

C# Databindings

- Das Event
- Das Property
- Oberflächen und Datenobjekte
- Das Binding
- Windows Forms und das Binding
- Bindings unter der Windows Presentation Foundation

Events / Ereignis

- Ein Event wird ausgeführt sobald ein Ereignis eingetreten ist.

```
#region INotifyPropertyChanged Member  
//EventHandler  
public event PropertyChangedEventHandler PropertyChanged;  
  
#endregion
```

- Es führt dann alle Methoden aus die in das Event Eingehängt wurden.

Einhängen

- Indem man sich auf ein Event einhängt, kann man Code ausführen lassen, sobald das Event aufgerufen wird.

```
public partial class Form1 : Form
{
    public void UsingEvents()
    {
        PropertyChangedBspClass myClass = new PropertyChangedBspClass();
        myClass.PropertyChanged += OnPropertyChanged; //Einhängen
        myClass.Text1 = "Löse Event aus"; //Changed event auslösen
        myClass.PropertyChanged -= OnPropertyChanged; //Aushängen
    }

    private void OnPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        string propertyName = e.PropertyName; //Property Namen auslesen
    }
}
```

Das Property

- Das Property
 - Das Property stellt die Methoden get/set bereit.
 - Dadurch kann ein Attribut Veröffentlicht werden und gleichzeitig der Verifikations-Code versteckt werden, genauso wie bei den Methoden get/set.
- Der Vorteil besteht darin das das Property sich ähnlich verhält wie ein Attribut.

Arbeiten mit Properties

- Das Property kann wie ein Attribut, zugewiesen und ausgelesen werden.

```
public static void main()
{
    PropertyChangedBspClass myClass = new
        PropertyChangedBspClass();
    myClass.Text = "hallo";           //Property zuweisen
    string text = myClass.Text;     //Property abrufen
}
```

Property Kurzschreibweise

```
class PropertyChangedBspClass : Window, INotifyPropertyChanged
{
    public int ShortProperty { get; set; }
}
-----
private int backingField;
public int PropertyWithBackingField
{
    get { return backingField; }
    set { backingField = value; }
}
}
```

Dependency Property

```
class PropertyChangedBspClass : Window, INotifyPropertyChanged
{
    public int DependencyProperty
    {
        get { return (int)GetValue(DependencyPropertyProperty); }
        set { SetValue(DependencyPropertyProperty, value); }
    }

    // Using a DependencyProperty as the backing store for DependencyProperty.
    // This enables animation, styling, binding, etc...
    public static readonly DependencyProperty DependencyPropertyProperty =
        DependencyProperty.Register("DependencyProperty", typeof(int),
            typeof(PropertyChangedBspClass), new UIPropertyMetadata(0));
}
```

Property mit Backingfield

```
class PropertyChangedClass : INotifyPropertyChanged
{
    private string text;    // Backing Field
    public string Text     // Property
    {
        get { return text; }
        set
        {
            text = value;
            this.OnPropertyChanged("Text");
        }
    }

    private void OnPropertyChanged(string propertyName)
    {
        // Property Changed Event
        if (PropertyChanged != null) PropertyChanged(this, new
            PropertyChangedEventArgs(propertyName));
    }

    #region INotifyPropertyChanged Members
    // Property Changed Event
    public event PropertyChangedEventHandler PropertyChanged;

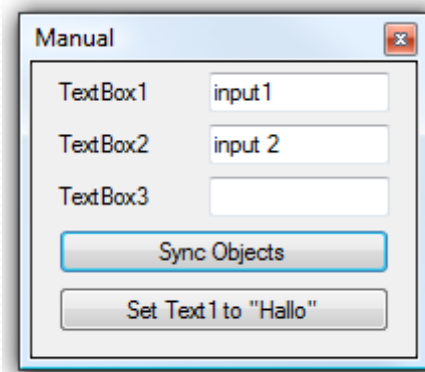
    #endregion
}
```



Arbeiten mit Oberflächen

- Grundlegendes Problem
 - Datenobjekte sind getrennt von Oberflächenobjekten
 - Änderungen an den Datenobjekten müssen an die Oberfläche durch geschleift werden.
 - Änderungen an den Oberflächenobjekten müssen an die Oberfläche durch geschleift werden.

