Урок 6

[Связывание данных. Триггеры. Обработка исключений.ListView.Виртуализация. 1](#_Toc488143182)

[1. Связывание данных 1](#_Toc488143183)

[1.1. Основные понятия связывания данных 1](#_Toc488143184)

[1.2. Синтаксис связывания данных 3](#_Toc488143185)

[1.3. Форматирование значений привязки и конвертеры значений 3](#_Toc488143186)

[1.4. Использование DataContext 6](#_Toc488143187)

[1.5. Обновление связывания данных(привязки). UpdateSourceTrigger. 7](#_Toc488143188)

[1.6. Реакция на изменения 8](#_Toc488143189)

[1.7. Отладка связывания данных 10](#_Toc488143190)

[2. Триггеры 12](#_Toc488143191)

[2.1. Триггеры свойств 13](#_Toc488143192)

[2.2. Триггеры данных 13](#_Toc488143193)

[2.3. Триггеры событий 14](#_Toc488143194)

[3. ListView 15](#_Toc488143195)

[4. Виртуализация 17](#_Toc488143196)

[5. Обработка исключений в WPF 17](#_Toc488143197)

[6. ДЗ 18](#_Toc488143198)

# Связывание данных. Триггеры. Обработка исключений. ListView. Виртуализация.

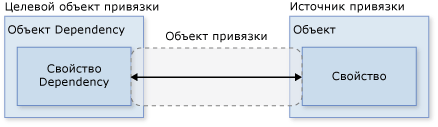
1. Связывание данных

В WPF термин «данные» применяется для описания произвольных объектов .NET. Данными могут называться коллекция объектов, XML файл, веб-сервис, таблица базы данных или даже диалоговые элементы WPF, например Button.

Следовательно, связывание данных представляет собой связывание произвольных объектов .NET. Классическим сценарием является визуальное представление в ListBox или DataGrid элементов XML файла, таблицы базы данных или коллекции в памяти. При использовании связывания данных, вместо поэлементного добавления ListBoxItem в ListBox достаточно указать источник данных для ListBox и формат их отображения.

* 1. Основные понятия связывания данных

Вне зависимости от того, какие элементы связываются, и какой источник данных используется, каждая привязка всегда соответствует следующей модели:



Связывание данных - это технология, которая реализует связи между [целевым](https://msdn.microsoft.com/ru-ru/library/gtmt#binding_target) объектом и [источником привязки](https://msdn.microsoft.com/ru-ru/library/gtmt#binding_source), а также поддерживает синхронизацию данных.

Целевой объект реализует связывание с определенным свойством объекта-источника. При изменение свойства объекта-источника происходит изменение свойства целевого объекта. Связывание достаточно объявить однократно и синхронизация данных между связанными объектами будет выполняться в течение всего времени работы приложения.

Обязательные компоненты связывания данных(привязки):

* [целевой объект привязки](https://msdn.microsoft.com/ru-ru/library/gtmt#binding_target);
* свойство целевого объекта,
* [источник привязки](https://msdn.microsoft.com/ru-ru/library/gtmt#binding_source);
* путь к значению используемого [источника привязки](https://msdn.microsoft.com/ru-ru/library/gtmt#binding_source).

Пример, демонстрирующий связывание значения TextBlock со свойством Text у TextBox.

TextBlock автоматически изменяет свое значение при вводе текста в TextBox. Без возможностей data binding потребовалось бы реализовывать обработчик событий для событий TextBox и затем изменять значение TextBlock при каждом изменении вводимого текста.

