

# Работа с данными

## Привязка данных и контекст данных

Большую роль при работе с данными играет механизм привязки. Ранее в одной из прошлых тем рассматривалась привязка элементов и их свойств. В этой же теме сделаем больший упор на привязку данных.

Для создания привязки применяется элемент **Binding** и его свойства:

- **ElementName**: имя элемента, к которому идет привязка. Если мы говорим о привязке данных, то данное свойство задействуется редко за исключением тех случаев, когда данные определены в виде свойства в определенном элементе управления
- **Path**: ссылка на свойство объекта, к которому идет привязка
- **Source**: ссылка на источник данных, который не является элементом управления

Свойства элемента **Binding** помогают установить источник привязки. Для установки источника или контекста данных в элементах управления WPF предусмотрено свойство **DataContext**. Рассмотрим на примерах их использование.

Пусть у нас в проекте определен следующий класс **Phone**:

```
public class Phone
{
    public string Name { get; set; }
    public Company Company { get; set; }
    public decimal Price { get; set; }
}

public class Company
{
    public string Title { get; set; }
}
```

Это сложный класс, который включает кроме простых данных типа `string` и `decimal` также и сложный объект `Company`.

Определим в классе окна `MainWindow` свойство, которое будет представлять объект `Phone`:

```

public partial class MainWindow : Window
{
    public Phone MyPhone { get; set; }
    public MainWindow()
    {
        InitializeComponent();

        MyPhone = new Phone
        {
            Name = "Lumia 630",
            Company = new Company { Title = "Microsoft" },
            Price = 1000
        };
        this.DataContext = MyPhone;
    }
}

```

Здесь установлено свойство DataContext класса MainWindow, после чего мы сможем получить значения из MyPhone в любом элементе в пределах MainWindow:

```

<Window x:Class="DataApp.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:DataApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300">
    <StackPanel>
        <TextBlock Text="{Binding Path=Name}" />
        <TextBlock DataContext="{Binding Path=Company}" Text="{Binding Path=Title}" />
    </StackPanel>
</Window>

```

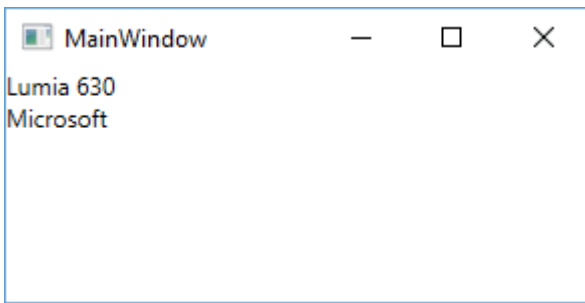
Причем контекст данных переходит от корневых элементов к вложенным вниз по логическому дереву. Так, мы установили в качестве контекста для всего окна объект MyPhone. Однако элементы внутри окна могут конкретизировать контекст, взять какую-то его часть:

```

<TextBlock DataContext="{Binding Path=Company}" Text="{Binding Path=Title}" />

```

Данный текстовый блок устанавливает в качестве контекста объект Company из общего контекста MyPhone:



В тоже время нам необязательно конкретизировать контекст для текстового блока, вместо этого мы могли бы с помощью нотации точки обратиться к вложенным свойствам:

```
<TextBlock Text="{Binding Path=Company.Title}" />
```

При этом надо учитывать, что когда мы определяем простое свойство объекта в коде *c#*, то установить в качестве контекста данных мы можем его только там же в коде *C#*, как, например, выше контекст данных устанавливается в конструкторе окна. Однако если мы вместо простого свойства определим свойство зависимостей, тогда мы сможем устанавливать контекст данных и в коде *xml*:

```

public partial class MainWindow : Window
{
    public static readonly DependencyProperty PhoneProperty;

    public Phone Phone
    {
        get { return (Phone)GetValue(PhoneProperty); }
        set { SetValue(PhoneProperty, value); }
    }

    static MainWindow()
    {
        PhoneProperty = DependencyProperty.Register(
            "Phone",
            typeof(Phone),
            typeof(MainWindow));
    }

    public MainWindow()
    {
        InitializeComponent();

        Phone = new Phone
        {
            Name = "Lumia 630",
            Company = new Company { Title = "Microsoft" },
            Price = 1000
        };
    }
}