Objekte Manuel Synchronisieren



```
public partial class EasySync : Form
{
    BspObject myObject = new BspObject();

    private void SyncGuiToObject(BspObject source)
    {
        myObject.text1 = textBox1.Text;
        myObject.text2 = textBox2.Text;
        myObject.text3 = textBox3.Text;
    }

    private void btnSyncObjects(object sender, EventArgs e)
    {
        //Aufruf wenn Button gedrückt wird.
        SyncGuiToObject(bspObject);
    }

    private void btnChangeText1
        (object sender, EventArgs e)
    {
        bspObject.text1 = "Hallo";
    }
}
```



Problem

- Was passiert wenn das Objekt sich im Hintergrund ändert?
- Oder wenn wir viel mehr Properties und Textboxen haben?

Textbox und TextChanged

```
// Designer Code (wird geladen sobald eine Form initialisiert wird)
this.textBox1.TextChanged +=
    new System.EventHandler(this.textBox1_TextChanged);

// Arbeitscode
BspObject bspObject = new BspObject();

private void textBox1_TextChanged(object sender, EventArgs e)
{
    bspObject.text1 = textBox1.Text; // Setzen unseres Objektes
}
```

- Nachteil
 - bspObject.text1 wird immer noch überschrieben

Von Hand

```
// Designer Code ...

// Arbeits Code ...
BspObject bspObject = new BspObject();
private void init()
{
    // Einhängen in das DatenObjekt
    this. bspObject.PropertyChanged += OnPropertyChanged;
}

private void OnFormClose(object sender, EventArgs e)
{
    // Aushängen aus dem DatenObjekt
    this. bspObject.PropertyChanged -= OnPropertyChanged;
}

private void textBox1_TextChanged(object sender, EventArgs e)
{
    bspObject.text1 = textBox1.Text; // Setzen unseres Objektes
}

private void OnPropertyChanged(object sender, PropertyChangedEventArgs e)
{
    if(e.PropertyName == "text1")
        textBox1.Text = bspObject.text1; // Setzen unseres Objektes
}
```

Nachteil

- Bisschen viel Code für ein Property
- Wir müssen uns für jedes Property ein/aus-hängen welches wir auf der Oberfläche binden wollen.

Das Binding

- Das Binding Synchronisiert ein Datenobjekt mit einem Oberflächenobjekt .
- Wie?
 - Das Binding hängt sich in das PropertyChanged event des Datenobjektes ein.
 - Bei Änderung des Datenobjektes veranlasst das Binding das Oberflächenobjekt sich zu aktualisieren.
 - Beim Austausch hängt sich das Binding wieder aus dem Datenobjekt aus.