MainWindow.xaml

<Window x:Class="BindingSample.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:BindingSample"

mc:Ignorable="d"

Title="MainWindow" Height="200" Width="200">

<StackPanel Margin="10">

<TextBox Name="txtValue" />

<WrapPanel Margin="0,10">

<TextBlock Text="Текст: " FontWeight="Bold" />

<TextBlock Text="{Binding ElementName=txtValue, Path=Text }" />

</WrapPanel>

</StackPanel>

</Window>

Аналогично, можно установить привязку в файле отделенного кода:

MainWindow.xaml

<Window x:Class="BindingSample.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:BindingSample"

mc:Ignorable="d"

Title="MainWindow" Height="200" Width="200">

<StackPanel Margin="10">

<TextBox Name="txtValue" />

<WrapPanel Margin="0,10">

<TextBlock Text="Текст: " FontWeight="Bold" />

<TextBlock x:Name="mirrorTextBlock"/>

</WrapPanel>

</StackPanel>

</Window>

MainWindow.xaml.cs

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

namespace BindingSample

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

Binding binding = new Binding();

binding.ElementName = "txtValue"; // элемент-источник

binding.Path = new PropertyPath("Text"); // свойство элемента-источника

mirrorTextBlock.SetBinding(TextBlock.TextProperty, binding); // установка привязки для элемента-приемника

}

}

}

* 1. Синтаксис связывания данных

Для связывания данных используется Binding расширение XAML, которое позволяет описывать взаимодействие целевого объекта и источника.

Синтаксис выражения определения привязки:

{Binding «ElementName»= «Имя\_объекта-источника», Path=«Свойство\_объекта-источника», Mode=«Mode»}

Свойство Mode объекта Binding позволяет настраивать направление потока данных между объектом-источником и целевым объектом. Свойство может принимать следующие значения:

* OneWay. Используется, если связанное свойство изменяет значения в пользовательском интерфейсе.
* OneWayToSource. Противоположность OneWay. Изменения значения в пользовательском интерфейсе изменяют связанное свойство.
* OneTime. Аналогично поведению OneWay, за исключением того, что изменение в пользовательском интерфейсе происходят один раз.
* TwoWay. Комбинация OneWay и OneWayToSource. Связанное свойство изменяет пользовательский интерфейс и изменения в пользовательском интерфейсе модифицируют связанное свойство.
* Default: по умолчанию (если меняется свойство TextBox.Text, то имеет значение TwoWay, в остальных случаях OneWay).
  1. Форматирование значений привязки и конвертеры значений

Часто возникают ситуации, когда необходимо изменить вид привязанных значений при их отображении в интерфейсе. Например, по-разному отображать положительные и отрицательные значения или отобразить размер файла в байтах, килобайтах и мегабайтах в зависимости от его размера.

Для таких случаев существует форматирование значений и конвертеры значений.

Пример форматирования значений:

MainWindow.xaml

<Window x:Class="StringFormat.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:StringFormat"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Window.Resources>

<local:Employee x:Key="Employee" Name="Петя" Age="30" Salary="25000" />

</Window.Resources>

<Grid>

<TextBlock Text="{Binding StringFormat=Зарплата составляет {0} рублей, Source={StaticResource Employee}, Path=Salary}" />

</Grid>

</Window>

MainWindow.xaml.cs

using System.Windows;

namespace StringFormat

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

}

public class Employee

{

public string Name { get; set; }

public string Age { get; set; }

public int Salary { get; set; }

}

}

Конвертеры значений (value converter) позволяют преобразовывать значения свойств объекта-источника в произвольные значения целевого объекта. При этом, подобное преобразование выполняется одновременно с применением связывания данных.

Чаще всего конвертеры значений используются для согласования типов данных значений свойств объекта-источника и целевого объекта.

Пример конвертера значений, который преобразовывает строковые значения к типу Boolean и наоборот:

MainWindow.xaml

<Window x:Class="CheckBoxValueConverter.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:CheckBoxValueConverter"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Window.Resources>

<local:YesNoToBooleanConverter x:Key="YesNoToBooleanConverter" />

</Window.Resources>

<StackPanel Margin="10">

<TextBox Name="txtValue" />

<WrapPanel Margin="0,10">

<TextBlock Text="Текущее значение: " />

<TextBlock Text="{Binding ElementName=txtValue, Path=Text, Converter={StaticResource YesNoToBooleanConverter}}"></TextBlock>

</WrapPanel>

<CheckBox IsChecked="{Binding ElementName=txtValue, Path=Text, Converter={StaticResource YesNoToBooleanConverter}}" Content="Yes" />

