AUSGABE 04/2021

DATENBANK ENTWICKLER

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

TOP-THEMEN:

WPF-BASICS	Von Page zu Page	SEITE 3
	Daten in Listen filtern mit ListCollectionView	SEITE 19
STEUERELEMENTE	Das WPF-Steuerelement CheckBox	SEITE 33
VB-BASICS	Typumwandlung unter VB.NET	SEITE 60

André Minhorst Verlag

Von Page zu Page

Wenn Sie Pages zur Darstellung verschiedener Daten wie Übersichten, Detailansichten oder auch allgemeine Pages nutzen, die in einem Frame-Element angezeigt werden, wollen Sie gegebenenfalls auch einmal Daten von einer Seite zur nächsten schicken. Ein Anwendungsfall wäre die Auswahl eines Artikels in einer Übersichtsseite, der dann in einer Detailseite angezeigt werden soll. Oder das Öffnen einer Übersichtsseite zur Auswahl eines Eintrags, der dann in die aufrufende Seite übernommen werden soll. Dieser Artikel zeigt, wie Sie beim Navigieren im Frame-Element Informationen zwischen Page-Elementen hin- und herschicken können.

Grundgerüst für die Beispiele

Für die Beispiele haben wir im Fenster **MainWindow.xaml** ein Grid mit zwei Zeilen definiert. Es enthält in der ersten Zeile ein **StackPanel**-Element, mit dem wir die einzelnen Beispiele aufrufen können. Die zweite Zeile enthält ein **Frame**-Element namens **fra**, dass die Seiten für die einzelnen Beispiele anzeigen soll:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal">
        <Button x:Name="btnVonANachB" Click="btnVonANachB_Click">Beispiel Von A nach B per Konstruktor
            starten</Button>
    </StackPanel>
    <Frame x:Name="fra" Grid.Row="1"></Frame>
</Grid>
```

Von A nach B per Konstruktor-Parameter

Wir wollen nun zwei Seiten definieren namens **VonANachB_Konstruktor_SeiteA.xaml** und **VonANachB_Konstruktor_SeiteB.xaml**. Die erste enthält ein Textfeld, in das Sie einen Text ein geben können. Mit einer Schaltfläche wollen wir die zweite Seite aufrufen und dieser den eingegebenen Text übergeben, sodass dieser dort in einem weiteren Textfeld angezeigt werden kann.

Die erste Seite **VonANachB_Konstruktor_SeiteA.xaml** definieren wir wie folgt:

```
<Grid>
    ...
    <Label FontWeight="Bold">Von A nach B per Konstruktor - Seite A</Label>
    <StackPanel Orientation="Horizontal" Grid.Row="1">
```

```
        <Label>Zu übergebender Wert:</Label>
        <TextBox x:Name="txtWert" Width="200"></TextBox>
    </StackPanel>
    <StackPanel Grid.Row="2">
        <Button x:Name="btnVonANachB" HorizontalAlignment="Left" Click="btnVonANachB_Click">
            Wert nach B senden</Button>
    </StackPanel>
</Grid>
```

Die erste Seite sieht in der Entwurfsansicht wie in Bild 1 aus.

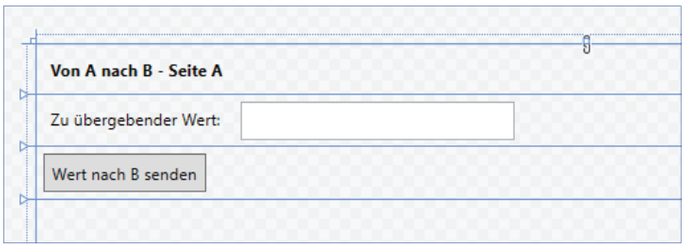


Bild 1: Aufbau der ersten Seite

Die zweite Seite ist noch einfacher aufgebaut – sie enthält lediglich ein **Label**- und ein **TextBox**-Steuerelement:

```
<Grid>
    ...
    <Label FontWeight="Bold">Von A nach B per Konstruktor - Seite B</Label>
    <TextBox x:Name="txtParameter" Grid.Row="1" HorizontalAlignment="Left" Width="200"></TextBox>
</Grid>
```

Erste Seite aufrufen

Um die erste Seite im **Frame**-Element anzuzeigen, betätigen wir die Schaltfläche **btnVonANachB_Konstruktor**. Diese erstellt die Seite **VonANachB_Konstruktor_SeiteA** neu und referenziert diese Instanz mit der Variablen **objVonANachB_Konstruktor_SeiteA**. Damit diese im **Frame**-Element erscheint, rufen wir dessen **Navigate**-Methode mit dem Objektnamen als Parameter auf:

```
Class MainWindow
    Private Sub btnVonANachB_Konstruktor_Click(sender As Object, e As RoutedEventArgs)
        Dim pgeVonANachB_Navigate_SeiteA As New VonANachB_Konstruktor_SeiteA
        fra.Navigate(pgeVonANachB_Navigate_SeiteA)
    End Sub
End Class
```

Konstruktor der zweiten Seite mit Parameter ausstatten

Wir wollen den Wert aus dem Textfeld **txtWert** der ersten Seite zur zweiten Seite schicken. Für die Übergabe wollen wir einen Parameter der Konstruktor-Methode der zweiten Seite nutzen. Die Konstruktormethode, die immer **New** heißt, erstellen Sie am einfachsten, indem Sie mit der rechten Maustaste in die Klasse **VonANach_Konstruktor_B** im Codefenster der Code behind-Klasse **VonANachB_Konstruktor_SeiteB.xaml.vb** klicken und dort den Eintrag **Schnellaktionen und Refactorings...** auswählen.

Nach diesem Befehl erscheint eine Liste der möglichen Schnellaktionen. Hier wählen wir den Eintrag **Konstruktor generieren...** aus (siehe Bild 2).

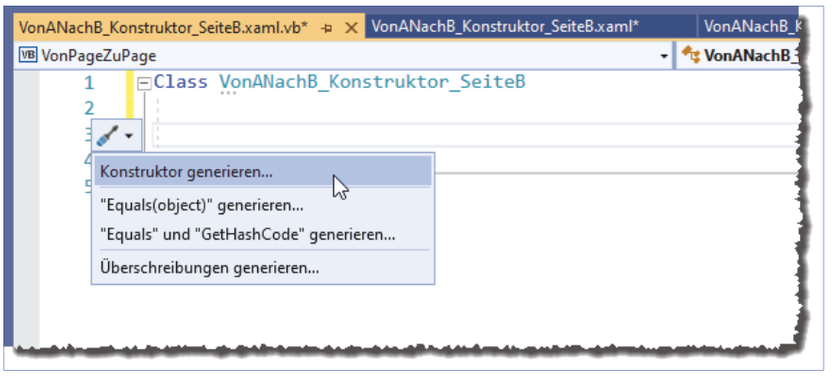


Bild 2: Konstruktor hinzufügen

Danach erscheint noch ein weiterer Dialog namens **Member auswählen**. Hier wählen Sie den Eintrag **_contentLoaded** ab (siehe Bild 3) und betätigen dann die **OK**-Schaltfläche.

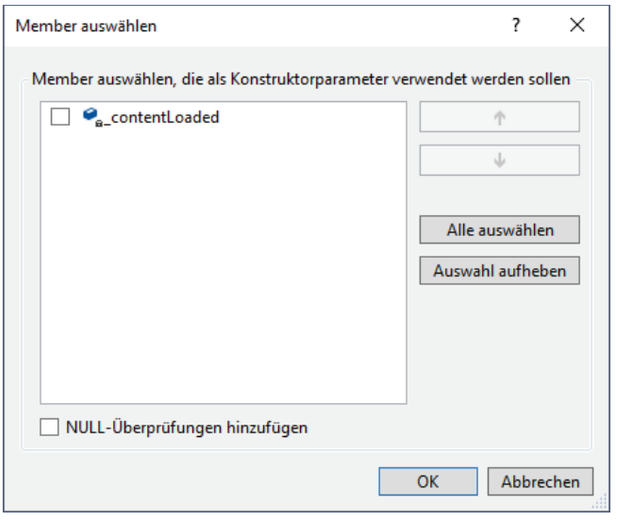


Bild 3: Dialog mit Eigenschaften des Konstruktors

Die Klasse mit der so erstellten Konstruktormethode sieht zunächst wie folgt aus:

```
Class VonANachB_Konstruktor_SeiteB
    Public Sub New()
    End Sub
End Class
```

Diese Methode wollen wir allerdings dazu nutzen, beim Aufruf einen Parameter zu übergeben. Deshalb erweitern wir die erste Zeile um den Parameter **strParameter** mit dem Datentyp **String**. Außerdem fügen wir der Konstruktormethode die Anweisung **InitializeComponent** hinzu, die beim Fehlen einer Konstruktormethode normalerweise implizit ausgeführt wird. Lassen Sie diese weg, wird die XAML-Definition nicht umgesetzt!

Schließlich schreiben wir den Inhalt des Parameters **strParameter** in das Textfeld **txtParameter**:

```
Public Sub New(strParameter As String)
    InitializeComponent()
    txtParameter.Text = strParameter
End Sub
```

Nun fehlt noch die Übergabe des Parameters. Diese erfolgt, wenn der Benutzer die Schaltfläche **btnVonANachB_Konstruktor** in der Klasse **VonANachB_Konstruktor_SeiteA** betätigt.

Die dadurch ausgelöste Methode definieren wir wie folgt:

```
Class VonANachB_Konstruktor_SeiteA
    Private Sub btnVonANachB_Konstruktor_Click(sender As Object, e As RoutedEventArgs)
        Dim pgeVonANachB_Konstruktor_SeiteB As VonANachB_Konstruktor_SeiteB
        Dim strParameter As String
        strParameter = txtWert.Text
        pgeVonANachB_Konstruktor_SeiteB = New VonANachB_Konstruktor_SeiteB(strParameter)
        NavigationService.Navigate(pgeVonANachB_Konstruktor_SeiteB)
    End Sub
End Class
```

Hier deklarieren wir die Zielseite **VonANachB_Konstruktor_SeiteB** mit der Variablen **pgeVonANachB_Konstruktor_SeiteB**. Außerdem deklarieren wir eine Variable namens **strParameter** für den Inhalt des Textfeldes **txtWert**, dass wir direkt in der folgenden Anweisung füllen. Danach erstellen wir eine neue Instanz von **VonANachB_Konstruktor_SeiteB** und übergeben der Konstruktormethode dabei den Wert von **strParameter**. Schließlich navigieren wir mit der **Navigate**-Methode des **NavigationService**-Objekts zu der mit **pgeVonANachB_Konstruktor_SeiteB** referenzierten Seite.

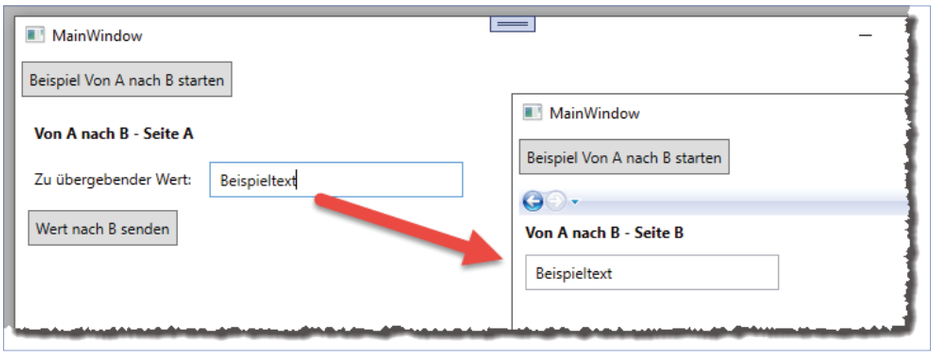


Bild 4: Übergeben eines Textes

Das Ergebnis sehen Sie in Bild 4. Den übergebenen Wert können Sie für beliebige Einsatzzwecke nutzen – beispielsweise für die Übergabe des Primärschlüsselwertes eines in einer Übersicht auf der ersten Seite ausgewählten Eintrags, dessen Detaildaten dann auf der zweiten Seite angezeigt werden sollen.

Von A nach B per Application.Properties

Im zweiten Beispiel nutzen wir keine explizite Übergabe des Wertes von Seite A nach Seite B, sondern wir speichern den zu übergebenden Wert in einer Eigenschaft der aktuellen Anwendung. Der Aufbau ist ähnlich wie im ersten Beispiel, allerdings heißen die beiden Seiten nun **VonANachB_Property_SeiteA** und **VonANachB_Property_SeiteB**. Auf Seite A befindet sich wieder ein Textfeld namens **txtWert**. Außerdem haben wir dort eine Schaltfläche namens **btnVonANachBProperty** angelegt, für die wir die folgende Methode hinterlegt haben:

```
Class VonANachB_Property_SeiteA
    Private Sub btnVonANachB_Property_Click(sender As Object, e As RoutedEventArgs)
        Dim pgeVonANachB_Property_SeiteB As VonANachB_Property_SeiteB
        My.Application.Properties("Parameter") = txtWert.Text
```

```
        pgeVonANachB_Property_SeiteB As New VonANachB_Property_SeiteB()
        NavigationService.Navigate(pgeVonANachB_Property_SeiteB)
    End Sub
End Class
```

Wir erstellen also wieder die Zielseite, diesmal allerdings mit einer Konstruktormethode ohne Parameter. Wir wollen den zu übergebenen Wert ja in einer Anwendungseigenschaft zwischenspeichern und dann abrufen. Eine Anwendungseigenschaft füllen wir, indem wir für die Auflistung **My.Application.Properties** einfach den Namen der Eigenschaft angeben und einen Wert zuweisen:

```
My.Application.Properties("Parameter") = txtWert.Text
```

Ist die Eigenschaft zu diesem Zeitpunkt noch nicht vorhanden, wird diese erstellt. Danach erstellen wir dann die neue Instanz von Seite B und steuern diese mit der **Navigate**-Methode an.

Damit der Wert aus den Anwendungseigenschaften auf dieser Seite angezeigt wird, benötigen wir wieder die Konstruktor-Methode. Diese füllen wir diesmal wie folgt:

```
Class VonANachB_Property_SeiteB
    Public Sub New()
        InitializeComponent()
        txtParameter.Text = My.Application.Properties("Parameter")
    End Sub
End Class
```

An dieser Stelle ist wichtig, dass Sie die richtige Reihenfolge einhalten. Wir müssen zuerst die Anwendungseigenschaft **Parameter** füllen und dann die neue Instanz von Seite B erstellen.

Wenn Sie zuerst die neue Instanz von Seite B erstellen, liest die Konstruktormethode den Wert von **My.Application.Properties("Parameter")** ein, bevor wir die Eigenschaft im aufrufenden Fenster gefüllt haben und das Textfeld von Seite B bleibt leer.

Vor- und Nachteile der Methoden

Der Vorteil der direkten Übergabe per Konstruktorparameter ist, dass wir erstens Typsicherheit haben und zweitens die Werte der Parameter nur dort bekannt sind, wo sie bekannt sein sollen. Bei der Verwendung von Anwendungseigenschaften können wir von überall innerhalb der Anwendung auf die Eigenschaft zugreifen – und diese nicht nur lesen, sondern diese auch verändern.

Außerdem sind die Anwendungseigenschaften nicht typsicher. Die Anwendungseigenschaften haben Vorteile, wenn Sie die enthaltenen Werte tatsächlich anwendungsweit verfügbar machen wollen – beispielsweise um einen aktuellen Kunden bekannt zu machen, zu dem Daten in verschiedenen Fenstern angezeigt werden sollen.

Fensterposition beim Öffnen einstellen

Nachdem ich mittlerweile mit zwei Bildschirmen arbeite (einer für die Hauptanwendungen, einer für andere Informationen wie Termine et cetera) und der Bildschirm für die Hauptanwendungen der zweite Bildschirm ist, nervt es mich etwas, dass Anwendungen beim Debuggen immer auf dem sekundären Bildschirm angezeigt werden und ich diese immer erst auf den Hauptbildschirm ziehen muss. Also schauen wir uns in diesem Artikel an, welche Möglichkeiten .NET bietet, festzulegen, auf welchem Bildschirm eine Anwendung geöffnet werden soll.

Ausgangssituation

Die aktuelle Situation ist also, dass ich zwei Bildschirme habe – der linke Bildschirm zeigt unwichtige Anwendungen an, der rechte (oder auch zentrale) Bildschirm dient zum Arbeiten. Wenn ich nun Visual Studio auf dem zentralen Bildschirm geöffnet habe und eine Anwendung zum Debuggen starte, erscheint diese immer auf dem linken, sekundären Bildschirm. Wenn ich die beim Debuggen erstellte **.exe**-Datei direkt aufrufe, wird diese wie gewünscht im Hauptbildschirm angezeigt. Wie kann ich nun steuern, welcher Bildschirm die Anwendung beim Debuggen anzeigt?

Lösung aus der Forms-Bibliothek

Die notwendigen Elemente für den Zugriff auf die Bildschirme finden sich in der Bibliothek **System.Windows.Forms**, die wir zunächst über einen entsprechenden Verweis zu unserem WPF-Projekt hinzufügen. Das erledigen wir über den Dialog, der nach dem Aufruf des Menüeintrags **Projekt|Verweise hinzufügen...** erscheint. Hier sehen Sie im Bereich **Assemblys** nach einer schnellen Suche den Eintrag **System.Windows.Forms**, den Sie per Klick in das Kontrollkästchen zum Projekt hinzufügen (siehe Bild 1). Anschließend können wir per Code auf die Elemente dieser Bibliothek zugreifen.

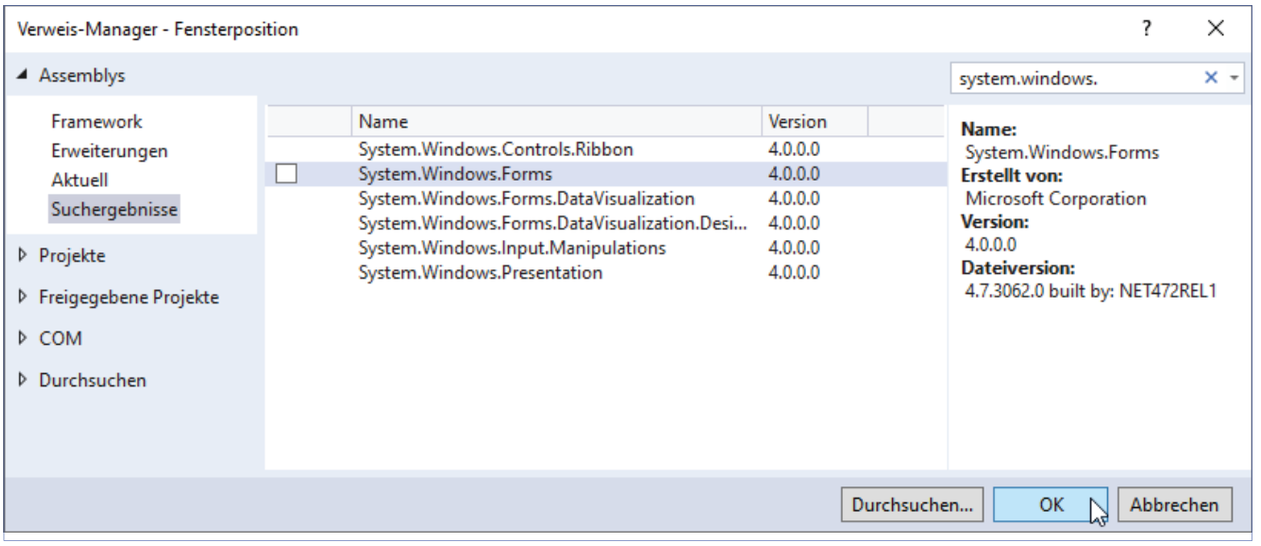


Bild 1: Hinzufügen eines Verweises auf die Bibliothek **System.Windows.Forms**

Die Screen-Klasse

Mit der **Screen**-Klasse eröffnen wir uns die Möglichkeit, auf die Bildschirme des Systems zuzugreifen. Die **Screen**-Klasse liefert uns die folgenden interessanten Auflistungen und Eigenschaften:

- **AllScreens**: Zugriff auf alle **Screen**-Elemente
- **BitsPerPixel**: Anzahl der Bits im Arbeitsspeicher, die einem Pixel von Daten zugeordnet sind
- **Bounds**: Ermittelt die Maße des Bildschirms. Dies liefert ein Objekt des Typs **Rectangle**, das wiederum die vier Eigenschaften **Left**, **Top**, **Width** und **Height** aufweist. Bei mehreren Bildschirmen werden die Werte bezogen auf das Gesamtsystem betrachtet. Wenn der linke Bildschirm beispielsweise eine Breite (**Width**) von 1920 Pixeln hat, dann ist der Wert für die Eigenschaft **Left** des rechten Bildschirms ebenfalls gleich 1920 Pixel.
- **DeviceName**: Liefert den Namen des Bildschirms im System.
- **Primary**: Gibt an, ob der Bildschirm der primäre Bildschirm ist.
- **PrimaryScreen**: Liefert einen Verweis auf den primären Bildschirm.
- **WorkingArea**: Ermittelt die Maße des Arbeitsbereichs des Bildschirms. Dieser unterscheidet sich von den mit Bounds ermittelten Maßen, in dem es nur den Bereich ohne Elemente wie die Taskleiste oder angedockte Fenster und Menüs ermittelt – also den tatsächlich verfügbaren Arbeitsbereich. Die Eigenschaft **WorkingArea** liefert ein Objekt des Typs Rectangle, das wiederum die vier Eigenschaften **Left**, **Top**, **Width** und **Height** aufweist. Auch hier gilt: Bei mehreren Bildschirmen werden die Werte bezogen auf das Gesamtsystem betrachtet. Wenn der linke Bildschirm beispielsweise eine Breite (**Width**) von 1920 Pixeln hat, dann ist der Wert für die Eigenschaft **Left** des rechten Bildschirms ebenfalls gleich 1920 Pixel.

Hier ist nun zu beachten, dass **Screen** selbst bereits ein Element des Typs **Screen** liefert, mit dem Sie einige Eigenschaften des Bildschirms wie **BitsPerPixel**, **Bounds**, **DeviceName**, **Primary** und **WorkingArea** auslesen können. Gleichzeitig können Sie aber über die Eigenschaft **PrimaryScreen** einen Verweis auf das **Screen**-Objekt im System ermitteln, das als primärer Bildschirm angegeben ist.

Für den Zugriff auf die **Screen**-Klasse benötigen wir auch noch die folgende **Imports**-Anweisung:

```
Imports System.Windows.Forms
```

Beispiel: Ausgabe der Bildschirmdaten per Meldungsfenster

Die folgende Methode, die wir in der Konstruktor-Methode des Fensters **MainWindow** der Beispielanwendung aufrufen, definiert eine Variable namens **objScreen** für ein Objekt des Typs **Screen**. In einer **For Each**-Schleife durchlaufen wir alle Elemente der Auflistung **AllScreens** des **Screen**-Objekts. Innerhalb der Schleife setzen wir eine Meldung zusammen, die alle relevanten Informationen über den Bildschirm enthält und zeigen diese dann

in einer Meldung an. Dabei greifen wir auf die einzelnen Werte der Eigenschaften **Bounds** und **WorkingArea** über deren Untereigenschaften **Left**, **Top**, **Width** und **Height** zu. Damit das möglich ist, benötigen Sie noch einen Verweis auf die Bibliothek **System.Drawing**:

```
Public Sub New()
    InitializeComponent()
    Dim objScreen As Screen
    For Each objScreen In Screen.AllScreens
        MessageBox.Show("DeviceName: " & objScreen.DeviceName & vbCrLf _
            & "Primary: " & objScreen.Primary & vbCrLf _
            & "Bounds: " & objScreen.Bounds.Left & ", " & objScreen.Bounds.Top & ", " _
            & objScreen.Bounds.Width & ", " & objScreen.Bounds.Height & vbCrLf _
            & "WorkingArea: " & objScreen.WorkingArea.Left & ", " & objScreen.WorkingArea.Top & ", " _
            & objScreen.WorkingArea.Width & ", " & objScreen.WorkingArea.Height & vbCrLf _
            & "BitsPerPixel: " & objScreen.BitsPerPixel)
    Next
End Sub
```

Das Ergebnis ist je eine Meldung für jeden Bildschirm, der am aktuellen Rechner angeschlossen ist (siehe Bild 2).

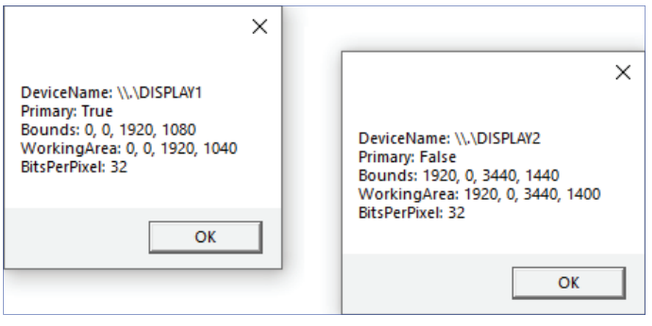


Bild 2: Ausgabe der Daten der beiden Bildschirme des Beispielsystems

Daten der Bildschirme im ListView ausgeben

Weil wir die Daten auch übersichtlich in einem ListView anzeigen wollen, fügen wir dem Fenster MainWindow.xaml die folgende Definition für ein ListView-Steuerelement hinzu:

```
<ListView x:Name="lvwScreens">
    <ListView.View>
        <GridView>
            <GridViewColumn Header="DeviceName" DisplayMemberBinding="{Binding DeviceName}" />
            <GridViewColumn Header="Primary" DisplayMemberBinding="{Binding Primary}" />
            <GridViewColumn Header="Bounds" DisplayMemberBinding="{Binding Bounds}" />
            <GridViewColumn Header="WorkingArea" DisplayMemberBinding="{Binding WorkingArea}" />
            <GridViewColumn Header="PrimaryScreen" DisplayMemberBinding="{Binding PrimaryScreen}" />
            <GridViewColumn Header="BitsPerPixel" DisplayMemberBinding="{Binding BitsPerPixel}" />
        </GridView>
    </ListView.View>
</ListView>
```

Daten in Listen filtern mit ListCollectionView

Daten aus einer SQL Server-Datenbank oder anderen Datenquellen können wir über das Entity Framework als Objekte zugänglich machen und auch als solche in Listen-Steuerelementen anzeigen. Aber wie sieht es mit weiteren Funktionen aus – etwa zum Filtern, Sortieren oder Gruppieren? Dazu benötigen wir einige spezielle Techniken, von denen wir die zum Filtern von Daten nach bestimmten Kriterien im vorliegenden Artikel vorstellen. Dabei nutzen wir Filter für Felder mit verschiedenen Datentypen und in verschiedenen Kombinationen.

Beispieldaten

Als Beispielmaterial wollen wir diesmal nicht auf ein extra anzulegendes Entity Data Model zugreifen, sondern einfach eine Liste von **Kunde**-Elementen erstellen, die ihre Daten aus einer Funktion bezieht statt aus einer Datenbank. Damit wir ein wenig Spielmaterial haben, soll die **Kunde**-Klasse String-, Zahlen-, Datums- und Boolean-Werte enthalten. Die Klasse **Kunde** sieht daher wie folgt aus:

```
Partial Public Class Kunde
    Public Property ID As System.Int32
    Public Property Firma As System.String
    Public Property Vorname As System.String
    Public Property Nachname As System.String
    Public Property Anrede As System.String
    Public Property Strasse As System.String
    Public Property PLZ As System.String
    Public Property Ort As System.String
    Public Property Land As System.String
    Public Property EMail As System.String
    Public Property Geburtsdatum As System.DateTime
    Public Property Alter As System.Int32
    Public Property Newsletter As System.Boolean
End Class
```

Um ein **ListView**-Steuerelement zu füllen, benötigen wir Beispieldaten, die wir mit einer Funktion namens **GetKunden** bereitstellen. Diese fügen wir im Code behind-Modul des Fensters **MainWindow** ein:

```
Private Function GetKunden() As ObservableCollection(Of Kunde)
    Dim Kunden As New ObservableCollection(Of Kunde)
    Kunden.Add(New Kunde() With {.ID = 1, .Firma = "Krahn GbR", .Vorname = "Adi", _
       .Nachname = "Stratmann", .Anrede = "Herr", .Strasse = "Kremser Straße 54", .PLZ = "10589", _
       .Ort = "Berlin", .Land = "Deutschland", .EMail = "adi@stratmann.de", _
       .Geburtsdatum = "20.02.1939", .Alter = 82, .Newsletter = -1})
```

```
    Kunden.Add(New Kunde() With {.ID = 2, .Firma = "Göllner AG", .Vorname = "Heidi", _
        .Nachname = "Eich", .Anrede = "Frau", .Strasse = "Moosstraße 30", .PLZ = "42289", _
        .Ort = "Wuppertal", .Land = "Deutschland", .EMail = "heidi@eich.de", _
        .Geburtsdatum = "30.11.1971", .Alter = 50, .Newsletter = 0})
    Kunden.Add(New Kunde() With {.ID = 3, .Firma = "Peukert GmbH & Co. KG", .Vorname = "Wernfried", _
        .Nachname = "Birk", .Anrede = "Herr", .Strasse = "Wiener Straße 78", .PLZ = "22297", _
        .Ort = "Hamburg", .Land = "Deutschland", .EMail = "wernfried@birk.de", _
        .Geburtsdatum = "23.01.1931", .Alter = 90, .Newsletter = -1})
    ...
    Kunden.Add(New Kunde() With {.ID = 100, .Firma = "Krahn KG", .Vorname = "Rosemaria", _
        .Nachname = "Grill", .Anrede = "Frau", .Strasse = "Kalvarienbergstraße 35", .PLZ = "09131", _
        .Ort = "Chemnitz", .Land = "Deutschland", .EMail = "rosemaria@grill.de", _
        .Geburtsdatum = "02.10.2009", .Alter = 12, .Newsletter = -1})
    Return Kunden
End Function
```

In der Klasse **MainWindow** hinterlegen wir außerdem noch eine private Variable für die Auflistung der Kunden:

```
Private _Kunden As ObservableCollection(Of Kunde)
```

Diese machen wir über die folgende öffentlich deklarierte Property verfügbar:

```
Public Property Kunden As ObservableCollection(Of Kunde)
    Get
        Return _Kunden
    End Get
    Set(value As ObservableCollection(Of Kunde))
        _Kunden = value
    End Set
End Property
```

Schließlich sorgen wir in der Konstruktor-Methode der Klasse, die beim Anzeigen automatisch ausgeführt wird, dass die XAML-Definition umgesetzt, der **DataContext** auf die aktuelle Code behind-Klasse gesetzt und die Eigenschaft **_Kunden** mit dem Ergebnis der Funktion **GetKunden** gefüllt wird:

```
Public Sub New()
    InitializeComponent()
    DataContext = Me
    _Kunden = GetKunden()
End Sub
```

Fenster mit ListView-Steuerelement

Danach definieren wir den XAML-Code für das Fenster zur Anzeige der Daten im **ListView**-Steuerelement:

```
<Window x:Class="MainWindow" ...
        Title="MainWindow" Height="450" Width="800" WindowStartupLocation="CenterScreen">
    ...
    <Grid>
        ...
        <ListView x:Name="lvw" ItemsSource="{Binding Kunden}">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}" />
                    <GridViewColumn Header="Firma" DisplayMemberBinding="{Binding Firma}" />
                    <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding Vorname}" />
                    <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding Nachname}" />
                    <GridViewColumn Header="Anrede" DisplayMemberBinding="{Binding Anrede}" />
                    <GridViewColumn Header="Strasse" DisplayMemberBinding="{Binding Strasse}" />
                    <GridViewColumn Header="PLZ" DisplayMemberBinding="{Binding PLZ}" />
                    <GridViewColumn Header="Ort" DisplayMemberBinding="{Binding Ort}" />
                    <GridViewColumn Header="Land" DisplayMemberBinding="{Binding Land}" />
                    <GridViewColumn Header="EMail" DisplayMemberBinding="{Binding EMail}" />
                    <GridViewColumn Header="Geburtsdatum"
                        DisplayMemberBinding="{Binding Path=Geburtsdatum, StringFormat='dd.MM.yyyy'}" />
                    <GridViewColumn Header="Alter" DisplayMemberBinding="{Binding Alter}" />
                    <GridViewColumn Header="Newsletter">
                        <GridViewColumn.CellTemplate>
                            <DataTemplate>
                                <CheckBox IsChecked="{Binding Newsletter}" />
                            </DataTemplate>
                        </GridViewColumn.CellTemplate>
                    </GridViewColumn>
                </GridView>
            </ListView.View>
        </ListView>
        <StackPanel Orientation="Horizontal" Grid.Row="1">
            <Button>Button</Button>
        </StackPanel>
    </Grid>
</Window>
```

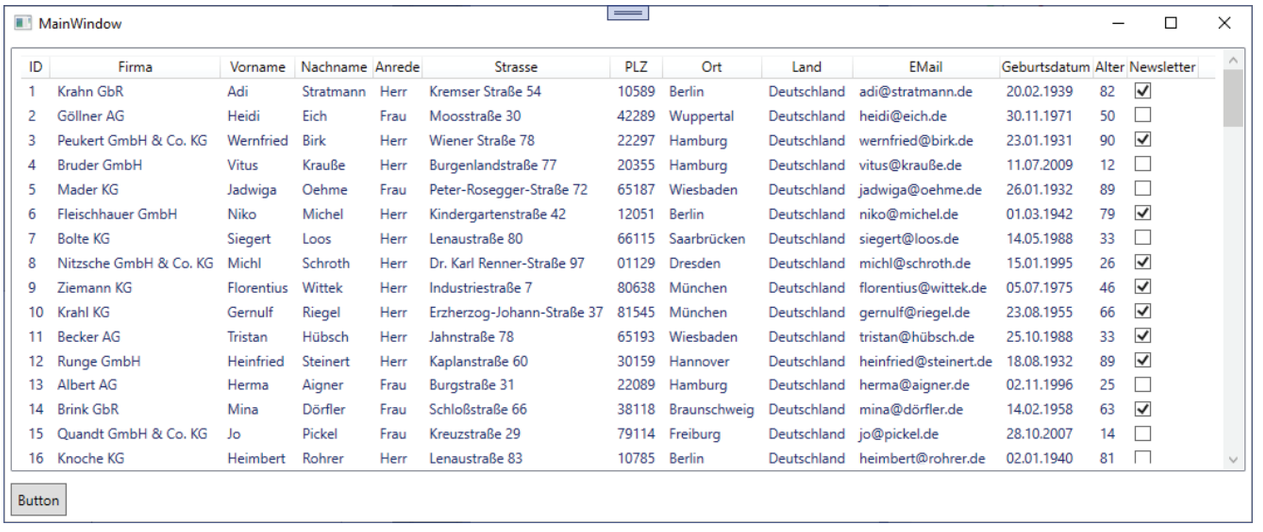
Das **ListView**-Steuerelement stellen wir mit dem Attribut **ItemsSource** so ein, dass es den Wert der Eigenschaft **Kunden** als Datenquelle verwendet. Diese wird in der Code behind-Klasse wie oben beschrieben definiert. Die einzelnen Spalten im **ListView**-Steuerelement definieren wir als **GridViewColumn**-Elemente. Dabei geben wir mit dem Attribut **Header** die Spaltenüberschrift an und mit **DisplayMemberBinding** das Feld, aus der die Spalte ihre Daten beziehen soll. Hier verwenden wir meist den Wert {**Binding [Feldname]**}, wobei **[Feldname]** durch den jeweiligen Feldnamen ersetzt wird. Bei zwei Feldern gibt es Abweichungen von dieser Vorgehensweise: Beim Feld **Geburtsdatum** verwenden wir den Ausdruck {**Binding Path=Geburtsdatum, StringFormat='dd.MM.yyyy'**}, um den Inhalt des Feldes **Geburtsdatum** entsprechend zu formatieren. Und im Falle des Feldes **Newsletter**, das den Datentyp **Boolean** hat, wollen wir den Inhalt in einem Kontrollkästchen anzeigen. Dazu benötigen wir eine etwas andere Struktur, die ein **CheckBox**-Steuerelement enthält, dessen Attribut **Checked** an das Feld **Newsletter** gebunden ist. Anschließend sieht das Fenster nach dem Starten der Anwendung wie in Bild 1 aus.

Grundlagen zum Filtern, Sortieren und Gruppieren

WPF stellt die Infrastruktur zum Filtern, Sortieren und Gruppieren von Daten, die in einem Steuerelement wie beispielsweise dem **ListView**-Steuerelement angezeigt werden, zur Verfügung. Dazu muss man wissen, dass das jeweilige Listen-Steuerelement nicht direkt an die Daten gebunden wird, die es anzeigen soll, sondern an eine View auf diese Daten. Diese View hat bei der in unserem Beispiel verwendeten Auflistung auf Basis der Schnittstelle **IList** den Typ **ListCollectionView**. Diese **ListCollectionView** bietet verschiedene Eigenschaften, die wir nutzen können – im Folgenden verwenden wir zunächst diese beiden:

- **Refresh**: Aktualisiert die Ansicht mit den aktuell angegebenen Filter-, Sortierungs- und Gruppierungseinstellungen

- **Filter**: Stellt ein, welche Funktion beim Aktualisieren zum Filtern verwendet werden soll



ID	Firma	Vorname	Nachname	Anrede	Strasse	PLZ	Ort	Land	EMail	Geburtsdatum	Alter	Newsletter
1	Krahn GbR	Adi	Stratmann	Herr	Kremser Straße 54	10589	Berlin	Deutschland	adi@stratmann.de	20.02.1939	82	☑
2	Göllner AG	Heidi	Eich	Frau	Moosstraße 30	42289	Wuppertal	Deutschland	heidi@eich.de	30.11.1971	50	☐
3	Peukert GmbH & Co. KG	Wernfried	Birk	Herr	Wiener Straße 78	22297	Hamburg	Deutschland	wernfried@birk.de	23.01.1931	90	☑
4	Bruder GmbH	Vitus	Krauße	Herr	Burgenlandstraße 77	20355	Hamburg	Deutschland	vitus@krauße.de	11.07.2009	12	☐
5	Mader KG	Jadwiga	Oehme	Frau	Peter-Rosegger-Straße 72	65187	Wiesbaden	Deutschland	jadwiga@oehme.de	26.01.1932	89	☐
6	Fleischhauer GmbH	Niko	Michel	Herr	Kindergartenstraße 42	12051	Berlin	Deutschland	niko@michel.de	01.03.1942	79	☑
7	Bolte KG	Siegert	Loos	Herr	Lenaustraße 80	66115	Saarbrücken	Deutschland	siegert@loos.de	14.05.1988	33	☐
8	Nitzsche GmbH & Co. KG	Michl	Schroth	Herr	Dr. Karl Renner-Straße 97	01129	Dresden	Deutschland	michl@schroth.de	15.01.1995	26	☑
9	Ziemann KG	Florentius	Wittek	Herr	Industriestraße 7	80638	München	Deutschland	florentius@wittek.de	05.07.1975	46	☑
10	Krahl KG	Gernulf	Riegel	Herr	Erzherzog-Johann-Straße 37	81545	München	Deutschland	gernulf@riegel.de	23.08.1955	66	☑
11	Becker AG	Tristan	Hübsch	Herr	Jahnstraße 78	65193	Wiesbaden	Deutschland	tristan@hübsch.de	25.10.1988	33	☑
12	Runge GmbH	Heinfried	Steinert	Herr	Kaplanstraße 60	30159	Hannover	Deutschland	heinfried@steinert.de	18.08.1932	89	☑
13	Albert AG	Herma	Aigner	Frau	Burgstraße 31	22089	Hamburg	Deutschland	herma@aigner.de	02.11.1996	25	☐
14	Brink GbR	Mina	Dörfler	Frau	Schloßstraße 66	38118	Braunschweig	Deutschland	mina@dörfler.de	14.02.1958	63	☑
15	Quandt GmbH & Co. KG	Jo	Pickel	Frau	Kreuzstraße 29	79114	Freiburg	Deutschland	jo@pickel.de	28.10.2007	14	☐
16	Knoche KG	Heimbert	Rohrer	Herr	Lenaustraße 83	10785	Berlin	Deutschland	heimbert@rohrer.de	02.01.1940	81	☐

Bild 1: Die Liste der Kunden im **ListView**-Steuerelement

Um die **ListCollectionView** für unsere Zwecke zu nutzen, machen wir diese zunächst per Variable zugreifbar:

```
Private objListCollectionView As ListCollectionView
```

Diese Variable füllen wir gleich in der Konstruktor-Methode des Code behind-Moduls, indem wir die Methode **GetDefaultView** der **CollectionViewSource**-Klasse aufrufen und dieser einen Verweis auf die Liste übergeben, deren **ListCollectionView**-Element wir erhalten wollen:

```
Public Sub New()
    ...
    objListCollectionView = CollectionViewSource.GetDefaultView(Kunden)
```

In der gleichen Methode stellen wir die Eigenschaft **Filter** von **objListCollectionView** ein, und zwar auf die Funktion **KundenFilter**, die wir mit **AddressOf** angeben:

```
    objListCollectionView.Filter = AddressOf KundenFilter
End Sub
```

Bevor wir fortsetzen, fügen wir dem XAML-Code für unser Fenster ein **StackPanel** hinzu, dass ein **Label**-, ein **TextBox**- und ein **Button**-Element enthält:

```
<StackPanel Orientation="Horizontal">
    <Label>Nachname:</Label>
    <TextBox x:Name="txtFilterNachname" Width="100"></TextBox>
    <Button x:Name="btnFiltern" IsDefault="True" Click="btnFiltern_Click">Filtern</Button>
</StackPanel>
```

Das **TextBox**-Element soll den Filterausdruck entgegennehmen, das **Button**-Element soll das Filtern auslösen. Deshalb geben wir dafür eine Ereignismethode an, die wie folgt aussieht und lediglich die **Refresh**-Methode von **objListCollectionView** auslöst:

```
Private Sub btnFiltern_Click(sender As Object, e As RoutedEventArgs)
    objListCollectionView.Refresh()
End Sub
```

Für das Attribut **IsDefault** des **Button**-Elements stellen wir außerdem den Wert **True** ein, damit die Methode hinter dieser Schaltfläche auch beim Betätigen der Eingabetaste ausgelöst wird.

Da wir eine Funktion für die Eigenschaft **Filter** von **objListCollectionView** angegeben haben, wird diese beim Aufruf der **Refresh**-Methode aufgerufen, um die anzuzeigenden Elemente zu ermitteln. Diese Funktion wird tatsächlich für jedes Element der **objListCollectionView** zugrunde liegenden Liste aufgerufen. Das jeweilige

Element wird dabei mit dem Parameter **item** als Parameter übergeben. Die Funktion gibt einen **Boolean**-Wert als Ergebnis zurück, dass angibt, ob der aktuelle Eintrag angezeigt werden soll oder nicht. Wir prüfen also innerhalb der Funktion, ob der aktuelle Eintrag den Filterbedingungen entspricht und geben nur dann den Wert **True** zurück und anderenfalls den Wert **False**. In der folgenden Prozedur prüfen wir, ob das Feld **Nachname** mit dem Ausdruck beginnt, den der Benutzer in das Textfeld **txtFilterNachname** eingegeben hat. Dazu lesen wir zuerst den Wert des Parameters **item** in eine Variable namens **objKunde** mit dem Typ **Kunde** ein. Dann lesen wir den Wert der Eigenschaft **Nachname** des Objekts aus **objKunde** aus. Die **IIf**-Bedingung sorgt dafür, dass im Falle des Wertes eine leere Zeichenkette und anderenfalls der Wert des Feldes **Nachname** in der Variablen **strNachname** landet.

Der Rückgabewert ist das Ergebnis der Prüfung, ob der Inhalt von **strNachname** mit dem Wert aus **txtFilterNachname** beginnt, wobei der Zeichenkettenvergleich mit dem Parameter **StringComparison.OrdinalIgnoreCase** durchgeführt wird – Klein-/Großschreibung wird also nicht berücksichtigt:

```
Private Function KundenFilter(item As Object) As Boolean
    Dim objKunde As Kunde
    Dim strNachname As String
    objKunde = TryCast(item, Kunde)
    strNachname = IIf(objKunde.Nachname = Nothing, "", objKunde.Nachname)
    If Not (strNachname.StartsWith(txtFilterNachname.Text, StringComparison.OrdinalIgnoreCase)) Then
        Return False
    End If
    Return True
End Function
```

Wenn die Bedingung nicht erfüllt ist, liefert die Funktion mit der Anweisung **Return** den Wert **False** zurück und beendet die Funktion. Anderenfalls ist **True** das Ergebnis.

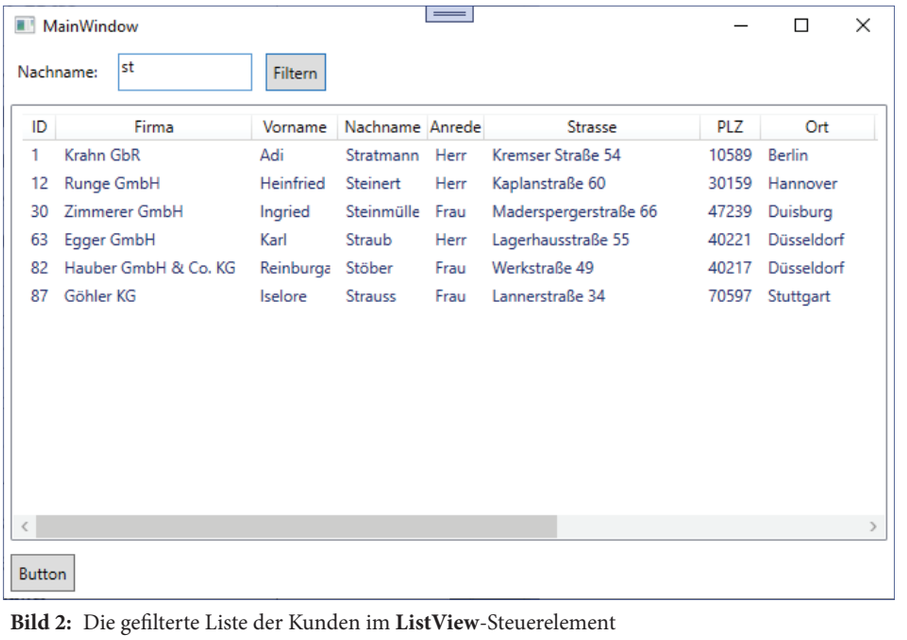


Bild 2: Die gefilterte Liste der Kunden im **ListView**-Steuerelement

Das Ergebnis sieht wie gewünscht aus (siehe Bild 2). Wenn Sie einen Wert in das Textfeld **txtFilterNachname** eingeben und die Eingabetaste oder die Schaltfläche Filtern betätigen, zeigt das

Das WPF-Steuerelement CheckBox

»Ja oder Nein – das ist hier die Frage!« Oder doch vielleicht? Wenn es um ein Steuerelement für Ja oder Nein, Wahr oder Falsch, 0 oder -1 geht, ist das Kontrollkästchen die erste Wahl. Oder, wie es in WPF heißt: CheckBox. Mit dem CheckBox-Steuerelement lassen sich üblicherweise zwei Zustände abbilden, die entweder durch einen Haken oder eben durch ein leeres Kästchen dargestellt werden. Ganz nebenbei bemerkt, kann man Kontrollkästchen auch noch für einen dritten Zustand verwenden, nämlich Null oder Nothing – je nach Programmiersprache. Dieser Artikel zeigt, was Sie mit dem CheckBox-Steuerelement von WPF anstellen können.

Das **CheckBox**-Steuerelement ist, was die Möglichkeiten für den Inhalt angeht, eines der einfacheren Steuerelemente von WPF – zumindest unter den Steuerelementen, die Sie an Daten binden können. Warum wir dieses erst im fünften Jahr dieses Magazins untersuchen, kann nur daran liegen, dass es so einfach scheint – dabei gibt es durchaus Eigenschaften, die nicht offensichtlich sind und die wir in diesem Artikel genauer untersuchen wollen.

Das wir das **CheckBox**-Steuerelement nun unter die Lupe nehmen, liegt daran, dass wir in einem anderen Artikel eine wichtige Eigenschaft benötigten, nämlich den dreifachen Zustand: Ist dieser aktiviert, kann das Steuerelement nicht nur die Werte Wahr und Falsch abbilden, sondern auch noch einen dritten Wert, der »nicht Wahr und nicht Falsch« oder auch **Null** (unter C#) oder **Nothing** (unter Visual Basic) lautet. Diesen benötigten wir in einer Suche, wo wir Kunden danach filtern wollten, ob sie einen Newsletter abonniert haben oder nicht. Und natürlich sollte die Suche auch ermöglichen, alle Kunden anzuzeigen, also mit und ohne Newsletterabonnement. Genau dazu brauchten wir den dritten Zustand des Kontrollkästchens. Doch eins nach dem anderen!

CheckBox-Steuerelement anlegen

Das **CheckBox**-Steuerelement legen Sie wie alle anderen Steuerelemente auch durch ein entsprechendes XAML-Element an, das logischerweise **CheckBox** heißt und mit x:Name einen Namen erhält, hier **chkBeispiel**:

```
<Grid>
    <CheckBox x:Name="chkBeispiel"></CheckBox>
</Grid>
```

Ohne weitere Elemente oder Formatierungen erscheint diese allerdings einfach links oben im Fenster (siehe Bild 1).

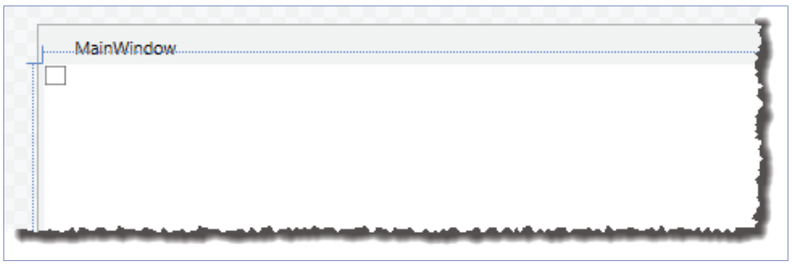


Bild 1: Einfache CheckBox

Beschriftung hinzufügen

Hier fehlt zumindest eine Beschriftung, die wir per **Label**-Element hin-

zufügen. Damit die beiden Elemente nebeneinander erscheinen, fassen wir diese in einem **StackPanel**-Element mit horizontaler Ausrichtung zusammen:

```
<StackPanel Orientation="Horizontal">
    <Label>Beispiel-Checkbox:</Label>
    <CheckBox x:Name="chkBeispiel"></CheckBox>
</StackPanel>
```

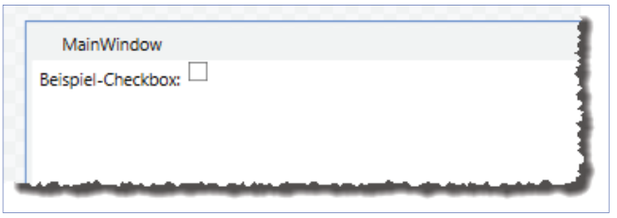


Bild 2: CheckBox mit Label

Bild 3: Vertikal ausgerichtete CheckBox mit Label

CheckBox-Elemente ausrichten

Damit erscheint die **CheckBox** allerdings nicht korrekt ausgerichtet (siehe Bild 2). Damit das geschieht, erstellen wir erstens einen Satz von **RowDefinition**-Elementen für das **Grid**, von denen die Höhe der ersten Zeile automatisch an den Inhalt angepasst werden soll. Zweitens stellen für das **CheckBox**-Element das Attribut **VerticalAlignment** auf den Wert **Center** ein:

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="*"></RowDefinition>
</Grid.RowDefinitions>
<StackPanel Orientation="Horizontal">
    <Label>Beispiel-Checkbox:</Label>
    <CheckBox x:Name="chkBeispiel" VerticalAlignment="Center"></CheckBox>
</StackPanel>
```

Das Ergebnis entspricht schon eher unseren Wünschen (siehe Bild 3).

Mehrere CheckBox-Elemente ausrichten

Wollen Sie jedoch mehrere Kontrollkästchen untereinander darstellen, wobei die Beschriftungen links und die Kontrollkästchen bündig dargestellt werden, benötigen wir eine Erweiterung des **Grid**-Elements um zwei Spalten. Dafür werfen wir das **StackPanel**-Element über Bord und teilen die **Label**- und **CheckBox**-Elemente auf die Zellen des Grids auf:

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

```
    <ColumnDefinition Width="*"></ColumnDefinition>
</Grid.ColumnDefinitions>
<Label>Beispiel-Checkbox:</Label>
<CheckBox x:Name="chkBeispiel" Grid.Column="1" VerticalAlignment="Center"></CheckBox>
<Label Grid.Row="1">Zweite Checkbox:</Label>
<CheckBox x:Name="chkZwei" Grid.Row="1" Grid.Column="1" VerticalAlignment="Center"></CheckBox>
```

Damit erhalten wir dann die ordentlich ausgerichteten **Label**- und **CheckBox**-Elemente aus Bild 4.

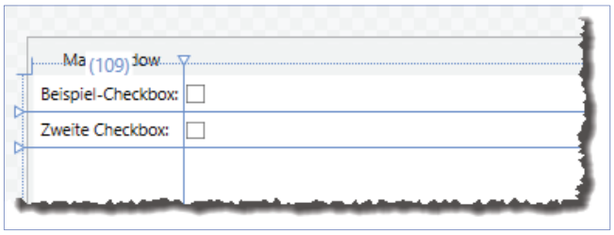


Bild 4: Mehrere ausgerichtete **CheckBox**-Elemente

CheckBox-Elemente links vom Label darstellen

Natürlich können Sie diese auch andersherum darstellen, also die **CheckBox**-Elemente auf der linken Seite platzieren. Auch das ist durchaus üblich, zumindest, wenn nur **CheckBox**-Elemente verwendet werden.

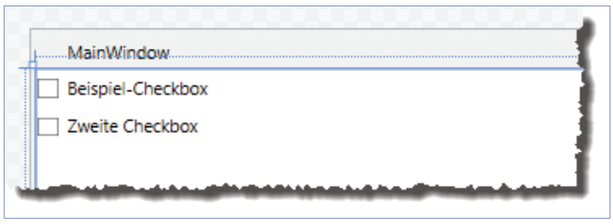


Bild 5: **CheckBox**-Elemente auf der linken Seite

Das hätte auch den Vorteil, dass die **CheckBox**- und die **Label**-Elemente auch bei Verwendung von **StackPanel**-Elementen immer korrekt ausgerichtet sind:

```
<StackPanel Orientation="Vertical">
    <StackPanel Orientation="Horizontal">
        <CheckBox x:Name="chkBeispiel" VerticalAlignment="Center"></CheckBox>
        <Label>Beispiel-Checkbox</Label>
    </StackPanel>
    <StackPanel Orientation="Horizontal">
        <CheckBox x:Name="chkZwei" VerticalAlignment="Center"></CheckBox>
        <Label>Zweite Checkbox</Label>
    </StackPanel>
</StackPanel>
```

Das Ergebnis finden Sie in Bild 5.

Das gelingt allerdings auch noch einfacher, denn wir können die Beschriftung in diesem Fall auch als Inhalt des CheckBox-Elements angeben. Der Code dazu sieht so aus:

```
<StackPanel Orientation="Vertical">
    <CheckBox x:Name="chkBeispiel">Beispiel-CheckBox</CheckBox>
    <CheckBox x:Name="chkZwei">Zweite Checkbox</CheckBox>
</StackPanel>
```

Verschiedene Zustände der CheckBox per XAML einstellen

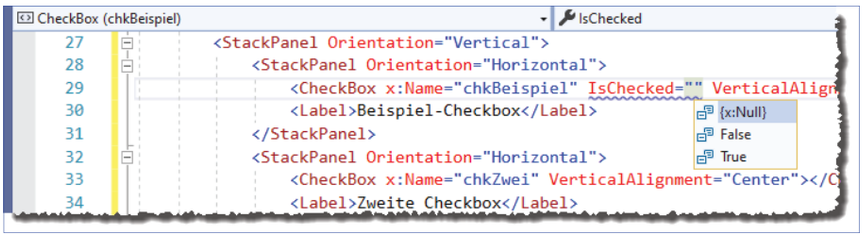


Bild 6: Werte für das Attribut **IsChecked**

Wie weiter oben erwähnt, kann das **CheckBox**-Steuerelement drei verschiedene Status einnehmen. Diese legen Sie im XAML-Code mit dem Attribut **IsChecked** fest. Die drei möglichen Werte lauten dort:

- **True**: Wahr, Ja, -1
- **False**: Falsch, Nein, 0
- **{x:Null}**: Kein Wert

Bild 6 zeigt die Auswahl der verschiedenen Werte per IntelliSense.

Der folgende XAML-Code definiert **CheckBox**-Elemente mit den drei verfügbaren Werten für das Attribut **IsChecked**:

```
<CheckBox x:Name="chkChecked" IsChecked="True" VerticalAlignment="Center"></CheckBox>
<Label>IsChecked = True</Label>
...
<CheckBox x:Name="chkNotChecked" IsChecked="False" VerticalAlignment="Center"></CheckBox>
<Label>IsChecked = False</Label>
...
<CheckBox x:Name="chkNull" IsChecked="{x:Null}" VerticalAlignment="Center"></CheckBox>
<Label>IsChecked = {x:Null}</Label>
```

Diese Elemente sehen nun wie in Bild 7 aus.

Dreifachen Zustand aktivieren

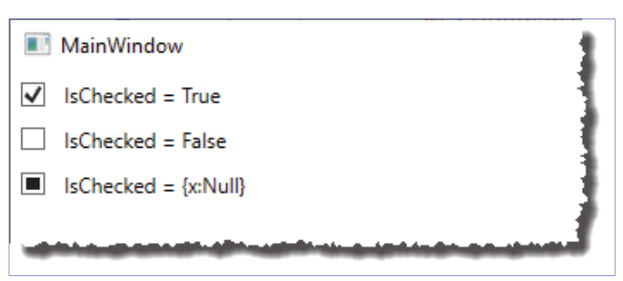


Bild 7: Verschiedene Zustände des **CheckBox**-Elements

Wenn wir das Projekt zum Debuggen starten, finden wir die drei verschiedenen Zustände wie in der Abbildung vor.