Bindings im Designer

```
// Designer Code

this.bindingSource1 = new
System.Windows.Forms.BindingSource(this.components);

this.textBox1.DataBindings.Add(new System.Windows.Forms.Binding("Text",
this.bindingSource1, "text1", true));

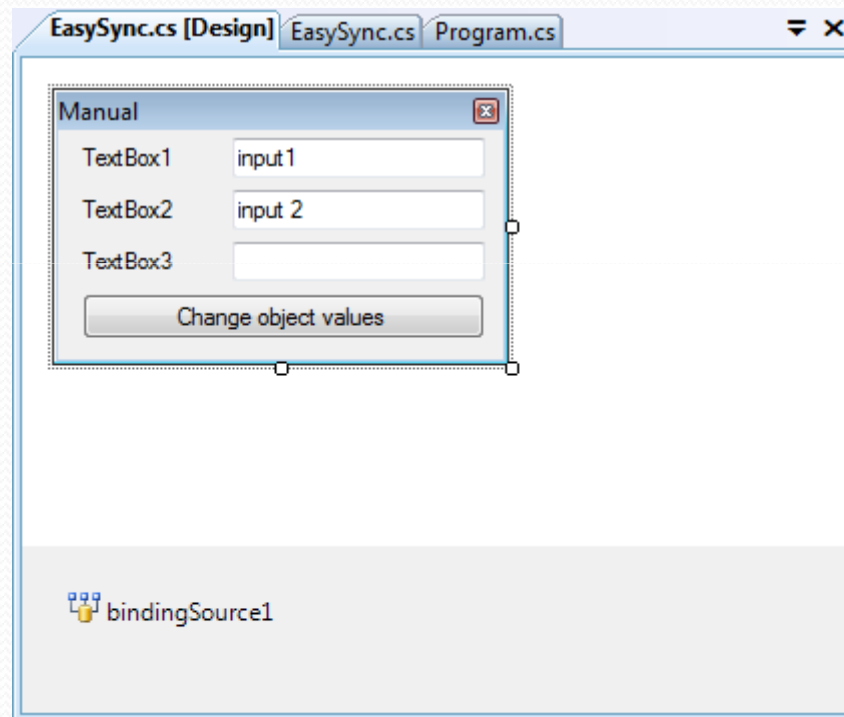
// auch für textbox 2 und 3

// Arbeitscode

private void EasySync_Load(object sender, EventArgs e)
{
    //Zuweisen des Datenobjectes auf das Binding
    bindingSource1.DataSource = bspObject;
}

private void EasySync_FormClosed(object sender, FormClosedEventArgs e)
{
    //Wichtig damit sich das Binding aus dem Datenobject wieder aushängt
    bindingSource1.DataSource = typeof(BspObject);
}
```

Binding unter Windows Forms





Vor/Nachteile

- Vorteile

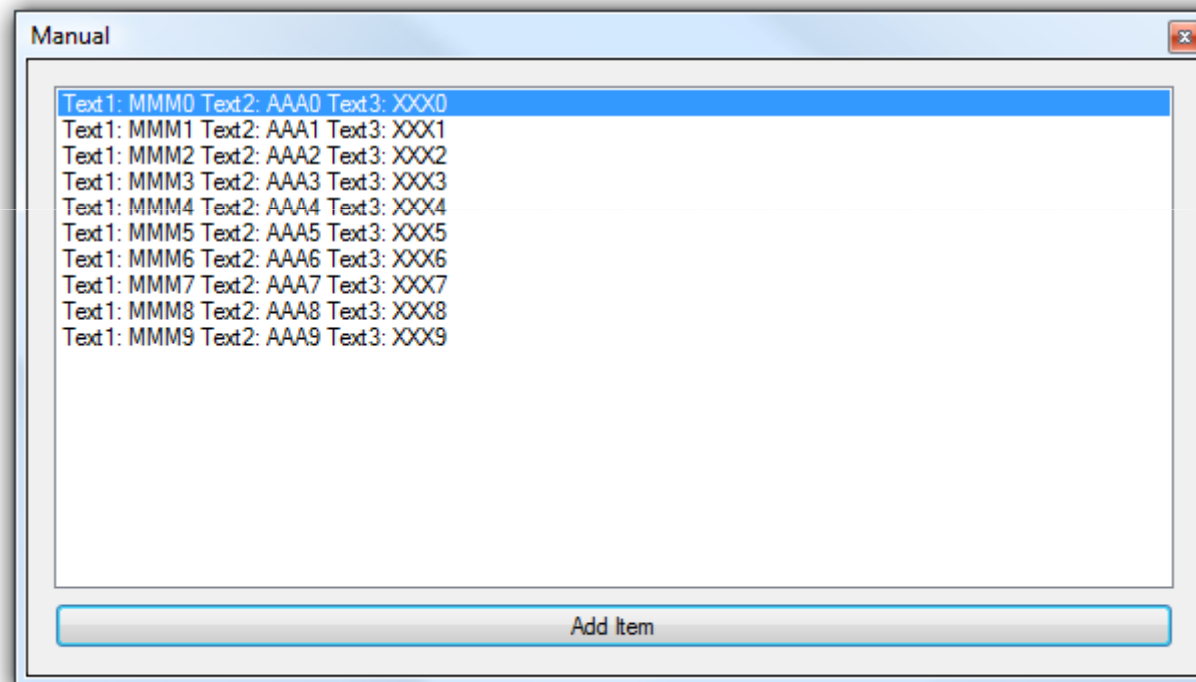
- Die Oberfläche ist mit minimalem Aufwand mit dem Objekt verknüpft
- Kein weiterer Code mehr nötig um Objekte aktuell zu halten