</StackPanel>

</Window>

MainWindow.xaml.cs

using System;

using System.Windows;

using System.Windows.Data;

namespace CheckBoxValueConverter

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

}

public class YesNoToBooleanConverter : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)

{

switch(value.ToString().ToLower())

{

case "yes":

case "true":

return true;

case "no":

case "false":

return false;

}

return false;

}

public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)

{

if(value is bool)

{

if((bool)value == true)

return "yes";

else

return "no";

}

return "no";

}

}

}

Метод Convert позволяет преобразовать строку к типу данных boolean, метод ConvertBack выполняет обратное преобразование.

* 1. Использование DataContext

По-умолчанию, свойство DataContext является объектом-источником для привязки, если не определен иной объект-источник. Это свойство определено в классе FrameworkElement, от которого наследуется большинство элементов интерфейса, включая Window.

При запуске приложения значение DataContext не определено, но т.к. свойство DataContext присутствует в классе Window, то возможно присвоить значение DataContext в классе Window и использовать его, в дальнейшем, в других элементах, входящих в Window.

MainWindow.xaml

<Window x:Class="DataContextWPF.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:DataContextWPF"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<StackPanel Margin="15">

<WrapPanel>

<TextBlock Text="Заголовок: " />

<TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" Width="150" />

</WrapPanel>

<WrapPanel Margin="0,10,0,0">

<TextBlock Text="Размеры: " />

<TextBox Text="{Binding Width}" Width="50" />

<TextBlock Text=" x " />

<TextBox Text="{Binding Height}" Width="50" />

</WrapPanel>

</StackPanel>

</Window>

MainWindow.xaml.cs

using System.Windows;

namespace DataContextWPF

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

this.DataContext = this;

}

}

}

В файле отделенного кода происходит присвоение this.DataContext = this, т.е. сам объект Window выступает в качестве DataContext.

Использование DataContext позволяет избежать необходимости указывать множество объектов-источников для связывания.

* 1. Обновление связывания данных. UpdateSourceTrigger.

Одностороннее связывание данных объекта-источника и целевого объекта практически мгновенно изменяет свойство целевого объекта. Но при использовании TwoWay или OneWayToSource связывания в случае с текстовыми полями (как в примере выше), возможны различные стратегии обновления объекта-источника. Например, объект источник может изменяться после ввода каждого символа с клавиатуры или по окончании ввода. В предыдущем примере было видно, что изменения в TextBox не сразу отправляются объекту-источнику. Объект-источник изменялся только после потери фокуса TextBox. Такое поведение контролируется с помощью свойства UpdateSourceTrigger.

UpdateSourceTrigger в качестве значения принимает одно из значений одноименного перечисления UpdateSourceTrigger:

* PropertyChanged. Свойство в объекте-источнике изменяется всякий раз когда изменяется значение свойства целевого объекта.
* LostFocus. После потери фокуса целевым объектом обновляется объект источник привязки.
* Explicit. Объект источник обновляется только после вызова метода BindingExpression.UpdateSource().
* Default. Значение по умолчанию. Значение по умолчанию для большинства свойств зависимостей — PropertyChanged, а свойство Text имеет значение по умолчанию LostFocus.

Изменим предыдущий пример так, чтобы заголовок формы изменялся явно, по нажатию кнопки, ширина формы изменялась при потере фокуса, а высота – сразу после изменения значения.

MainWindow.xaml

<Window x:Class="UpdateSource.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:UpdateSource"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<StackPanel Margin="15">

<WrapPanel>

<TextBlock Text="Заголовок: " />

<TextBox Name="txtWindowTitle" Text="{Binding Title, UpdateSourceTrigger=Explicit}" Width="150" />

<Button Name="btnUpdateSource" Click="btnUpdateSource\_Click" Margin="5,0" Padding="5,0">Изменить</Button>

