.StartsWith("D"))
Debug.Print(Produkt4.Produktname)
```

Dies gibt als Ergebnis **Datenbankentwickler** aus.

Auf die gleiche Weise können Sie auch mehrere Elemente ermitteln:

```
... 'Liste Produkte mit Elementen füllen
Dim ProdukteMitA As List(Of Produkt)
ProdukteMitA = Produkte.FindAll(Function(p) p.Produktname.StartsWith("A"))
For Each p As Produkt In ProdukteMitA
    Debug.Print(p.Produktname)
Next p
```

Generische Auflistungsklassen aufsteigend nach einem Feld sortieren

Mit der **Sort**-Methode können Sie eine generische Auflistungsklasse sortieren. Die folgenden Anweisungen sortieren eine wie in den vorherigen Beispielen gefüllte **List(Of T)** und geben diese in der neuen Reihenfolge aus. Im Gegensatz zu **Sort** für eine nicht-generischen Auflistungsklasse benötigen wir hier allerdings noch einen Parameter für die **Sort**-Funktion. Da **T** unter Umständen eine Klasse ist, die mehrere Eigenschaften bereitstellt, müssen wir **Sort** mitteilen, nach welchen Eigenschaften sortiert werden soll. Dies erledigen wir in Form eines Lambda-Ausdrucks. Diesem teilen wir mit, nach welchem Feld wir überhaupt sortieren wollen. In diesem Ausdruck werden mit der **CompareTo**-Funktion jeweils zwei Elemente **x** und **y** verglichen, und zwar die Eigenschaft **Produktname** von **x** mit der Eigenschaft **Produktname** von **y**:

```
Produkte.Sort(Function(x,y) x.Produktname.CompareTo(y.Produktname))
For Each p In Produkte
    Debug.Print(p.Produktname)
Next p
```

Generische Auflistungsklassen absteigend nach einem Feld sortieren

Für eine absteigende Sortierung vertauschen Sie einfach **x** und **y** im **Lambda**-Ausdruck:

```
Produkte.Sort(Function(x,y) y.Produktname.CompareTo(x.Produktname))
For Each p In Produkte
    Debug.Print(p.Produktname)
Next p
```

Die generische Auflistungsklasse Dictionary(Of TKey, TValue)

Das Gegenstück zu der nicht-generischen Auflistungsklasse **HashTable** ist die Auflistungsklasse **Dictionary(Of TKey, TValue)**. Während bei Auflistungsklassen wie **List(Of T)** nur ein Objekt je Element gespeichert werden kann, sind es bei **HashTable** und **Dictionary(Of TKey, TValue)** derer zwei. Um die **Dictionary(Of TKey, TValue)**-Auflistungsklasse typsicher zu machen, müssen Sie natürlich auch sowohl für den Schlüssel als auch für den Wert den Typ angeben, daher folgen hinter dem **Of**-Schlüsselwort gleich zwei Parameter.

Dies müssen nicht gleich benutzerdefinierte Klassen wie oben **Person** sein, sondern Sie können auch als Schlüssel den Datentyp **String** und für den Wert den Datentyp **Integer** definieren:

CSV-Datei in Klassen importieren

Unter Access war es einfach: Dort haben wir eine CSV-Datei oder auch Excel-Tabellen einfach als verknüpfte Tabelle eingebunden und konnten dann direkt auf die Daten zugreifen. Beispielsweise, um diese dann per Anfügeabfrage in die Zieltabelle zu schreiben. Unter Entity Framework ist das so nicht möglich. Natürlich könnte man das Öffnen der Datei von Hand realisieren und die einzelnen Elemente einlesen. Aber es gibt einen anderen, effizienteren Weg, den wir mit einem