Klicken Sie allerdings mehrfach auf eines der **CheckBox**-Elemente, stellen Sie schnell fest, dass Sie immer nur zwischen den beiden Zuständen **Wahr** und **Falsch** wechseln können. Der Zustand **{x:Null}** ist nicht er-

reichbar. Das liegt daran, dass wir das Attribut **IsThreeState** nicht auf den Wert **True** eingestellt haben. Wenn wir das für eines der Elemente nachholen, können Sie zwischen allen drei Zuständen wechseln:

```
<CheckBox x:Name="chkNull" IsChecked="{x:Null}" IsThreeState="True" ...></CheckBox>
```

Der Wert des Attributs **IsThreeState** wirkt sich allerdings nur auf das Anklicken des **CheckBox**-Steuerelements aus. Per Visual Basic können Sie die Eigenschaft durch Einstellen der Eigenschaft **IsChecked** auf den Wert Nothing auf den dritten Zustand einstellen.

CheckBox per Visual Basic abfragen

Um den Wert eines **CheckBox**-Elements abzufragen, legen wir ein Kontrollkästchen namens **chkLesen** an und eine Schaltfläche namens **btnCheckBoxWert**. Diese soll beim Anklicken eine Methode auslösen, die den aktuellen Wert des **CheckBox**-Elements ausgibt:

```
<CheckBox x:Name="chkLesen" IsThreeState="True">Checkbox</CheckBox>
<Button x:Name="btnCheckBoxLesen" Click="btnCheckBoxLesen_Click">Wert ausgeben</Button>
```

Die Methode zeigt ein Meldungsfenster an, das einen Text und den Zustand des **CheckBox**-Elements anzeigt. Dabei ist wichtig, dass das **CheckBox**-Steuerelement keine Eigenschaft namens **Value** aufweist wie viele andere Steuerelemente. Stattdessen verwenden wir die Eigenschaft **IsChecked**, die wir ja auch schon im XAML-Code zum Einstellen des Zustands genutzt haben:

```
Private Sub btnCheckBoxLesen_Click(sender As Object, e As RoutedEventArgs)
    MessageBox.Show("Die CheckBox hat den Wert: " + chkLesen.IsChecked.ToString)
End Sub
```

Dies liefert für ein angeklicktes Kontrollkästchen den Wert **True**, für ein leeres Kontrollkästchen den Wert **False** und für den leeren Zustand eine leere Zeichenfolge. Das ist natürlich nicht der tatsächliche Wert, sondern nur der mit **ToString** in eine Zeichenkette konvertierte Wert.

Den tatsächlichen Wert finden wir beim Debuggen heraus, wenn wir einen Haltepunkt in die Zeile mit der **MessageBox** setzen und den Wert von **chkLesen.IsChecked** prüfen. Dieser hat für ein **CheckBox**-Steuerelement ohne Wert den Wert **Nothing**. Also können wir genauer wie folgt unterscheiden:

```
Private Sub btnCheckBoxLesen_Click(sender As Object, e As RoutedEventArgs)
    If chkLesen.IsChecked Is Nothing Then
        MessageBox.Show("Die Checkbox ist leer.")
    Else
        MessageBox.Show("Die CheckBox hat den Wert: " & chkLesen.IsChecked.ToString)
    End If
End Sub
```

Optionsgruppen mit GroupBox und RadioButtons

Wer von Access kommt, hat vermutlich hier und da Optionsgruppen genutzt. Ein solches Steuerelement bietet WPF nicht direkt an. Allerdings gibt es Elemente, mit dem wir Optionsgruppen nachbauen können. Dabei handelt es sich um das GroupBox- und RadioButton-Steuerelement. Wie Sie damit Optionsgruppen mit Optionen nachbilden, zeigen wir in diesem Artikel.

Einfache Optionsgruppe mit dem GroupBox-Element

Wenn wir von Optionsgruppen reden, benötigen wir auch noch Optionsfelder. Diese bilden wir unter WPF mit dem **RadioButton**-Element ab. In der Definition einer Optionsgruppe können wir nicht einfach die benötigten **RadioButton**-Elemente unter dem **GroupBox**-Element anordnen. Der Grund dafür ist, dass das **GroupBox**-Element nur ein weiteres Element direkt aufnehmen kann – also beispielsweise ein **StackPanel**-Element, um die Optionsfelder wie in Bild 1 nebeneinander oder untereinander anzuordnen.

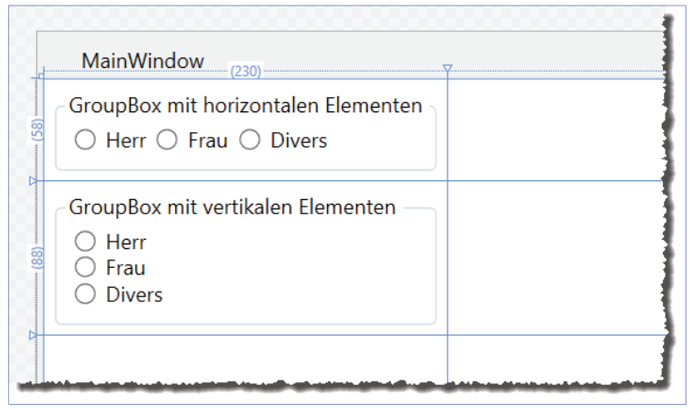


Bild 1: Einfache **GroupBox**-Beispiele

Die **GroupBox** mit horizontal angeordneten Elementen erhalten wir mit dem folgenden Code:

```
<GroupBox Header="Ein GroupBox-Element">
    <StackPanel Orientation="Horizontal">
        <RadioButton>Herr</RadioButton>
        <RadioButton>Frau</RadioButton>
        <RadioButton>Divers</RadioButton>
    </StackPanel>
</GroupBox>
```

Damit die Elemente den Abstand wie in der Abbildung angezeigt erhalten, haben wir noch das Attribut **Margin** für alle Elemente des Typs **RadioButton** angepasst – und gleich auch noch den inneren und äußeren Abstand des **GroupBox**-Elements:

```
<Window.Resources>
    <Style TargetType="GroupBox">
        <Setter Property="Margin" Value="5"></Setter>
        <Setter Property="Padding" Value="5"></Setter>
    </Style>
    <Style TargetType="RadioButton">
```

```
        <Setter Property="Margin" Va-
lue="0,0,5,0"></Setter>
    </Style>
</Window.Resources>
```

Die Definition der **GroupBox** mit vertikal angeordneten Elementen lautet wie folgt, wobei der Unterschied im Wert für das Attribut **Orientation** des **StackPanel**-Elements liegt:

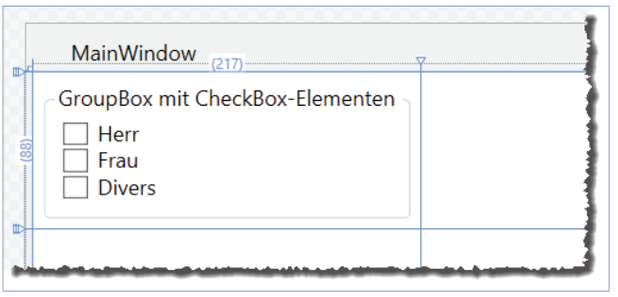


Bild 3: **GroupBox** mit **CheckBox**-Elementen

```
<GroupBox Header="Noch ein GroupBox-Element" Grid.Row="1">
    <StackPanel Orientation="Vertical">
        <RadioButton>Herr</RadioButton>
        <RadioButton>Frau</RadioButton>
        <RadioButton>Divers</RadioButton>
    </StackPanel>
</GroupBox>
```

Wie wir aus den Beispielen erkennen können, bietet das **GroupBox**-Element primär einen Rahmen und die Möglichkeit, oben in diesem Rahmen eine Bezeichnung einzufügen – und zwar mit dem Attribut **Header**.

GroupBox mit CheckBox-Elementen

Alternativ zu den **RadioButton**-Elementen können Sie auch **CheckBox**-Elemente nutzen:

```
<GroupBox Header="GroupBox mit CheckBox-Elementen" Grid.Row="2">
    <StackPanel Orientation="Vertical">
        <CheckBox>Herr</CheckBox>
        <CheckBox>Frau</CheckBox>
        <CheckBox>Divers</CheckBox>
    </StackPanel>
</GroupBox>
```

Diese erscheinen wie in Bild 3.

Header der GroupBox gestalten

Das **Header**-Element der **GroupBox** können Sie im Gegensatz zu dem der Optionsgruppe aus Access beliebig gestalten. Dazu nehmen Sie das Attribut aus dem eigentlichen Element heraus und formulieren es als Property Element. Hier können Sie dann beispielsweise Attribute defi-

Bild 2: **GroupBox** mit fetter Überschrift

nieren, welche die Schriftart für den Inhalt des Headers festlegen. In folgendem Beispiel haben wir im Element **GroupBox.Header** ein **Label**-Element hinzugefügt und dessen Attribut **FontWeight** auf **Bold** eingestellt:

```
<GroupBox Grid.Row="2">
    <GroupBox.Header>
        <Label FontWeight="Bold">GroupBox mit fetter Überschrift</Label>
    </GroupBox.Header>
    <StackPanel Orientation="Vertical">
        ...
    </StackPanel>
</GroupBox>
```

Das Ergebnis sehen Sie in Bild 2.

Und Sie können nicht nur die Schriftart der Überschrift der Optionsgruppe nach Belieben gestalten. Sie können dem **Header**-Element auch noch beliebige andere Elemente unterordnen – so können Sie beispielsweise auch noch ein Bild hinzufügen. Das Bild hinterlegen wir in einem neuen Ordner im Projekt namens **images**. Der Code zum Anzeigen des dort hinzugefügten Bildes sieht wie folgt aus:

```
<GroupBox Grid.Row="2">
    <GroupBox.Header>
        <StackPanel Orientation="Horizontal">
            <Image Source="images/users_meeting.png"></Image>
            <Label FontWeight="Bold">GroupBox mit fetter Überschrift</Label>
        </StackPanel>
    </GroupBox.Header>
    <StackPanel Orientation="Vertical">
    ...
    </StackPanel>
</GroupBox>
```

Das Property Element **GroupBox.Header** kann ebenfalls nur ein Element in der ersten Ebene enthalten, daher fassen wir das **Image**- und das **Label**-Element in einem **StackPanel**-Element mit horizontaler Orientierung zusammen. Das **Header**-Element mit Bild und Text sehen Sie in Bild 4.

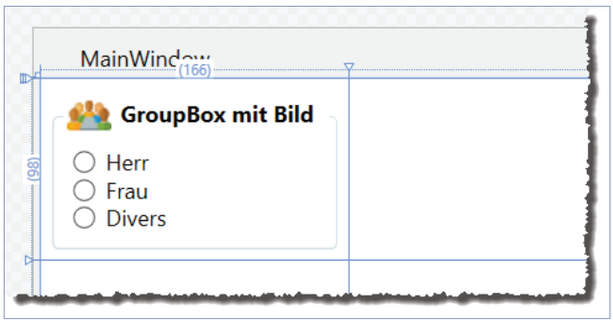


Bild 4: **GroupBox** mit Bild im **Header**-Element

Wert einer GroupBox auslesen

Die bisherigen Ausführungen haben gezeigt, wie Sie eine Optionsgruppe als **GroupBox** mit **RadioButton**-

Markierung bei Fokuserhalt der TextBox

Im Artikel »Das TextBox-Steuerelement« haben wir die Grundlagen zum TextBox-Steuerelement beschrieben. Dort haben wir noch längst nicht alle Möglichkeiten behandelt, die es bei der Programmierung von TextBox-Steuerelementen unter WPF gibt. Im vorliegenden Artikel schauen wir uns an, wie Sie das Verhalten einer TextBox beim Fokuserhalt steuern können, genau genommen: Wie erreiche ich, dass die Einfügemarke an einer bestimmten Stelle positioniert wird beziehungsweise der komplette Inhalt markiert wird? Und das nicht nur für den Fokuserhalt durch die Tabulator- oder Eingabetaste, sondern auch beim Anklicken mit der Maustaste.