</WrapPanel>

<WrapPanel Margin="0,10,0,0">

<TextBlock Text="Размеры: " />

<TextBox Text="{Binding Width, UpdateSourceTrigger=LostFocus}" Width="50" />

<TextBlock Text=" x " />

<TextBox Text="{Binding Height, UpdateSourceTrigger=PropertyChanged}" Width="50" />

</WrapPanel>

</StackPanel>

</Window>

MainWindow.xaml.cs

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

namespace UpdateSource

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

this.DataContext = this;

}

private void btnUpdateSource\_Click(object sender, RoutedEventArgs e)

{

BindingExpression binding = txtWindowTitle.GetBindingExpression(TextBox.TextProperty);

binding.UpdateSource();

}

}

}

* 1. Реакция на изменения

Существует два сценария обработки изменений в приложении: обработка изменений в интерфейсе приложения и обработка изменений в связанных данных. Как обрабатывать подобные изменения, зависит от того какое приложение вы реализуете и что хотите получить в результате.

В WPF целям обработки изменений в связанных данных служит обобщенная коллекция ObservableCollection. Объекты в данной коллекции могут быть добавлены, удалены или изменены. При этом, когда объекты добавляются или удаляются из коллекции автоматически обновляется интерфейс приложения. Такой эффект достигается благодаря тому, что при связывании интерфейса с указанной коллекцией, WPF автоматически добавляет обработчик события CollectionChanged для событий коллекций.

Интерфейс [INotifyPropertyChanged](http://msdn.microsoft.com/en-us/library/system.componentmodel.inotifypropertychanged.aspx) используется для уведомления объектов пользовательского интерфейса об изменениях свойств объектов связанных данных. Классы, реализующие данный интерфейс генерируют события [PropertyChanged](http://msdn.microsoft.com/en-us/library/system.componentmodel.inotifypropertychanged.propertychanged.aspx) каждый раз, когда меняются значения свойства объектов связанных данных. Такое поведение позволяет привязкам данных отслеживать состояния объектов и обновлять данные пользовательского интерфейса при изменении значения связанных свойства.

MainWindow.xaml

<Window x:Class="Respond.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Respond"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<DockPanel Margin="10">

<StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">

<Button Name="btnAddUser" Click="btnAddUser\_Click">Добавить</Button>

<Button Name="btnChangeUser" Click="btnChangeUser\_Click" Margin="0,5">Изменить</Button>

<Button Name="btnDeleteUser" Click="btnDeleteUser\_Click">Удалить</Button>

</StackPanel>

<ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>

</DockPanel>

</Window>

MainWindow.xaml.cs

using System.Collections.ObjectModel;

using System.ComponentModel;

using System.Windows;

namespace Respond

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

private ObservableCollection<User> users = new ObservableCollection<User>();

public MainWindow()

{

InitializeComponent();

users.Add(new User() { Name = "Петя" });

users.Add(new User() { Name = "Коля" });

lbUsers.ItemsSource = users;

}

private void btnAddUser\_Click(object sender, RoutedEventArgs e)

{

users.Add(new User() { Name = "Вася" });

}

private void btnChangeUser\_Click(object sender, RoutedEventArgs e)

{

if (lbUsers.SelectedItem != null)

(lbUsers.SelectedItem as User).Name = "Иван";

}

private void btnDeleteUser\_Click(object sender, RoutedEventArgs e)

{

if (lbUsers.SelectedItem != null)

users.Remove(lbUsers.SelectedItem as User);

}

}

public class User : INotifyPropertyChanged

{

private string name;

public string Name

{

get { return this.name; }

set

{

if (this.name != value)

{

this.name = value;

this.NotifyPropertyChanged("Name");

}

}

}

public event PropertyChangedEventHandler PropertyChanged;

public void NotifyPropertyChanged(string propName)

{

if (this.PropertyChanged != null)

this.PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

}

}

* 1. Отладка связывания данных

Ошибки в связывании данных довольно сложно выявить, поскольку связывание данных выполняется во время исполнения программы и не вызывает исключений. Ошибки в связывании могут иметь разные причины, но наиболее распространена попытка связывания с несуществующим свойством.