NuGet-Paket namens CsvHelper beschreiten können. Diese Klasse erlaubt es, die Zeilen einer CSV-Datei automatisch in die Elemente auf Basis einer entsprechenden Klasse einzulesen. Oder Sie können damit auch die Zeilen durchlaufen und die Inhalte auf Basis des Indexes oder der Spaltenüberschrift einlesen und weiterverarbeiten.

CsvHelper hinzufügen

Das erwähnte NuGet-Paket **CsvHelper** fügen wir über den Nuget-Paket-Manager hinzu, den Sie über den Menüeintrag **Projekt|NuGet-Pakete verwalten...** öffnen. Hier wechseln Sie zum Bereich **Durchsuchen** und geben **csvhelper** in das Suchfeld ein. Es erscheint der Eintrag **CsvHelper**, den wir auswählen und mit einem Klick auf die Schaltfläche **Installieren** zum Projekt hinzufügen (siehe Bild 1).

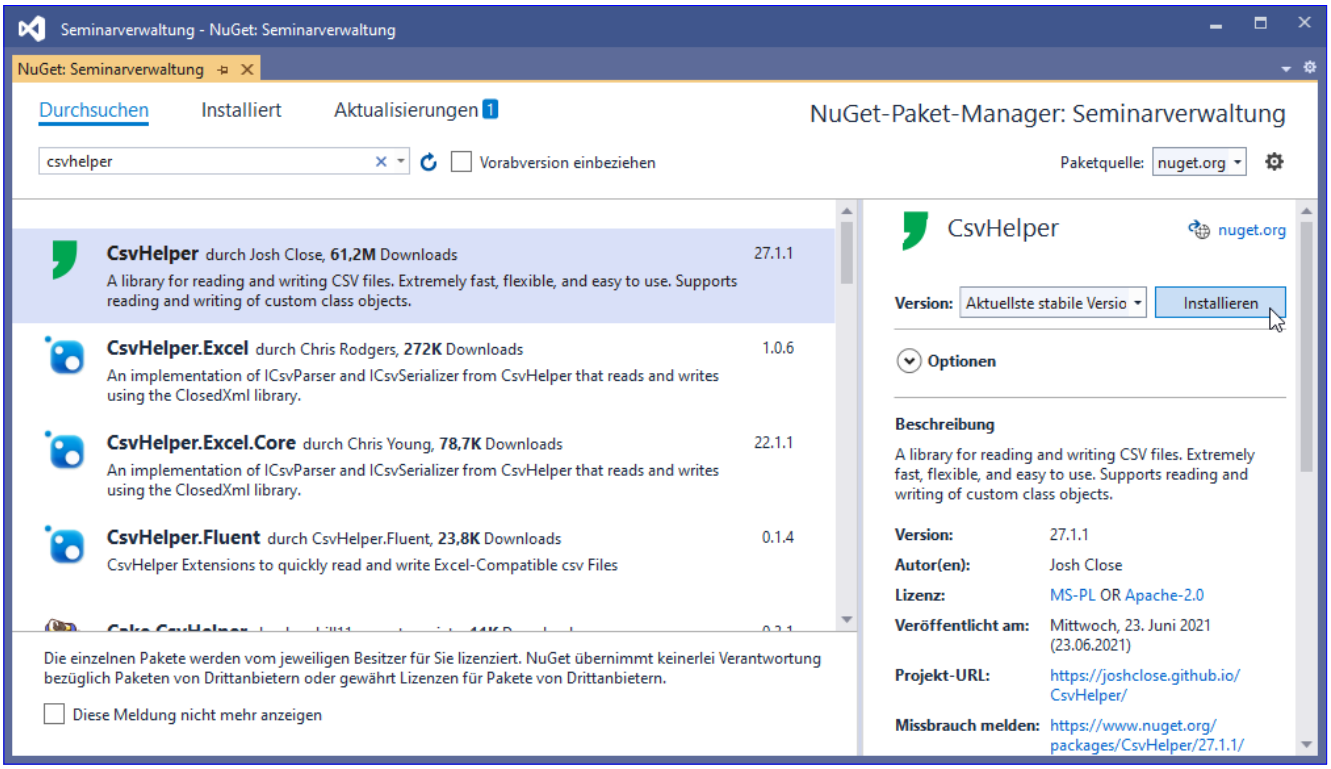


Bild 1: Hinzufügen des **CsvHelper**-Pakets

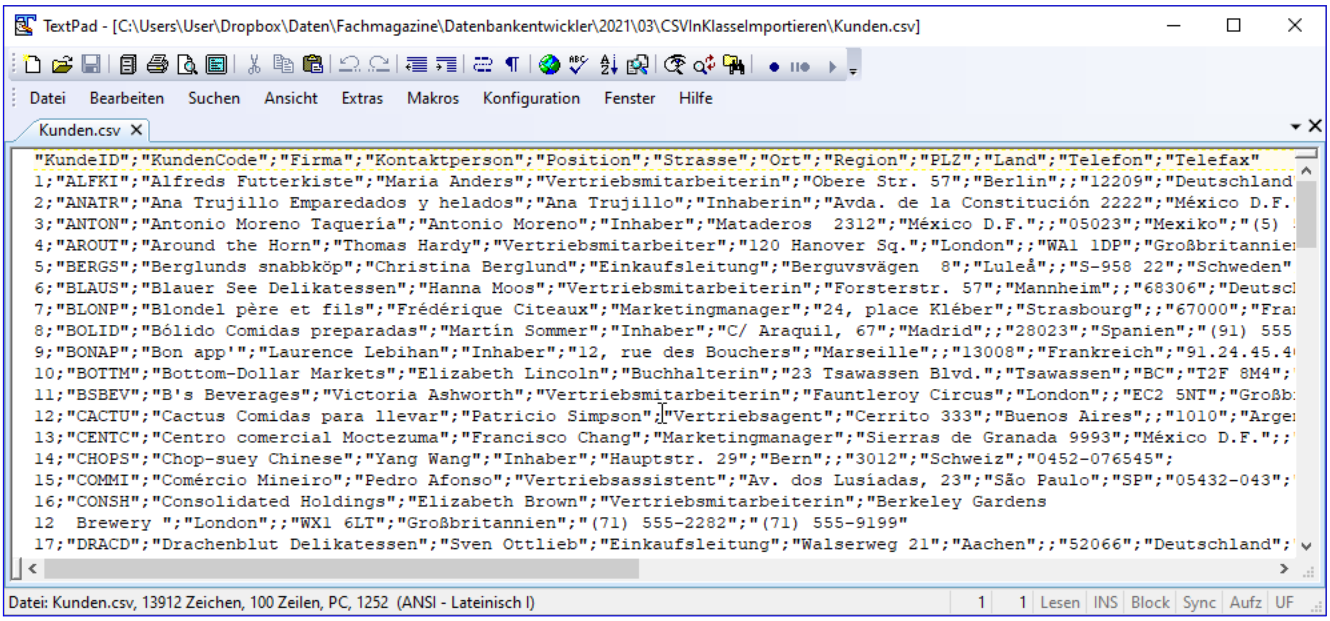


Bild 2: Beispiel-Datei

Beispieldatei für den CSV-Import

Als Beispiel für die folgenden Importtechniken wollen wir eine CSV-Datei verwenden, die wir auf Basis der Tabelle **tblKunden** aus einer Access-Beispieldatenbank exportiert haben (siehe Bild 2).

Zielklasse für die Beispieldaten definieren

Das Ziel ist, die Daten aus einer **.csv**-Datei strukturiert zu erfassen, was in Form des Parsens direkt in Elemente einer Klasse geschehen soll. Dazu benötigen wir natürlich auch eine solche, und zwar eine namens **Kunde**. Diese definieren wir wie folgt:

```
Public Class Kunde
    Public Property KundeID As Long
    Public Property KundenCode As String
    Public Property Firma As String
    Public Property Kontaktperson As String
    Public Property Position As String
    Public Property Strasse As String
    Public Property Ort As String
    Public Property Region As String
    Public Property PLZ As String
    Public Property Land As String
    Public Property Telefon As String
    Public Property Telefax As String
End Class
```

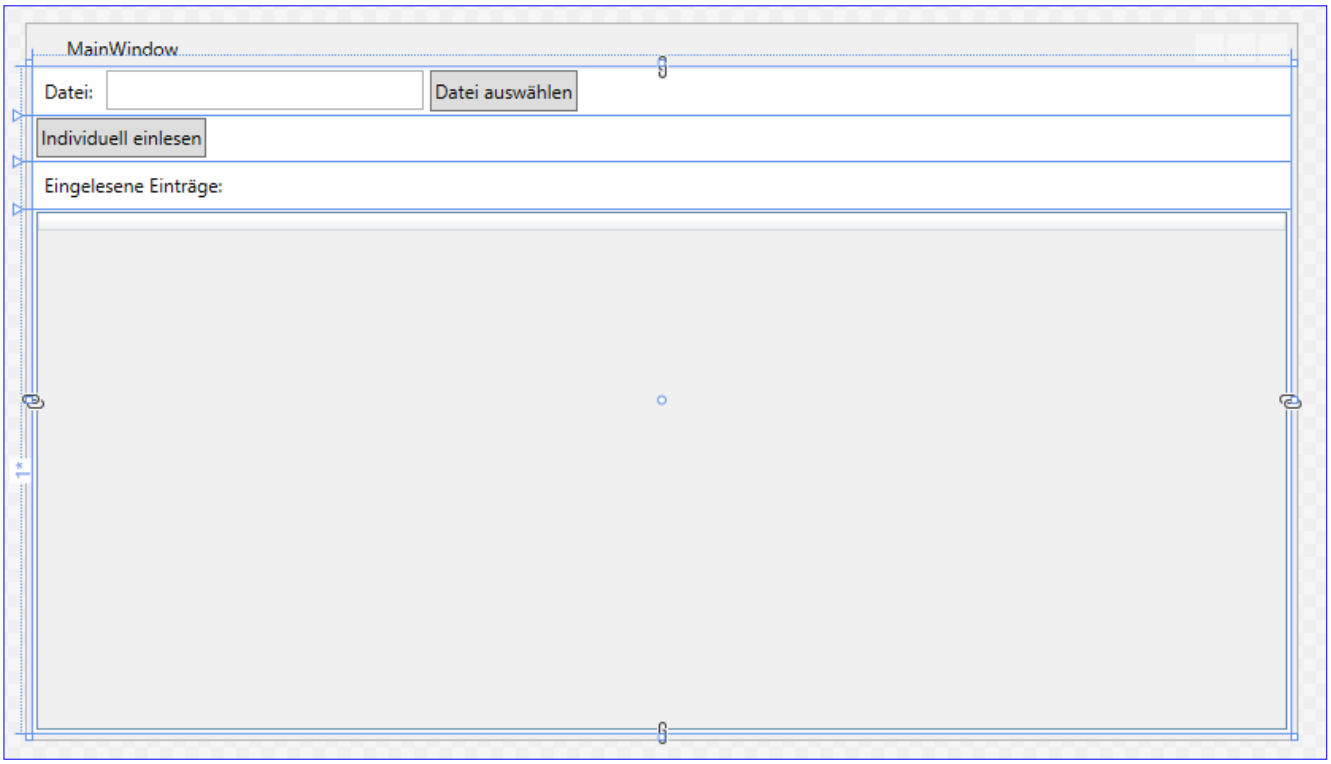


Bild 3: Benutzeroberfläche für die Beispiele

Benutzeroberfläche für die Beispiele

Die Beispiele wollen wir in der Benutzeroberfläche der Anwendung darstellen. Dazu fügen wir einige Elemente hinzu:

- Textfeld **txtImportFile** zur Eingabe/Anzeige der einzulesenden Datei
- Schaltfläche **btnChooseFile** zum Öffnen eines Dateiauswahl-Dialogs
- Schaltfläche **btnImportNoAssignment** zum Starten des ersten Beispiels für den Einlesevorgang
- DataGrid **dgrKunden** zur Anzeige der eingelesenen Daten

Die Steuerelemente ordnen wir wie in Bild 3 an.

Die Elemente definieren wir per XAML-Code wie folgt:

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
```

```
      <RowDefinition Height="*"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
   </Grid.RowDefinitions>
   <GroupBox Header="Beispiele Artikel CSV-Datei in Klassen importieren" FontWeight="Bold" Grid.Row="0">
      <ContentControl FontWeight="Normal">
         <StackPanel Orientation="Vertical">
            <Grid>
               <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="Auto"></ColumnDefinition>
                  <ColumnDefinition Width="*"></ColumnDefinition>
                  <ColumnDefinition Width="Auto"></ColumnDefinition>
               </Grid.ColumnDefinitions>
               <Label Grid.Column="0">Datei:</Label>
               <TextBox x:Name="txtImportfile" Grid.Column="1" HorizontalAlignment="Stretch"></TextBox>
               <Button x:Name="btnChooseFile" Grid.Column="2"
                  Click="btnChooseFile_Click">Datei auswählen</Button>
            </Grid>
            <StackPanel Orientation="Horizontal">
               <Button x:Name="btnImportNoAssignment"
                  Click="btnImportNoAssignment_Click">Ohne Zuordnung einlesen</Button>
            </StackPanel>
         </StackPanel>
      </ContentControl>
   </GroupBox>
   <Label Grid.Row="1">Eingelesene Einträge:</Label>
   <DataGrid x:Name="dgrKunden" Grid.Row="2"></DataGrid>
</Grid>
```