```

В данном случае контекст данных уже не устанавливается, а вместо обычного свойства определено свойство зависимостей. Тогда в коде xaml мы можем обратиться к этому свойству следующим образом:

```

<Window x:Class="DataApp.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:DataApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="250" Width="300" Name="mainWindow">
    <StackPanel>
        <TextBlock Text="{Binding ElementName=mainWindow, Path=Phone.Name}" />
        <TextBlock Text="{Binding ElementName=mainWindow, Path=Phone.Company.Title}" />
    </StackPanel>
</Window>

```

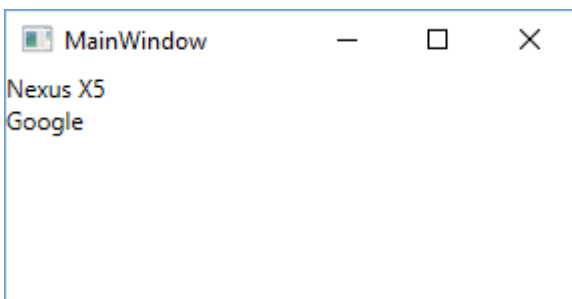
Для ссылки на свойство необходимо установить имя окна: Name="mainWindow". Также мы могли бы сделать то же самое, используя контекст данных:

```
<StackPanel DataContext="{Binding ElementName=mainWindow, Path=Phone}">
    <TextBlock Text="{Binding Path=Name}" />
    <TextBlock DataContext="{Binding Path=Company}" Text="{Binding Path=Title}" />
</StackPanel>
```

## Подключение к ресурсам

Также в качестве контекста данных можно установить какой-нибудь ресурс. Например:

```
<Window x:Class="DataApp.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:DataApp"
        mc:Ignorable="d"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="MainWindow" Height="250" Width="300" Name="mainWindow">
    <Window.Resources>
        <local:Company x:Key="googleCompany" Title="Google" />
        <local:Phone x:Key="nexusPhone" Name="Nexus X5" Price="25000" Company="{StaticResource g
    </Window.Resources>
    <StackPanel DataContext="{StaticResource nexusPhone}">
        <TextBlock Text="{Binding Path=Name}" />
        <TextBlock DataContext="{Binding Path=Company}" Text="{Binding Path=Title}" />
    </StackPanel>
</Window>
```



## Работа с коллекциями данных. ObservableCollection

В прошлых темах были рассмотрены примеры привязки отдельных объектов к элементам интерфейса. Но, как правило, приложения оперируют не одиночными данными, а большими наборами, коллекциями объектов. Для работы непосредственно с наборами данных в WPF определены различные элементы управления списками, такие как `ListBox`, `ListView`, `DataGrid`, `TreeView`, `ComboBox`.

Их отличительной особенностью является то, что они наследуются от базового класса **ItemsControl** и поэтому наследуют ряд общей функциональности для работы с данными. Прежде всего можно выделить свойства:

- **Items**: устанавливает набор объектов внутри элемента
- **ItemsSource**: ссылка на источник данных
- **ItemStringFormat**: формат, который будет использоваться для форматирования строк, например, при переводе в строку числовых значений
- **ItemContainerStyle**: стиль, который устанавливается для контейнера каждого элемента (например, для `ListBoxItem` или `ComboBoxItem`)
- **ItemTemplate**: представляет шаблон данных, который используется для отображения элементов
- **ItemsPanel**: панель, которая используется для отображения данных. Как правило, применяется `VirtualizingStackPanel`
- **DisplayMemberPath**: свойство, которое будет использоваться для отображения в списке каждого объекта

При работе с элементами управления списками важно понимать, что эти элементы предназначены прежде всего для отображения данных, а не для хранения. В каких-то ситуациях мы, конечно, можем определять небольшие списки непосредственно внутри элемента. Например:

```
<ListBox>
  <ListBox.Items>
    <ListBoxItem>iPhone 6S Plus</ListBoxItem>
    <ListBoxItem>Nexus 6P</ListBoxItem>
    <ListBoxItem>Galaxy S7 Edge</ListBoxItem>
  </ListBox.Items>
</ListBox>
```

Но в большинстве случаев предпочтительнее использовать привязку к спискам и разделять источник данных от их представления или визуализации. Например, определим `ListBox`:

```
<ListBox x:Name="phonesList" />
```