MainWindow.xaml

<Window x:Class="BindingDebug.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:BindingDebug"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Grid Margin="10" Name="pnlMain">

<TextBlock Text="{Binding NonExistingProperty, ElementName=pnlMain}" />

</Grid>

</Window>

В первую очередь ошибку связывания можно обнаружить в VisualStudio в окне Output.

Output:

System.Windows.Data Error: 40 : BindingExpression path error: 'NonExistingProperty' property not found on 'object' ''Grid' (Name='pnlMain')'. BindingExpression:Path=NonExistingProperty; DataItem='Grid' (Name='pnlMain'); target element is 'TextBlock' (Name=''); target property is 'Text' (type 'String')

Это сообщение говорит, что была попытка использовать свойство NonExistingProperty объекта pnlMain типа Grid.

Еще один пример ошибки связывания:

MainWindow.xaml

<Window x:Class="TraceLevel.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:TraceLevel"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Grid Margin="10">

<TextBlock Text="{Binding Title}" />

</Grid>

</Window>

В данном примере происходит связывание со свойством Title, но не указано к какому объекту оно относится. WPF будет пытаться использовать DataContext для получения значения, но в данном примере DataContext не инициализирован. В результате WPF не обнаружит никакой ошибки и окно Output не будет содержать сообщений об ошибке.

Если данное поведение программы не совпадает с ожиданиями разработчика, то необходимо повысить TraceLevel у объекта PresentationTraceSources из пространства имен System.Diagnostics.

В результате в окне Output будут отображаться все подробности выполнения связывания данных:

System.Windows.Data Warning: 67 : BindingExpression (hash=51220585): Resolving source

System.Windows.Data Warning: 70 : BindingExpression (hash=51220585): Found data context element: TextBlock (hash=28990061) (OK)

System.Windows.Data Warning: 71 : BindingExpression (hash=51220585): DataContext is null

System.Windows.Data Warning: 67 : BindingExpression (hash=51220585): Resolving source

System.Windows.Data Warning: 70 : BindingExpression (hash=51220585): Found data context element: TextBlock (hash=28990061) (OK)

System.Windows.Data Warning: 71 : BindingExpression (hash=51220585): DataContext is null

System.Windows.Data Warning: 67 : BindingExpression (hash=51220585): Resolving source (last chance)

System.Windows.Data Warning: 70 : BindingExpression (hash=51220585): Found data context element: TextBlock (hash=28990061) (OK)

System.Windows.Data Warning: 78 : BindingExpression (hash=51220585): Activate with root item <null>

System.Windows.Data Warning: 106 : BindingExpression (hash=51220585): Item at level 0 is null - no accessor

'TraceLevel.vshost.exe' (CLR v4.0.30319: TraceLevel.vshost.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\PresentationFramework-SystemXml\v4.0\_4.0.0.0\_\_b77a5c561934e089\PresentationFramework-SystemXml.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.

System.Windows.Data Warning: 80 : BindingExpression (hash=51220585): TransferValue - got raw value {DependencyProperty.UnsetValue}

System.Windows.Data Warning: 88 : BindingExpression (hash=51220585): TransferValue - using fallback/default value ''

System.Windows.Data Warning: 89 : BindingExpression (hash=51220585): TransferValue - using final value ''

Просматривая текст в окне Output можно увидеть все действия, выполняющиеся в попытке найти подходящее значение для TextBlock: несколько сообщений DataContext is null и в конце сообщение, что будет использовано значение по умолчанию, пустая строка.

Другой способ обнаружения ошибок связывания состоит в использовании отладчика и файла отделенного кода. В файле отделенного кода необходимо реализовать «искусственный» конвертер данных, цель которого состоит в том, чтобы диагностировать

успешность связывания данных.