Einfügemarke beim Fokuserhalt an bestimmter Position platzieren

Die folgenden Definitionen von **TextBox**-Elementen sollen die Einfügemarke an verschiedenen Positionen platzieren, wenn das Element den Fokus erhält. Das erste Element soll die Einfügemarke am Start des Textes platzieren. Dazu nutzen wir das Ereignis **GotFocus**:

```
<Label>Einfügemarke vorn:</Label>
<TextBox x:Name="txtCursorStart" GotFocus="txtCursorStart_GotFocus">Dies ist ein Beispieltext.</TextBox>
```

Dafür hinterlegen wir die folgende Ereignismethode:

```
Private Sub txtCursorStart_GotFocus(sender As Object, e As RoutedEventArgs)
    txtCursorStart.Select(0, 0)
End Sub
```

Diese verwendet die **Select**-Methode, um mit dem ersten Parameter die Startposition und mit der zweiten die Länge der Markierung festzulegen. Der Wert **0** für den ersten Parameter fügt die Einfügemarke ganz vorn ein und der Wert **0** für den zweiten Parameter sorgt dafür, dass die Einfügemarke nicht zu einer Markierung wird.

Für die Anzeige der Einfügemarke ganz hinten im Textfeld fügen wir ein weiteres Textfeld hinzu. Für dieses definieren wir das gleiche Ereignis:

```
<Label >Einfügemarke hinten:</Label>
<TextBox x:Name="txtCursorEnd" GotFocus="txtCursorEnd_GotFocus">Dies ist ein Beispieltext.</TextBox>
```

Die Methode, die durch dieses Ereignis ausgelöst wird, ermittelt für den ersten Parameter der **Select**-Methode jedoch mit der **Length**-Eigenschaft die Länge des im **TextBox**-Element enthaltenen und mit der Eigenschaft **Text** gelieferten Text. Der zweite Parameter hat wie im ersten Beispiel den Wert **0**:

```
Private Sub txtCursorEnd_GotFocus(sender As Object, e As RoutedEventArgs)
    txtCursorEnd.Select(txtCursorEnd.Text.Length, 0)
```

```
End Sub
```

Im dritten Textfeld wollen wir den vollständigen Text markieren, wenn das Textfeld den Fokus erhält:

```
<Label>Alles markieren:</Label>
<TextBox x:Name="txtCursorAll" GotFocus="txtCursorAll_GotFocus">Dies ist ein Beispieltext.</TextBox>
```

Wir könnten auch hier die **Select**-Methode verwenden und als Startposition der Markierung den Wert **0** und als Endposition die Länge des zu markierenden Textes übergeben. Eleganter ist in diesem Fall jedoch der Einsatz der **SelectAll**-Methode:

```
Private Sub txtCursorAll_GotFocus(sender As Object, e As RoutedEventArgs)
    txtCursorAll.SelectAll()
End Sub
```

All diese Möglichkeiten funktionieren gut, wenn der Benutzer den Fokus mit der Tabulator-Taste oder mit der Eingabe-Taste verschiebt (siehe Bild 1).

Sobald er jedoch mit der Maus in eines der Textfelder klickt, erscheint die Einfügemarke genau an der Stelle des Mausklicks. Beim Anklicken der dritten Schaltfläche, die den ganzen Text markieren sollte, blinkt sogar kurz die vollständige Markierung auf, bevor die Einfügemarke an der Mausposition erscheint.

In manchen Fällen wünschen wir uns jedoch auch beim Anklicken eines **TextBox**-Elements mit der Maus, dass dieses komplett markiert wird. Für diesen Fall müssen wir uns also eine alternative Möglichkeit einfallen lassen. Aber benötigen wir ein anderes Ereignis? Nein, wir ändern lediglich den Code des Ereignisses:

```
Private Sub txtCursorAll_GotFocus(sender As Object, e As RoutedEventArgs)
    Dim txt As TextBox
    txt = sender
    txt.Dispatcher.BeginInvoke(New Action(Sub() txt.SelectAll()))
End Sub
```

Was geschieht hier genau? Es handelt sich um ein Timing-Problem. **GotFocus** wird bereits ausgelöst, wenn Sie die Maustaste im **TextBox**-Steuerelement herunterdrücken. Dann markiert der Code den kompletten Inhalt und durch das Loslassen der Maustaste wird dann die Einfügemarke dort platziert, wo dies geschieht.

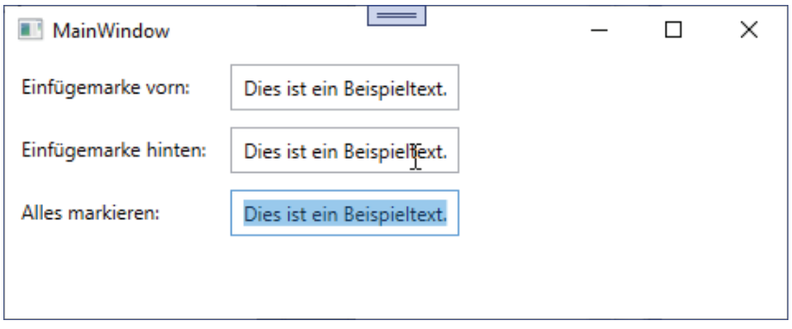


Bild 1: Markieren von Text

AfterUpdate-Ereignis für TextBox-Elemente

Das TextBox-Steuerelement bietet einige Ereignisse, die sich rund um die Dateneingabe drehen. Eines davon heißt TextChanged – dieses haben wir bereits im Artikel »Das TextBox-Steuerelement« vorgestellt. Während dieses Ereignis bei Eingabe jedes einzelnen Zeichens feuert, benötigen wir auch noch ein Ereignis, das uns mitteilt, wann der Benutzer die Eingabe in ein TextBox-Element abgeschlossen hat – beispielsweise, um nach der Eingabe der Nummer eines Datensatzes in einer Datensatznavigation direkt zum gewünschten Datensatz zu springen oder auch eine Suche nach dem eingegebenen Begriff zu starten. In diesem Artikel zeigen wir, wie das gelingt.

Im Artikel **Das TextBox-Steuerelement** (**www.datenbankentwickler.net/215**) haben wir bereits das Ereignis **TextChanged** vorgestellt. Dieses Ereignis ist praktisch, aber in manchen Fällen benötigen wir doch noch weitere Möglichkeiten zum Programmieren von Textfeldern. Das Ereignis **TextChanged** feuert nach jeder Änderung des enthaltenen Texts, also auch dann, wenn der Benutzer nur ein Zeichen hinzugefügt oder entfernt hat.

Wenn wir aber beispielsweise wissen möchten, wann die Änderung abgeschlossen ist – also beispielsweise durch Betätigen der Eingabetaste oder der Tabulator-Taste oder durch Verschieben des Fokus auf ein anderes Steuerelement mit der Maus –, dann benötigen wir andere Ereignisse.

Ereignis beim Beenden der Eingabe

Was wir suchen, ist ein Pendant zum Ereignis **Nach Aktualisierung** beziehungsweise **AfterUpdate** von Textfeldern unter Microsoft Access.

Im ersten Näherungsschritt wollen wir das Betätigen der Eingabe- oder Tabulatortaste bei der Eingabe in das Textfeld abfangen. Dazu schauen wir uns zunächst an, wann genau beim Betätigen einer dieser Tasten unter Access das **AfterUpdate**-Ereignis ausgelöst wird. Dabei stellen wir fest, dass dies bereits beim Herunterdrücken der Taste geschieht. Also bauen wir genau dieses Verhalten nach – und zwar mit der **KeyDown**-Methode.

Dazu legen wir folgendes **TextBox**-Element an und hinterlegen direkt das Attribut **KeyDown** mit der Angabe der auszuführenden Ereignismethode:

```
<TextBox x:Name="txtAfterUpdate" KeyDown="txtAfterUpdate_KeyDown"></TextBox>
```

Diese soll die folgende Ereignismethode auslösen:

```
Private Sub txtAfterUpdate_KeyDown(sender As Object, e As KeyEventArgs)
    Dim txt As TextBox
    txt = sender
    Select Case e.Key
        Case Key.Enter, Key.Tab
```

```
            MessageBox.Show("Der Text lautet '" & txt.Text & "'")
    End Select
End Sub
```

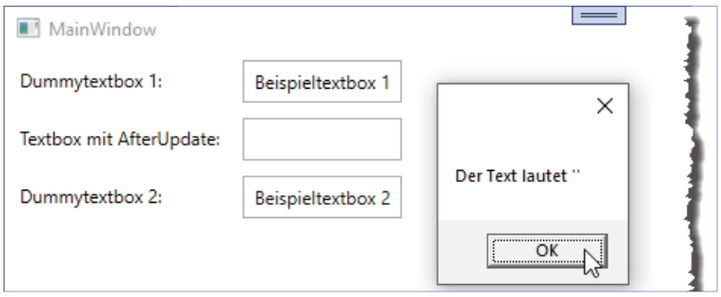


Bild 1: Ausgabe einer Meldung beim Verlassen des Feldes

Die Methode referenziert das mit dem Parameter **sender** übergebene auslösende Steuerelement mit der Variablen **txt** und prüft dann, ob das mit dem Parameter **e** gelieferte Objekt des Typs **KeyEventArgs** für die Eigenschaft **Key** entweder den Wert **Key.Enter** oder **Key.Tab** liefert. Ist das der Fall, soll eine Meldung mit dem aktuellen Inhalt des Textfeldes ausgegeben werden.

Das Ergebnis ist für den ersten Zwischenschritt schon nicht schlecht: Wenn wir das Textfeld mit der Eingabe- oder Tabulatortaste verlassen, wird das Ereignis ausgelöst und gibt den aktuellen Text aus (siehe Bild 1).

AcceptsReturn und AcceptsTab

Es gibt jedoch zwei Attribute namens **AcceptsReturn** und **AcceptsTab**, die angeben, ob das Betätigen der Eingabe- beziehungsweise der Tabulatortaste sich auf den Inhalt des Textfeldes auswirkt oder ob diese als Abschluss der Eingabe interpretiert werden sollen.

Stellen Sie beispielsweise **AcceptsReturn** auf **True** ein, dann löst die Betätigung der Eingabetaste innerhalb des Textfeldes nicht das Ereignis **KeyDown** aus. Gleiches gilt für die Tabulatortaste in Verbindung mit dem Attribut **AcceptsTab**.

Ereignis nur nach Änderung

Nun wollen wir aber dass Ereignis After Update abbilden, was nur dann ausgelöst wird, wenn der Benutzer gegenüber dem vorherigen Zustand auch eine Änderung am Inhalt des Textfeldes durchgeführt hat. Wir müssen also auf irgendeine Weise abgleichen, ob der beim Auslösen der Methode im **TextBox**-Element enthaltene Wert sich von dem unterscheidet, der vor der Änderung vorhanden war.

Alternative: Datenbindung mit UpdateSourceTrigger

Sie können auch einfach eine der Datenbindungseigenschaften von WPF nutzen. In diesem Fall hinterlegen wir in der Code behind-Klasse des Fensters eine private Variable mit einer entsprechenden Property mit Getter und Setter namens **GebundenerText**:

```
Private _GebundenerText As Object

Public Property GebundenerText
    Get
        Return _GebundenerText
    End Get
```

Typumwandlung unter VB.NET

Unter VB.NET gibt es verschiedene Möglichkeiten, den Typ von Variablen umzuwandeln. Dieser Artikel erläutert den Unterschied zwischen impliziter und expliziter Typumwandlung, die Bedeutung von Option Strict, die verschiedenen Typumwandlungsfunktionen wie CBool und die Methoden der Convert-Klasse. Außerdem gibt es noch flexible Umwandlungsfunktionen wie CType, DirectCast oder TryCast – auch diese schauen wir uns genauer an.