А в коде с# создадим источник данных и установим привязку к нему:

```
public partial class MainWindow : Window
{
    List<string> phones;
    public MainWindow()
    {
        InitializeComponent();

        phones = new List<string> { "iPhone 6S Plus", "Nexus 6P", "Galaxy S7 Edge" };
        phonesList.ItemsSource = phones;
    }
}
```

## ObservableCollection

В примере выше в качестве источника данных использовался список List. Также в качестве источника мы бы могли использовать другой какой-нибудь тип набора данных - массив, объект HashSet и т.д. Но нередко в качестве источника применяется класс **ObservableCollection**, который находится в пространстве имен **System.Collections.ObjectModel**. Его преимущество заключается в том, что при любом изменении ObservableCollection может уведомлять элементы, которые применяют привязку, в результате чего обновляется не только сам объект ObservableCollection, но и привязанные к нему элементы интерфейса.

Например, рассмотрим следующую ситуацию. У нас кроме элемента ListBox есть текстовое поле и кнопка для добавления нового объекта:

```

<Window x:Class="DataApp.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:DataApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300" Name="mainWindow">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <ListBox x:Name="phonesList" />
        <StackPanel Grid.Row="1" Orientation="Horizontal" Margin="0 15 0 0" HorizontalAlignment=
            <TextBox Name="phoneTextBox" Width="190" />
            <Button Content="Сохранить" MaxWidth="70" Margin="10 0 0 0" Click="Button_Click" />
        </StackPanel>
    </Grid>
</Window>

```

В файле кода определим обработчик кнопки, в котором новый элемент добавлялся бы в источник данных:

```

public partial class MainWindow : Window
{
    List<string> phones;
    public MainWindow()
    {
        InitializeComponent();

        phones = new List<string> { "iPhone 6S Plus", "Nexus 6P", "Galaxy S7 Edge" };
        phonesList.ItemsSource = phones;
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        string phone = phoneTextBox.Text;
        // добавление нового объекта
        phones.Add(phone);
    }
}

```

По нажатию на кнопку должно произойти добавления в список phones введенной в текстовое поле строки. И мы ожидаем, что после добавления ListBox отобразит нам добавленный объект. Однако так как в качестве источника применяется List, то обновления элемента ListBox не произойдет. Поэтому заменим List на ObservableCollection:



```

public partial class MainWindow : Window
{
    ObservableCollection<string> phones;
    public MainWindow()
    {
        InitializeComponent();

        phones = new ObservableCollection<string> { "iPhone 6S Plus", "Nexus 6P", "Galaxy S7 Edge" };
        phonesList.ItemsSource = phones;
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        string phone = phoneTextBox.Text;
        // добавление нового объекта
        phones.Add(phone);
    }
}

```

И теперь у нас уже не возникнет подобной проблемы:

<https://metanit.com/sharp/wpf/pics/14.3.png>

<https://metanit.com/sharp/wpf/pics/14.4.png>

## ItemTemplate и DataTemplate

Все элементы управления списками имеют свойство **ItemTemplate**, которое позволяет задать свой шаблон отображения данных. В качестве значения оно принимает объект **DataTemplate**. В качестве самих данных, которые отображаются в DataTemplate, как правило, выступают объекты пользовательских классов, создаваемых самим разработчиком. Сам шаблон данных представляет разметку xaml, которая управляет визуализацией элемента.

Итак, добавим в проект следующий класс Phone, который будет представлять модель телефона:

```

public class Phone
{
    public int Id { get; set; }
    public string Title { get; set; } // модель телефона
    public string Company { get; set; } // производитель
    public string ImagePath { get; set; } // путь к изображению
}

```

В коде xaml определим шаблон данных для списка смартфонов:

```

<Window x:Class="DataApp.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:DataApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="250" Width="300">
    <Grid>
        <ListBox x:Name="phonesList" SelectionChanged="phonesList_SelectionChanged">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Margin="5">
                        <Image Width="100" Height="75" Source="{Binding Path=ImagePath}" />
                        <TextBlock FontSize="16" Text="{Binding Path=Title}" HorizontalAlignment
                        <TextBlock FontSize="16" Text="{Binding Path=Company}" HorizontalAlignme
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</Window>

```

Итак, здесь шаблон данных представлен следующей разметкой:

```

<DataTemplate>
    <StackPanel Margin="5">
        <Image Width="100" Height="75" Source="{Binding Path=ImagePath}" />
        <TextBlock FontSize="16" Text="{Binding Path=Title}" HorizontalAlignment="Center" />
        <TextBlock FontSize="16" Text="{Binding Path=Company}" HorizontalAlignment="Center" />
    </StackPanel>
</DataTemplate>

```

Для каждого элемента применяется привязка к определенному свойству объекта Phone. То есть фактически в визуальном плане каждый элемент будет представлять собой StackPanel с набором элементов Image и TextBlock.