MainWindow.xaml

<Window x:Class="ValueConverter.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:ValueConverter"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525" Name="wnd">

<Window.Resources>

<local:DebugDummyConverter x:Key="DebugDummyConverter" />

</Window.Resources>

<Grid Margin="10">

<TextBlock Text="{Binding Title, ElementName=wnd, Converter={StaticResource DebugDummyConverter}}" />

</Grid>

</Window>

MainWindow.xaml.cs

using System;

using System.Diagnostics;

using System.Windows;

using System.Windows.Data;

namespace ValueConverter

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

}

public class DebugDummyConverter : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)

{

Debugger.Break();

return value;

}

public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)

{

Debugger.Break();

return value;

}

}

}

Если отладчик не остановиться в методах Convert или ConvertBack, это будет означать, что конвертер не используется. Это, обычно, говорит об ошибке связывания данных.

1. Триггеры

С помощью стилей происходит присваивание статических значений свойствам объектов. Триггеры, в свою очередь, позволяют изменять значения определенных свойств в зависимости от заданных условий. Они позволяют выполнять те действия, которые обычно реализуются в файле отделенного кода, используя только разметку xaml.

Существует 4 категории триггеров:

* Триггеры свойств. Отслеживают изменения определенного свойства у родительского контрола и в случае, если значение данного свойства совпадает с заданным значением, происходит изменение значения другого свойства.
* Триггеры данных: Вызываются в ответ на изменения значений любых свойств (не только свойств зависимостей). Используют выражения для связывания с обычными свойствами, изменения которых и отслеживают. Позволяют связывать триггер со свойствами другого контрола.
* Триггеры событий: вызываются в ответ на генерацию событий.
* Мультитриггеры: вызываются при выполнении ряда условий.
  1. Триггеры свойств

Триггеры свойств задаются с помощью объекта Trigger. Они следят за значениями свойств и когда эти значения оказываются равными заданным величинам, свойства могут быть изменены. Изменения свойств осуществляются с помощью объекта Setter.

MainWindow.xaml

<Window x:Class="WpfTrigger.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:WpfTrigger"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Grid>

<TextBlock Text="Триггер" FontSize="28" HorizontalAlignment="Center" VerticalAlignment="Center">

<TextBlock.Style>

<Style TargetType="TextBlock">

<Setter Property="Foreground" Value="Blue"></Setter>

<Style.Triggers>

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Foreground" Value="Red" />

<Setter Property="TextDecorations" Value="Underline" />

</Trigger>

</Style.Triggers>

</Style>

</TextBlock.Style>

</TextBlock>

</Grid>

В данном примере, изначально, с помощью стиля, устанавливается синий цвет текста. С помощью триггера происходит анализ свойства IsMouseOver и, когда значение свойства становиться равным true, происходит изменение цвета текста и добавляется оформление текста в виде подчеркивания.

* 1. Триггеры данных

DataTrigger отслеживает изменение свойств, которые необязательно должны представлять свойства зависимостей. Для соединения с отслеживаемыми свойствами триггеры данных

используют выражения привязки.

MainWindow.xaml

<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">

<CheckBox Name="cbSample" Content="Вы знаете C#?" />

<TextBlock HorizontalAlignment="Center" Margin="0,20,0,0" FontSize="48">

<TextBlock.Style>

<Style TargetType="TextBlock">

<Setter Property="Text" Value="Нет" />

<Setter Property="Foreground" Value="Red" />

<Style.Triggers>

<DataTrigger Binding="{Binding ElementName=cbSample, Path=IsChecked}" Value="True">

<Setter Property="Text" Value="Да" />

<Setter Property="Foreground" Value="Green" />

</DataTrigger>

</Style.Triggers>

</Style>

</TextBlock.Style>

</TextBlock>

* 1. Триггеры событий

Триггеры событий, <EventTrigger>, чаще всего используются для определения анимации. Триггер событий реагирует на определенные события также, как обработчик событий.