Visual Basic.NET bietet sehr viele verschiedene Datentypen. Manchmal müssen Sie Inhalte von einem zum anderen Datentyp konvertieren. Wie das gelingt, zeigen die folgenden Abschnitte.

Implizite Typumwandlung

Die implizite Typumwandlung erfolgt bei der Zuweisung eines Wertes oder eines Objekts zu einer Variablen. Diese Typumwandlung gelingt jedoch nur, wenn der umzuwandelnde Datentyp ein niederwertiger Datentyp ist und der Zieldatentyp ein höherwertiger Datentyp. Das bedeutet zum Beispiel, dass alle Datentypen in den Typ **Object** umgewandelt werden können, da dieser der oberste Datentyp in der Hierarchie der Datentypen ist. Die nachfolgenden Beispiele liefern gültige Zuweisungen, da der Zieldatentyp jeweils **Object** ist:

```
Dim objInteger As Object
objInteger = 10
Dim objKunde As Object
objKunde = New Kunde
```

Für die Zuweisung eines neuen **Object**-Elements zur Objektvariablen des Typs **Kunde** wie in der folgenden Anweisung gelingt das nicht:

```
Dim objKunde2 As Kunde
objKunde2 = New Object
```

Hier erhalten wir eine Fehlermeldung (siehe Bild 1).

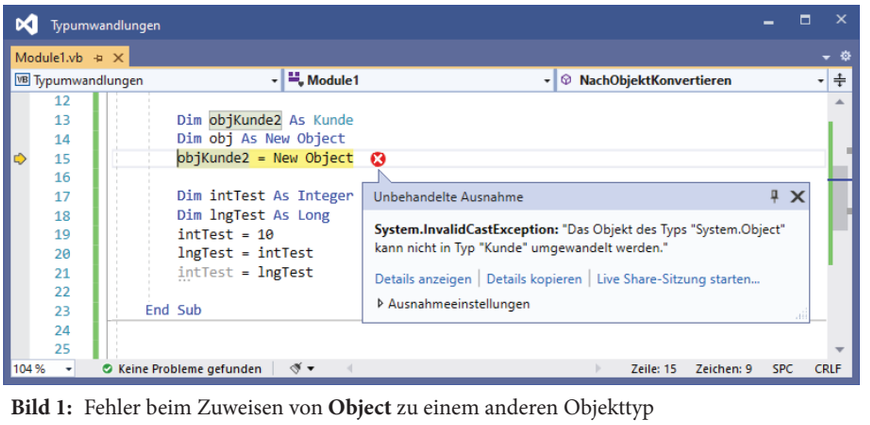


Bild 1: Fehler beim Zuweisen von **Object** zu einem anderen Objekttyp

Anzeige von ungültigen Zuweisungen per Option aktivieren

Sie können solche Fehler schon vor dem Ausführen ausschließen. Dazu stellen Sie einfach eine spezielle Option namens **Option Strict** auf den Wert **On** ein. Diese finden sie, wenn Sie

im Projektmappen-Explorer doppelt auf den Eintrag **My Project** klicken und dort zum Bereich **Kompilieren** wechseln. Hier finden Sie im Bereich **Kompilierungsoptionen** die Eigenschaft **Option Strict**, die wir auf den Wert **On** einstellen (siehe Bild 2).

Damit sorgen wir dafür, dass fehlerhafte Zuweisungen bereits im Codefenster angezeigt werden. Fahren Sie mit der Maus über einen entsprechend markierten Ausdruck, zeigt dieser das Problem an (siehe Bild 3). Hier sehen Sie weiter unten auch eine Markierung einer nicht gültigen Konvertierung von **Long** in **Integer**.

Option Strict als Standard setzen

Standardmäßig ist **Option Strict** für neue Projekte nicht auf den Wert **On** eingestellt, sondern auf **Off**. Wenn Sie wollen, dass diese Einstellung für alle neuen Projekte auf **On** gesetzt wird, öffnen Sie mit **Extras|Optionen** den

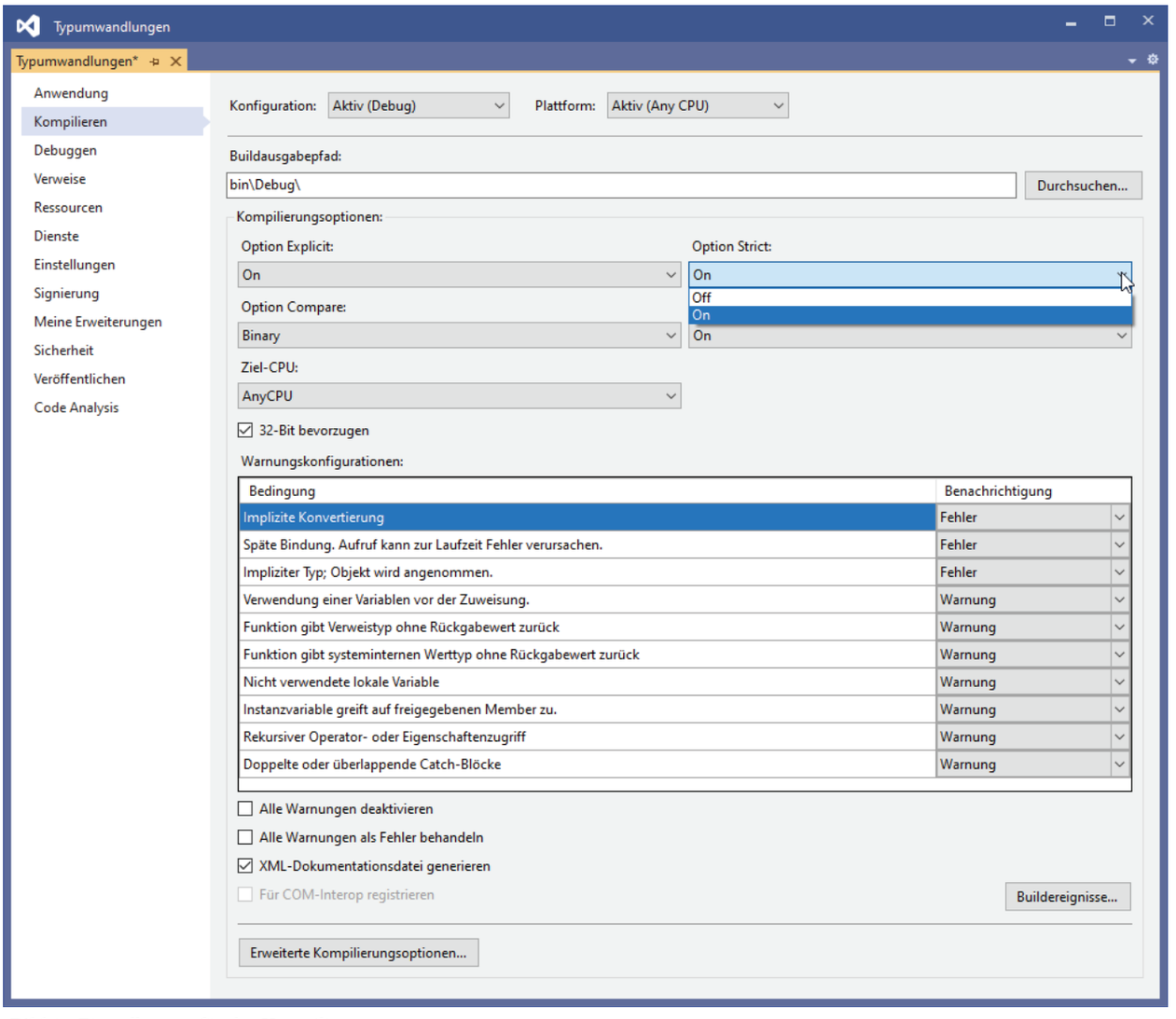


Bild 2: Einstellungen für das Kompilieren

Optionen-Dialog von Visual Studio und navigieren dort zum Bereich **Projekte und Projektmappen|VB-Standard**. Hier stellen Sie die Option **Option Strict** auf **On** ein (siehe Bild 4).

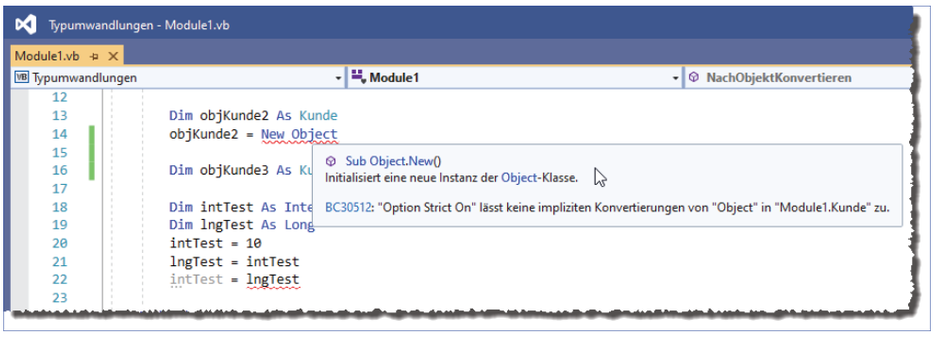


Bild 3: Anzeige nicht erlaubter Konvertierungen

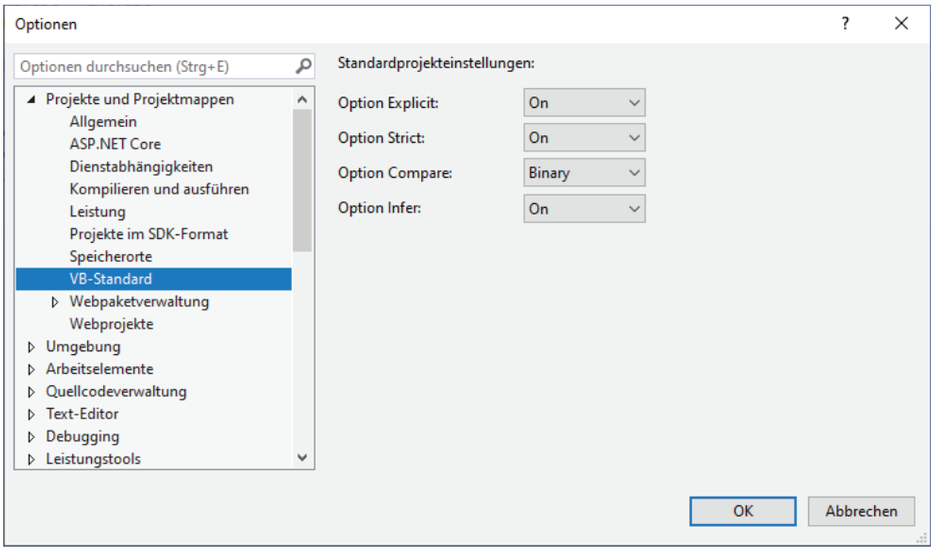


Bild 4: Voreinstellen von **Option Strict** für neue Projekte

Explizite Typkonvertierung

Neben der impliziten gibt es die explizite Typkonvertierung. Dabei verwenden wir eine Funktion, um die Konvertierung von dem einen in den anderen Typ vorzunehmen. Einige dieser Funktionen können Sie zum Konvertieren in spezielle Datentypen verwenden, zum Beispiel **CBool**. Damit konvertieren Sie den als Parameter angegebenen Ausdruck in ein Objekt des Typs **Boolean**:

```
Dim bolBeispiel As Boolean
bolBeispiel = CBool(-1)
```

Typumwandlungsfunktionen

Es gibt die folgenden Typumwandlungsfunktionen:

- **CBool**: Wandelt numerische Typen, **String** oder **Object** in den Datentyp **Boolean** um.
- **CByte**: Wandelt numerische Typen, **Boolean**, **String** und **Object** in den Datentyp **Byte** um.
- **CChar**: Konvertiert **String** und **Object** in den Datentyp **Char**.
- **CDate**: Wandelt **String** und **Object** in den Datentyp **Date** um.