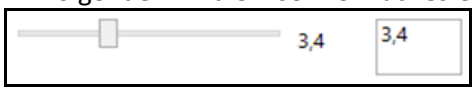
UserControls

Inhaltsverzeichnis

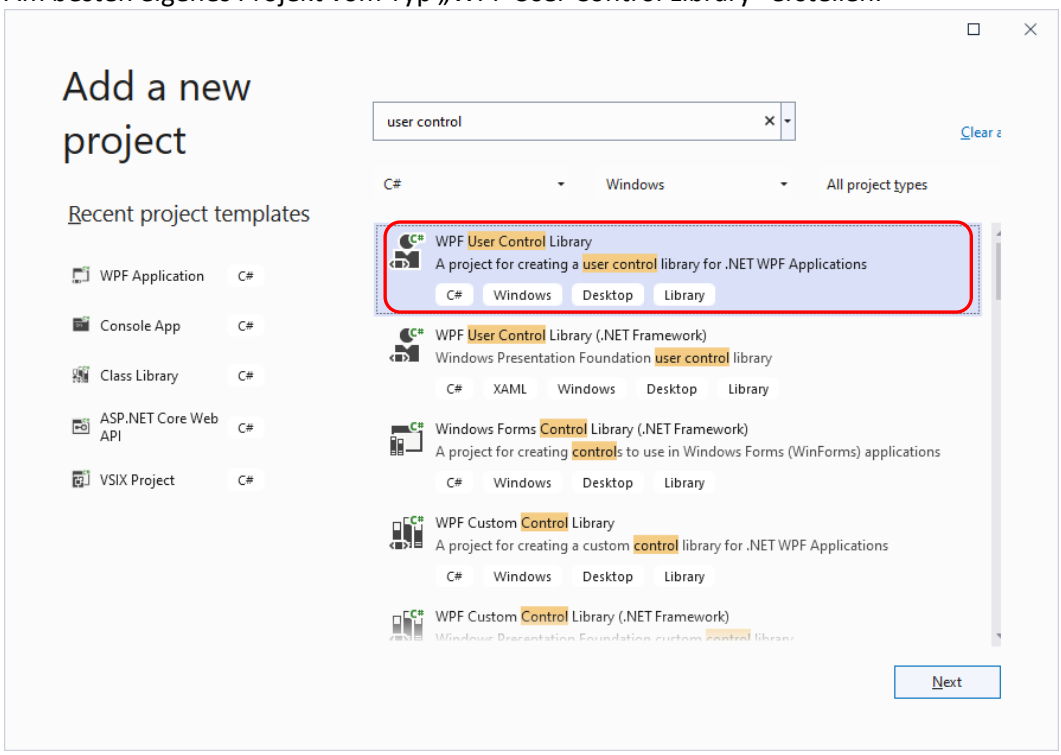
1 User Control

Ein UserControl ist ein neu erstelltes Control, das **aus bereits vorhandenen anderen Controls erzeugt** wird. Man erweitert damit also die Toolbox. Man erstellt es also im Wesentlichen mit Komponenten aus der Toolbox wie bei einem Window auch.
Im Folgenden wird ein sehr einfaches UserControl erzeugt und benutzt, das so aussieht:

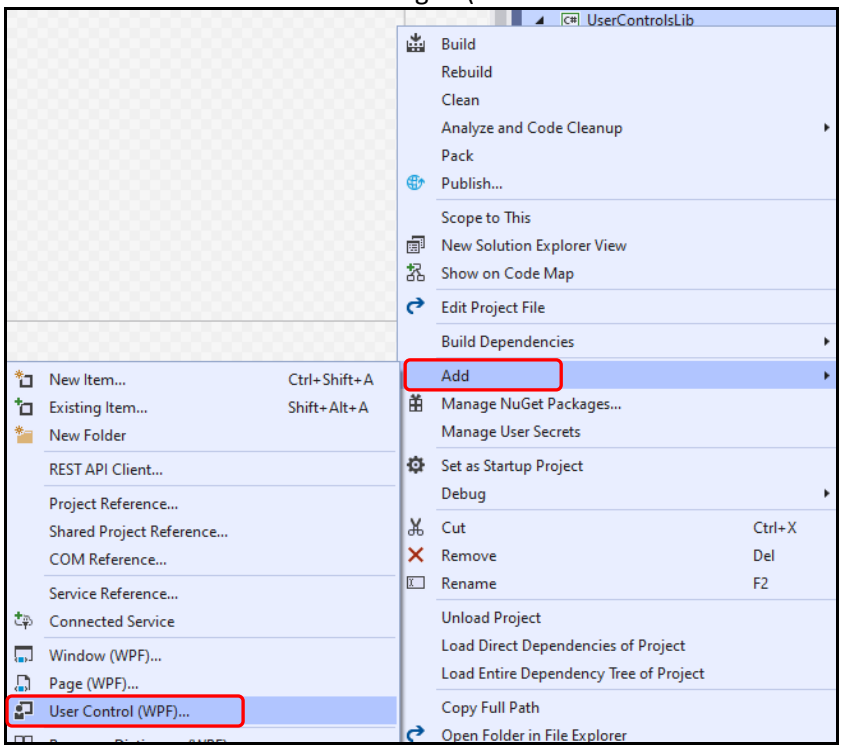


1.1 UserLib-Projekt

Am besten eigenes Projekt vom Typ „WPF User Control Library“ erstellen.



Darin dann die UserControls erzeugen (bzw. UserControl1 umbenennen):



Mit der Toolbox gestalten, so wie man das auch für ein Window machen würde, z.B. ein Slider mit Label und Textbox:

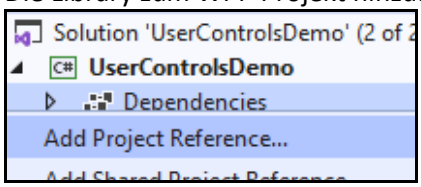
1.2 Intern

Der aktuelle Wert des Sliders soll auch im Label und in der Textbox angezeigt werden. Das UserControl muss also in sich konsistent sein.
Dazu im **ValueChanged**-Callback des Sliders die beiden anderen Controls den Wert anzeigen lassen:

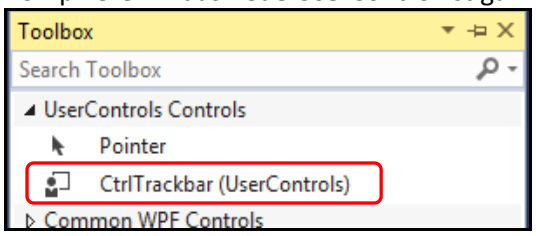
```
private void SliVal_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)
{
  lblVal.Content = $"{e.NewValue:0}";
  txtVal.Text = $"{e.NewValue:0}";
}
```

1.3 Verwenden

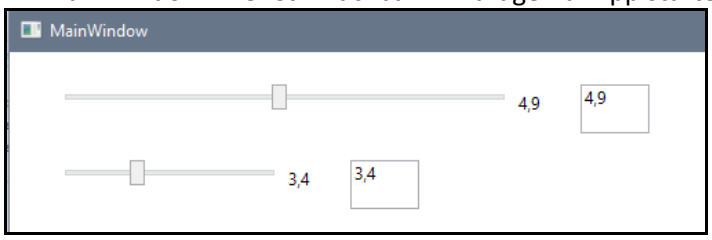
Die Library zum WPF-Projekt hinzufügen (References):



Kompilieren➔das neue UserControl ist ganz oben in der Toolbox sichtbar:



Im MainWindow zwei CtrlTrackbar hinzufügen u. App starten:



1.4 Properties – Konfiguration des UserControls

Da unser neues Control von UserControl abgeleitet ist, erbt es davon schon sehr viele Properties, die auch im Properties-Fenster verfügbar sind und eingestellt werden können.
Möchte man zusätzliche Eigenschaften definieren, muss man im UserControl nur ganz normale Properties definieren. Da man aber die Einstellung üblicherweise auch an ein darin enthaltenes Control übergeben möchte (in unserem Fall der Slider), erstellt man die Property mit dem Code-Snippet **propfull**).
Meist wird mit dem gesetzten Wert ein internes Control entsprechend geändert (im Beispiel der Wert des Sliders):

```
private double val;
public double Val
{
  get => val;
  set
  {
    val = value;
    sliVal.Value = val;
  }
}
```

Allerdings kann man sich (in diesem speziellen Fall) eigentlich die private Variable val sparen, weil ja der Wert ohnehin schon im Slider gespeichert wird. Man kann die Property also so abkürzen:

```
public double Val
{
  get { return sliVal.Value; }
  set { sliVal.Value = value; }
}
```

bzw.

```
public double Val
{
  get => sliVal.Value;
  set => sliVal.Value = value;
}
```

Im XAML-Designer erscheint dann die Property-Fenster diese Property im Bereich „Miscellaneous“.

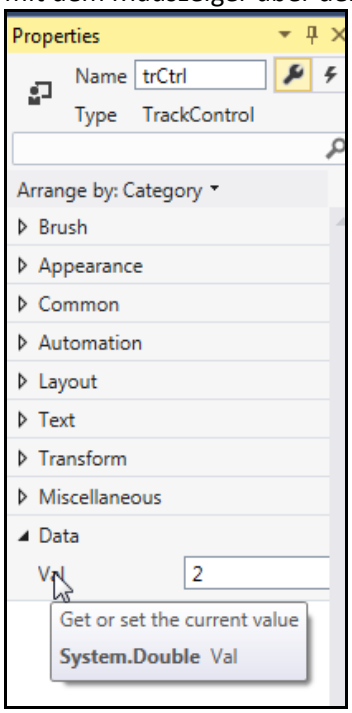
1.4.1 Category/Description

Über die Attribute Category und Description kann man die Anzeige im Properties-Fenster steuern:

```
[Category("Data"), Description("Get or set the current value")]
public double Val
{
  get { return slider.Value; }
  set { slider.Value = value; }
}
```

Hinweis: **`using System.ComponentModel`** nicht vergessen

Damit wird die Property in der angegebenen Kategorie angezeigt. Die Beschreibung wird angezeigt, wenn man mit dem Mauszeiger über der Property bleibt:



1.1 Delegate/EventArgs definieren

Aus Gründen die im Unterreicht erläutert wurden, sollte man als Signatur immer **`(object sender, XxxEventArgs e)`** benutzen.
In einer eigenen Datei daher das Delegate definieren, am besten in jener Datei, in der auch die EventArgs definiert werden.

```
public class ValueChangedEventArgs : EventArgs
{
  public double Val { get; set; }
  public DateTime Clock { get; set; }
}
```

Dann entweder ein Delegate definieren:

```
public delegate void ValueChangedEventHandler(object sender, ValueChangedEventArgs e);
```

Oder kürzer das generische Delegate **`EventHandler<ValueChangedEventArgs>`** benutzen.
Bei Events gibt es die Konvention, dass es immer 2 Argumente gibt:

- **`object sender`**: der Sender übergibt mit **`this`** eine Referenz auf sich
- Eine von **`EventArgs`** abgeleitete Klasse, die die Daten kapselt

Mit dem zweiten Parameter wird die Push-Variante des Observer-Patterns ermöglicht, mit dem ersten Parameter die Pull-Variante.