MainWindow.xaml

<Window x:Class="EventTrigger.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:EventTrigger"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Grid>

<TextBlock Name="lbl" Text="EventTrigger" FontSize="18" HorizontalAlignment="Center" VerticalAlignment="Center">

<TextBlock.Style>

<Style TargetType="TextBlock">

<Style.Triggers>

<EventTrigger RoutedEvent="MouseEnter">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimation Duration="0:0:0.300" Storyboard.TargetProperty="FontSize" To="28" />

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

<EventTrigger RoutedEvent="MouseLeave">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimation Duration="0:0:0.800" Storyboard.TargetProperty="FontSize" To="18" />

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

</Style.Triggers>

</Style>

</TextBlock.Style>

</TextBlock>

</Grid>

</Window>

Триггер событий подписан на два события MouseEnter и MouseLeave. Если курсор мыши оказывается над областью занятой текстом, то происходит постепенное увеличение размера символов до 28 пикселей. Когда курсор мыши покидает область текста, то происходит плавное уменьшение размеров символов до 18.

1. ListView

Контрол ListView, который является наследником ListBox, выглядит и работает также, как и ListBox, за исключением того, что он использует по умолчанию ExtendedSelectionMode. Но, кроме этого, ListView содержит свойство View, которое дает значительно больше возможностей по построению многофункциональных списковых представлений, чем ItemsPanel.

Свойство View относиться к абстрактному типу ViewBase. В WPF реализован единственный класс данного типа, GridView. По умолчанию, его представление соответствует окну Details в Проводнике.

Свойство View принимает в качестве значения объект GridView, который управляет отображением данных.

GridView содержит свойство Columns, которое включает в себя коллекцию определений столбцов – GridViewColumn. GridViewColumn с помощью свойства Header определяет название столбца.

Строки ListView описываются, как и в ListBox, в виде обычного списка. Возможность отображения разных данных в каждой отдельной колонке достигается благодаря свойству

DisplayMemberBinding класса GridViewColumn. Идея состоит в том, что ListView содержит объекты с множеством свойств для каждой строки и значение для каждой колонки определяется свойствами отдельного объекта.

GridView поддерживает следующие возможности:

* Изменение порядка колонок с помощью Drag&Drop.
* Изменение размера колонок путем перемещения сепаратора.
* Автоматическое изменение размеров колонок, чтобы вместить содержание колонки, с помощью двойного клика по сепаратору.

ListView не поддерживает автоматическую сортировку с помощью двойного клика по заголовку колонки.

MainWindow.xaml

<Window x:Class="GridView.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:GridView"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Grid>

<ListView Margin="10" Name="lvEmployee">

<ListView.View>

<GridView>

<GridViewColumn Header="Имя" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Возраст" Width="50" DisplayMemberBinding="{Binding Age}" />

<GridViewColumn Header="Зарплата" Width="150" DisplayMemberBinding="{Binding Salary}" />

</GridView>

</ListView.View>

</ListView>

</Grid>

MainWindow.xaml.cs

using System.Collections.Generic;

using System.Windows;

namespace GridView

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

List<Employee> items = new List<Employee>();

items.Add(new Employee() { Name = "Петя", Age = 42, Salary = 25000 });

items.Add(new Employee() { Name = "Коля", Age = 39, Salary = 45000 });

items.Add(new Employee() { Name = "Иван", Age = 7, Salary = 33000 });

lvEmployee.ItemsSource = items;

}

}

public class Employee

{

public string Name { get; set; }

public int Age { get; set; }

public int Salary { get; set; }

}

}

Использование свойства DisplayMemberBinding не позволяет производить форматирование отображаемых свойств. Но это ограничение можно обойти, определив свойство CellTemplate. C его помощью достигается полный контроль над содержимым ячейки ListView.

MainWindow.xaml

<Window x:Class="CellTemplate.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:CellTemplate"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Grid>

<ListView Margin="10" Name="lvEmployee">

<ListView.View>

<GridView>

<GridViewColumn Header="Имя" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Возраст" Width="50" DisplayMemberBinding="{Binding Age}" />

<GridViewColumn Header="Зарплата" Width="150">

<GridViewColumn.CellTemplate>

<DataTemplate>

<TextBlock Text="{Binding Salary}" Foreground="Blue" FontWeight="Bold" />

</DataTemplate>

</GridViewColumn.CellTemplate>

</GridViewColumn>

</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>

В примере выше для колонки «Зарплата» изменен цвет символов и увеличена их толщина.