CSV-Datei auswählen

Um die Datei auszuwählen, kann der Benutzer mit der Schaltfläche **btnChooseFile** einen Dateiauswahldialog öffnen. Für den Code benötigen wir die folgende **Imports Namespace**-Anweisung:

```
Imports Microsoft.Win32
```

Die Schaltfläche löst die folgende Methode aus. Hier stellen wir das Verzeichnis auf das aktuelle Anwendungsverzeichnis ein (also das Verzeichnis, in dem sich die **.exe**-Datei befindet). Außerdem legen wir die Standarddateiendung auf **.csv** fest. Danach öffnen wir den Dialog und lesen das Ergebnis in das Textfeld **txtImportfile** ein:

```
Private Sub btnChooseFile_Click(sender As Object, e As RoutedEventArgs)
    Dim objOpenFileDialog As New OpenFileDialog
    objOpenFileDialog.InitialDirectory = AppDomain.CurrentDomain.BaseDirectory
```

```
    objOpenFileDialog.DefaultExt = ".csv"
    If (objOpenFileDialog.ShowDialog = True) Then
        txtImportfile.Text = objOpenFileDialog.FileName
    End If
End Sub
```

Einlesen mit Spaltenüberschriften, aber ohne Mapping

Die einfachste Art, die Daten aus der CSV-Datei in eine Auflistung mit Elementen des Typs **Kunde** einzulesen, ist die hinter der Schaltfläche **btnImportNoAssignment**.