- Nachteile

- Diese Automatisierung kostet Zeit
 - Was bei benutzeroberflächen zu vernachlässigen ist
- Bei größeren Objekten ist der Designer Äußerst langsam

Binden von Listen

- Listen können auch gebunden werden



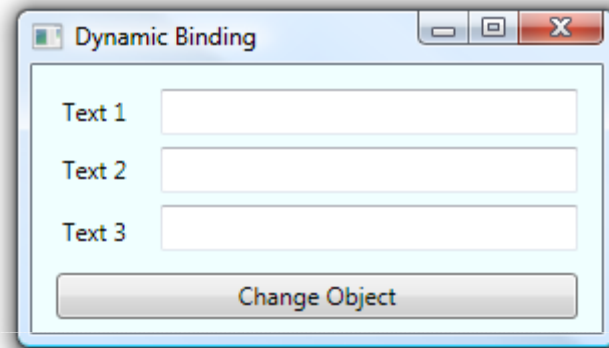
Binding typen in WPF

- Statisches Binding
 - Werden einmal gesetzt (Satic Ressources)
- Dynamisches Binding
 - Bindet auf Properties deren Daten-Objekte sich ändern oder getauscht werden können.
- Relatives Binding
 - Bindet auf Properties innerhalb der XML Hierarchie (diese können auch wechseln).
 - Das zu bindende Objekt anhand des Namens innerhalb der XML Hierarchie ermitteln

Static Binding

```
<Window.Resources>  
    <Brush x:Key="background">Azure</Brush>  
</Window.Resources>  
  
<Grid Background="{StaticResource background}">  
</Grid>
```

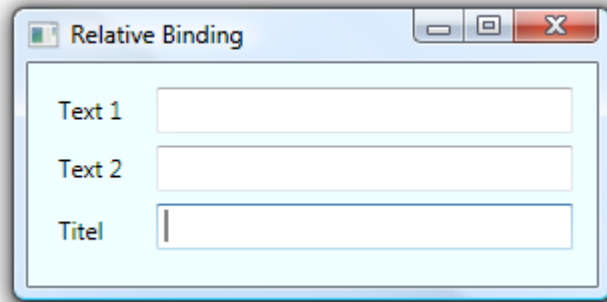
Dynamic Binding



```
<Window x:Class="WpfApplication1.DynamicBinding"
        Title="Dynamic Binding">

<Window.Resources>
    <Brush x:Key="background">Azure</Brush>
</Window.Resources>
<Grid Background="{StaticResource background}">
    <TextBox Text="{Binding Path=Text1}" />
    <TextBox Text="{Binding Path=Text2}" />
    <TextBox Text="{Binding Path=Text3}" />
</Grid>
```

Relative Binding



```
<Window x:Class="WpfApplication1.RelativeBinding"
        Title="Relative Binding" Name="myWindow">

    <Grid>
        <TextBox Text="start" Name="textBox1"/>
        <TextBox Text="{Binding ElementName=textBox1, Path=Text}"
                Name="textBox2" />
        <TextBox Text="{Binding ElementName=myWindow, Path=Title}"
                Name="textBox3" />
    </Grid>
</Window>
```



Vorteile

- Funktioniert ein Binding nicht richtig zeigt es einfach nichts an.
- Es können alle Objekte als DataSource dienen solange der Pfad im Binding korrekt ist, wird dies vom Binding Verstanden.
- Kurze Implementierung durch XML automatisierung

Nachteile

- Keine Unterstützung durch den Refactoring von Property Namen da alle Objekte als Datasource dienen
- Bei Fehlern nahezu keine Hilfe durch Fehlermeldungen
- Falls ein Binding abstürzt dann nur noch zur Laufzeit auch in Windows Forms



Ende

- Fragen?