1.2 Event definieren u. auslösen

Im UserControl wird jetzt ein Event definiert, das obigem Delegate entspricht:

```
public partial class CtrlTrackbar : UserControl
{
  public event EventHandler<ValueChangedEventArgs> ValueChanged;
```

Dieses Event wird immer dann ausgelöst, wenn sich der Wert des UserControls ändert. In unserem Fall ist das immer genau dann, wenn der Slider einen neuen Wert einstellt:

```
private void SliVal_ValueChanged(object sender, Rout
{
  lblVal.Content = $"{sliVal.Value:0.0}";
  txtVal.Text = $"{sliVal.Value:0.0}";
  if (ValueChanged == null) return;
  ValueChanged(this, new ValueChangedEventArgs
  {
    Val = sliVal.Value,
    Clock = DateTime.Now
  });
}
```

Wie bereits oben erwähnt kann das Event null sein, wenn sich kein Listener darauf registriert hat!
Beachte:

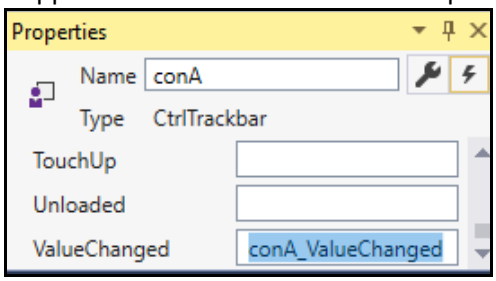
- Das Event muss mit Schlüsselwort **`event`** definiert werden.
- Der Typ des Events ist der vorher definierte Delegate-Typ.
- Das Auslösen des Events erfolgt durch Aufruf der Event-Variable. Die Parameter ergeben sich aus der Definition des Delegates.
- hat sich kein Listener registriert, ist der EventHandler **`null`** ➔ entsprechend prüfen!

1.3 Listener-Methode

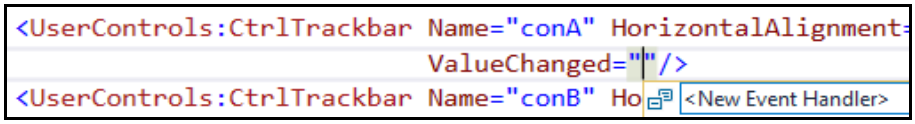
Die Zuweisung eines Listeners auf das Event erfolgt jetzt genau so wie auch z.B. bei einem Button:

- Doppelklick im Designer ➔ Default-Event wird registriert

- Doppelklick im Event-Tab bei den Properties im Designer:



- Attribut ins XAML schreiben



- im Code mit += einen Listener zuweisen

```
conA.ValueChanged += ConA_ValueChanged;
```

Die Listener-Methode im MainWindow muss genau die Signatur des Delegates haben:

```
private void ConA_ValueChanged(object sender, ValueChangedEventArgs e)
{
  Console.WriteLine($"Value changed to {e.Val:0.0}");
}
```

1.4 Weitere Infos

1.4.1 DefaultEvent

Üblicherweise möchte man bei der Verwendung eines Controls bei Doppelklick im Designer automatisch das gebräuchlichste Event implementieren, also z.B. bei einem Button das Click-Event.
Durch Angabe von DefaultEvent über der Klasse kann man auch hier ein Event dermaßen auszeichnen:

```
[DefaultEvent(nameof(ValueChanged))]
public partial class CtrlTrackbar : UserControl
{
  public event EventHandler<ValueChangedEventArgs> ValueChanged;
```

Hier bietet sich die Funktion **`nameof()`** an!

1.4.2 Null-Propagation

Um die oben erwähnte Nullpointer-Exception zu vermeiden, würde sich als bessere Variante jene mit Null-Propagation anbieten:

```
private void SliVal_ValueChanged(object sender, RoutedP
{
  lblVal.Content = $"{sliVal.Value:0.0}";
  txtVal.Text = $"{sliVal.Value:0.0}";
  ValueChanged?.Invoke(this, new ValueChangedEventArgs
  {
    Val = sliVal.Value,
    Clock = DateTime.Now
  });
}
```

1.4.3 Anzahl Listener

Man kann jederzeit ein event fragen, wie viele Listener registriert sind:

```
int nrListeners = ValueChanged.GetInvocationList().Length;
```