Für die Anweisungen in dieser Methode benötigen wir einige Namespaces, die wir wie folgt deklarieren:

```
Imports System.Globalization
Imports System.IO
Imports System.Text
Imports CsvHelper
```

In der Methode definieren wir je ein Element der Typen **StreamReader** und **CsvReader**. Dann schreibt die Prozedur den Inhalt des Textfeldes **txtImportfile** in die Variable **strImportFile**:

```
Private Sub btnImportNoAssignment_Click(sender As Object, e As RoutedEventArgs)
    Dim strImportFile As String
    Dim objReader As StreamReader
    Dim objCsvReader As CsvReader
    Dim objRecords As Object
    Dim objKunde As Kunde
    Dim objKunden As New List(Of Kunde)
    strImportFile = txtImportfile.Text
```

Anschließend öffnen wir die CSV-Datei als neues **StreamReader**-Objekt:

```
    objReader = New StreamReader(strImportFile, Encoding.UTF8)
```

Auf dieses greift dann das neue **CsvReader**-Objekt zu und liest den Inhalt mit der **GetRecords**-Methode für den Datentyp **Kunde** in die Variable **objRecords** ein:

```
    objCsvReader = New CsvReader(objReader, CultureInfo.CurrentCulture)
    objRecords = objCsvReader.GetRecords(Of Kunde)()
```

Die in dieser Liste enthaltenen **Kunde**-Objekte schreiben wir dann in eine **List(Of Kunde)**-Auflistung und zeigen diese im **DataGrid**-Element an:

EDM-Daten in CSV exportieren

Im Artikel »CSV-Datei in Klassen importieren« haben wir bereits gezeigt, wie Sie die Daten aus CSV-Dateien in .NET-Anwendungen importieren und diese auf die enthaltenen Klassen aufteilen können. Dazu haben wir das NuGet-Paket CsvHelper verwendet. Im vorliegenden Artikel schauen wir uns den umgekehrten Weg an: Wie können wir die Daten aus den Objekten eines Entity Data Models in eine CSV-Datei exportieren? Dazu setzen wir direkt an der Beispieldatenbank für den oben genannten Artikel an und speichern die darauf importierten Daten direkt wieder in eine neue CSV-Datei.

CsvHelper hinzufügen

Das erwähnte NuGet-Paket **CsvHelper** fügen wir über den Nuget-Paket-Manager hinzu, den Sie über den Menüeintrag **Projekt|NuGet-Pakete verwalten...** öffnen. Hier wechseln Sie zum Bereich **Durchsuchen** und geben **csvhelper** in das Suchfeld ein. Es erscheint der Eintrag **CsvHelper**, den wir auswählen und mit einem Klick auf die Schaltfläche **Installieren** zum Projekt hinzufügen.

Beispielanwendung

Die Benutzeroberfläche der Beispielanwendung bauen wir auf dem Beispiel aus dem oben genannten Artikel auf. Dort finden wir bereits Steuerelemente, mit denen Sie eine CSV-Datei zum Importieren ihrer Inhalte auswählen