1. Виртуализация

Панели, являющиеся наследниками абстрактного класса System.Windows.Controls.VirtualizingPanel, позволяют реализовать интересный механизм отображения для таких контролов как ListBox, ListView, DataGrid. VirtualizingStackPanel, которая действует как StackPanel, позволяет загружать в память только те элементы ListBox, ListView, DataGrid, которые отображаются на экране. Тем самым повышается производительность приложения. Данный механизм работает только при использовании связывания данных. Благодаря данному механизму виртуализации, VirtualizingStackPanel является лучшим вариантом среди панелей при связывании больших объемов данных. ListBox использует данный вид панелей по умолчанию.

Для включения виртуализации для элементов, производных от ItemsControl, или для уже существующих элементов управления, которые используют StackPanel (например, ComboBox), надо установить свойство ItemsPanel для класса VirtualizingStackPanel и присвоить свойству IsVirtualizing значение true. Например:

<ComboBox VirtualizingStackPanel.IsVirtualizing="True">

<ComboBox.ItemsPanel>

<ItemsPanelTemplate>

<VirtualizingStackPanel />

</ItemsPanelTemplate>

</ComboBox.ItemsPanel>

</ComboBox>

Многие контролы используют VirtualizingStackPanel как свою ItemsPanel, по умолчанию, для улучшения производительности. В WPF 4, такие панели поддерживают новый режим, который еще более повышает производительность при скролинге. Данный режим требует явного включения. Для этого нужно присвоить свойству VirtualizingStackPanel.VirtualizationMode значение Recycling. В результате, панель будет повторно использовать контейнеры, которые создаются при прокрутке элементов списка на экране, вместо того чтобы создавать новые контейнеры для каждого элемента.

1. Обработка исключений в WPF

Помимо использования конструкций try …catch при вызове функций, которые могут формировать исключения, WPF позволяет перехватывать необработанные исключения глобально, в рамках приложения. Данный механизм осуществляется путем регистрации обработчика события DispatcherUnhandledException в классе Application.

App.xaml

<Application x:Class="WpfException.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:WpfException"

DispatcherUnhandledException="Application\_DispatcherUnhandledException"

StartupUri="MainWindow.xaml">

<Application.Resources>

</Application.Resources>

</Application>

App.xaml.cs

using System.Windows;

namespace WpfException

{

/// <summary>

/// Interaction logic for App.xaml

/// </summary>

public partial class App : Application

{

private void Application\_DispatcherUnhandledException(object sender, System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e)

{

MessageBox.Show("Необработанное исключение: " + e.Exception.Message, "Exception", MessageBoxButton.OK, MessageBoxImage.Warning);

e.Handled = true;

}

}

}

MainWindow.xaml

<Window x:Class="WpfException.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:WpfException"

mc:Ignorable="d"

Title="MainWindow" Height="350" Width="525">

<Grid>

<Button x:Name="button" Content="Button" HorizontalAlignment="Center" VerticalAlignment="Center" Width="75" Click="button\_Click"/>

</Grid>

</Window>

MainWindow.xaml.cs

using System.Windows;

namespace WpfException

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private void button\_Click(object sender, RoutedEventArgs e)

{

string s = null;

int length = s.Length;

}

}

}

1. ДЗ

Измените WPF приложение для ведения списка сотрудников компании, из урока №5, **используя связывание данных, ListView, ObservableCollection и INotifyPropertyChanged**.  
1. Создайте сущности Employee и Department и заполните списки сущностей начальными данными.  
2. Для списка сотрудников и списка департаментов предусмотрите визуализацию (отображение). Это можно сделать, например, с использованием ComboBox или ListView.  
3. Предусмотрите возможность редактирования сотрудников и департаментов. Должна быть возможность изменить департамент у сотрудника. Список департаментов для выбора, можно выводить в ComboBox, это все можно выводить на дополнительной форме.  
4. Предусмотрите возможность создания новых сотрудников и департаментов. Реализуйте данную возможность либо на форме редактирования, либо сделайте новую форму.