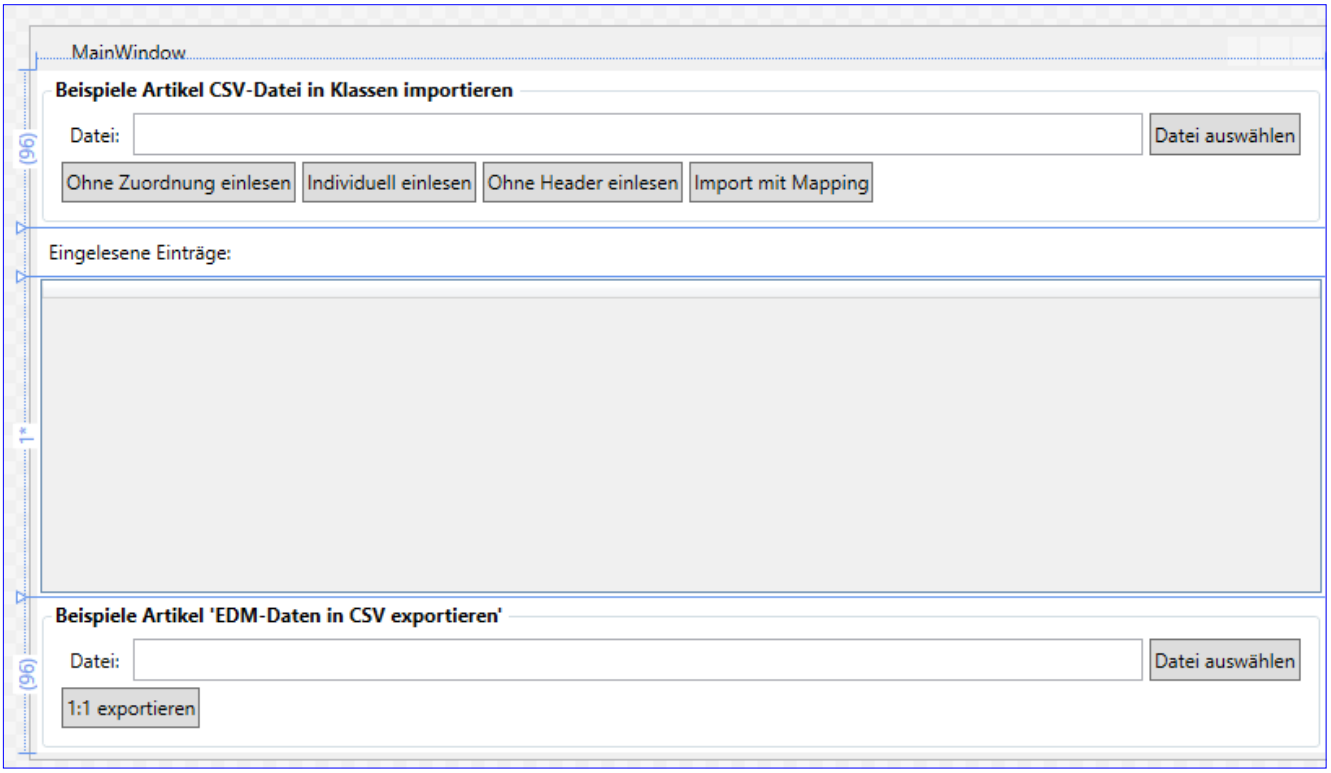


Bild 1: Entwurf des Hauptfensters der Anwendung

können. Die Inhalte werden dann in einem **DataGrid**-Steuerelement angezeigt. Hier fügen wir einen weiteren Bereich an, der die Auswahl eines Dateinamens zum Speichern der Inhalte des **DataGrid**-Elements in einer weiteren CSV-Datei erlaubt. Dieses sieht wie in Bild 1 aus.

Hier ist der XAML-Code der hier enthaltenen Steuerelemente. Wir verwenden das DataGrid **dgrKunden**, um die zu exportierenden Daten zu ermitteln. Das **TextBox**-Steuerelement **txtExportfile** nimmt den Pfad der zu erstellenden Datei auf.

Diesen ermittelt der Benutzer durch einen Klick auf die Schaltfläche **btnChooseSaveFile**. Und der eigentliche Export (die erste Version, es folgen weitere), erfolgt über die Schaltfläche **btnExport11**:

```
    ...
    <Label Grid.Row="1">Eingelesene Einträge:</Label>
    <DataGrid x:Name="dgrKunden" Grid.Row="2"></DataGrid>
    <GroupBox Header="Beispiele Artikel 'EDM-Daten in CSV exportieren'" FontWeight="Bold" Grid.Row="3">
        <ContentControl FontWeight="Normal" HorizontalAlignment="Stretch">
            <StackPanel Orientation="Vertical" HorizontalAlignment="Stretch">
                <Grid>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto"></ColumnDefinition>
                        <ColumnDefinition Width="*"></ColumnDefinition>
                        <ColumnDefinition Width="Auto"></ColumnDefinition>
                    </Grid.ColumnDefinitions>
                    <Label Grid.Column="0">Datei:</Label>
                    <TextBox x:Name="txtExportfile" Grid.Column="1"></TextBox>
                    <Button x:Name="btnChooseSaveFile" Grid.Column="2"
                        Click="btnChooseSaveFile_Click" HorizontalAlignment="Right">Datei auswählen</Button>
                </Grid>
                <StackPanel Orientation="Horizontal">
                    <Button x:Name="btnExport11" Click="btnExport11_Click">1:1 exportieren</Button>
                </StackPanel>
            </StackPanel>
        </ContentControl>
    </GroupBox>
</Grid>
```

Datei zum Speichern festlegen

Die Schaltfläche **btnChooseSaveFile** löst die folgende Methode aus. Diese erfordert das Vorhanden des folgenden Namespaces:

```
Imports Microsoft.Win32
```

Die Methode ermittelt das Startverzeichnis, das gleich beim Öffnen des Dialogs angezeigt werden soll, aus dem Verzeichnis aus dem Textfeld **txtImportfile**. Ist dieses leer oder enthält es keine vorhandene Datei, wird stattdessen das Verzeichnis der **.exe**-Datei dieser Anwendung genutzt.

Danach stellt die Methode die Dateiendung auf **.csv** ein und öffnet den **Speichern unter**-Dialog. Enthält dieser beim Schließen eine Zieldatei, wird diese samt Pfad in das Textfeld **txtExportFile** geschrieben:

```
Private Sub btnChooseSaveFile_Click(sender As Object, e As RoutedEventArgs)
    Dim objSaveFileDialog As New SaveFileDialog
    If File.Exists(txtImportfile.Text) Then
        objSaveFileDialog.InitialDirectory = Path.GetDirectoryName(txtImportfile.Text)
    Else
        objSaveFileDialog.InitialDirectory = AppDomain.CurrentDomain.BaseDirectory
    End If
    objSaveFileDialog.DefaultExt = ".csv"
    If (objSaveFileDialog.ShowDialog = True) Then
        txtExportfile.Text = objSaveFileDialog.FileName
    End If
End Sub
```

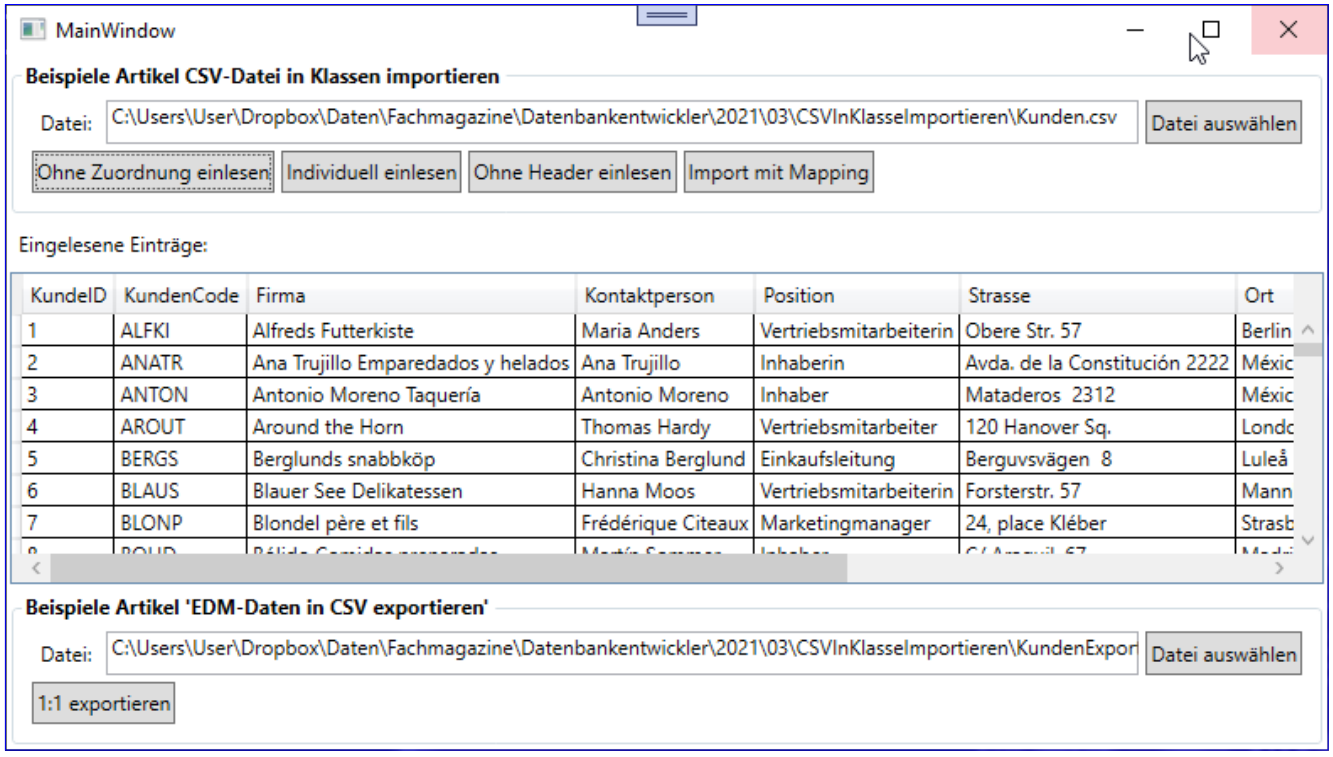


Bild 2: Zu exportierende Daten im **DataGrid**-Steuerelement

Einfacher Export in eine CSV-Datei

Wir gehen an dieser Stelle davon aus, dass das **DataGrid**-Steuerelement wie in Bild 2 einige Datensätze enthält. In der durch die Schaltfläche **btnExport11** aufgerufenen Methode überprüfen wir diese Voraussetzung. Falls Datensätze enthalten sind, liest sie diese in die **List(Of Kunde)**-Variable **objKunden** ein. Dann erstellt die Prozedur ein neues Objekt des Typs **StreamWriter** und übergibt den Namen der zu erzeugenden Datei als Parameter des Konstruktors.

Anschließend landet der **StreamWriter** als erster Parameter im Konstruktor eines neuen Objekts des Typs **CsvWriter**. Mit **WriteRecords** schreiben wir die Elemente aus **objKunden** in den **CsvWriter** und mit **Flush** sorgen wir dafür, dass der Inhalt aus dem Writer in der Datei landet:

```
Private Sub btnExport11_Click(sender As Object, e As RoutedEventArgs)
    Dim objKunden As List(Of Kunde)
    Dim objWriter As StreamWriter
    Dim objCsvWriter As CsvWriter
    Dim strExportFile As String
    strExportFile = txtExportfile.Text
    If Not dgrKunden.Items.Count = 0 Then
        objKunden = dgrKunden.ItemsSource
        objWriter = New StreamWriter(strExportFile)
        objCsvWriter = New CsvWriter(objWriter, CultureInfo.CurrentCulture)
        With objCsvWriter
            .WriteRecords(objKunden)
```

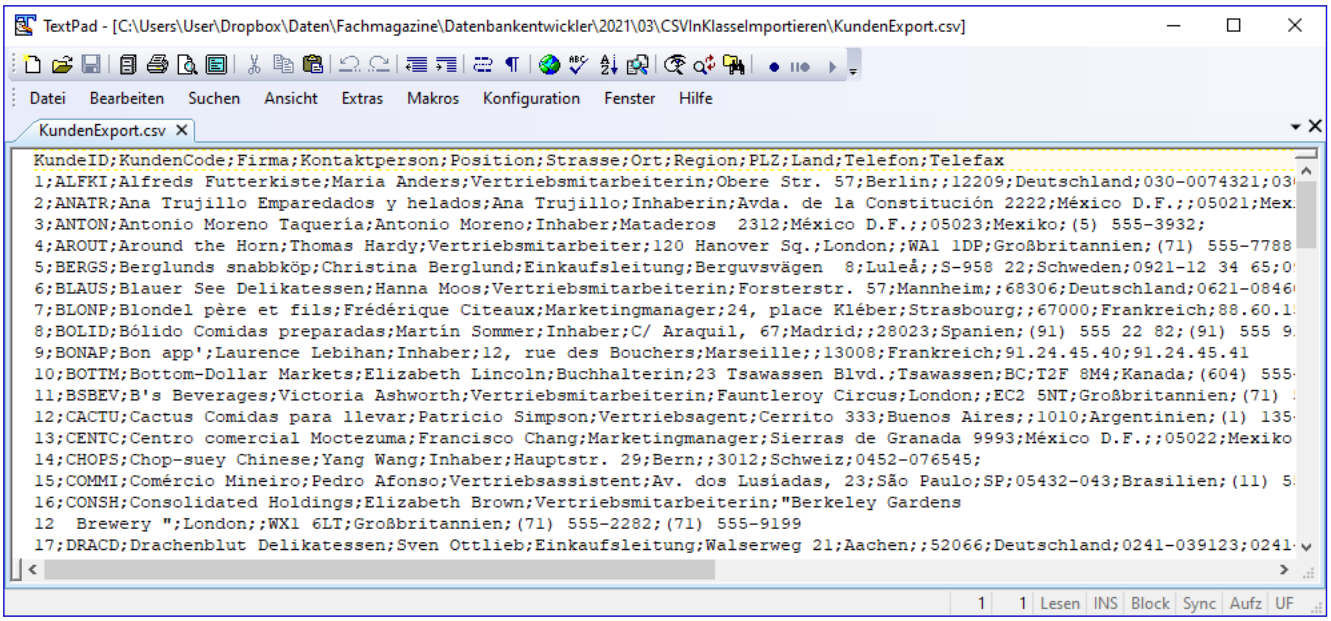


Bild 3: Ergebnis des ersten Exports