Далее создадим в коде C# коллекцию смартфонов и установим ее в качестве источника данных:

```

public partial class MainWindow : Window
{
    public ObservableCollection<Phone> Phones { get; set; }
    public MainWindow()
    {
        InitializeComponent();

        Phones = new ObservableCollection<Phone>
        {
            new Phone {Id=1, ImagePath="/Images/iphone6s.jpg", Title="iPhone 6S", Company="Apple"},
            new Phone {Id=2, ImagePath="/Images/lumia950.jpg", Title="Lumia 950", Company="Microsoft"},
            new Phone {Id=3, ImagePath="/Images/nexus5x.jpg", Title="Nexus 5X", Company="Google"},
            new Phone {Id=4, ImagePath="/Images/galaxyS6.jpg", Title="Galaxy S6", Company="Samsung"}
        };
        phonesList.ItemsSource = Phones;
    }

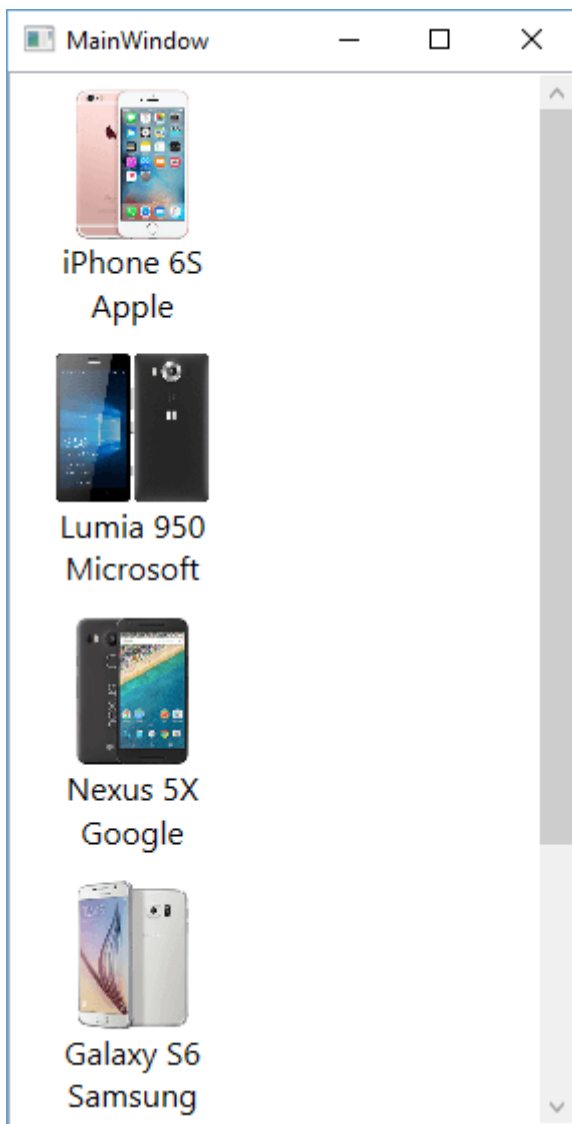
    private void phonesList_SelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        Phone p = (Phone)phonesList.SelectedItem;
        MessageBox.Show(p.Title);
    }
}

```

Здесь создается 4 объекта Phone. Для каждого объекта устанавливаются необходимые значения. Причем свойство ImagePath указывает на путь к изображению в папке Images. Предполагается, что в проект добавлена папка Images, в которой есть четыре изображения для каждой из моделей.

И также здесь определяется обработчик выбора элемента в списке. Несмотря на то, что список представляет сложные данные, мы все равно сможем обработать выбор пользователя и получить выбранный объект. И так как DataTemplate отображает объекты Phone, то выделенный элемент мы можем привести к типу Phone.

В итоге получится следующее приложение:



Одним из преимуществ шаблонов данных является то, что их можно вынести во вне, например, в ресурсы. В этом случае мы повторно сможем использовать одни и тот же шаблон данных для разных элементов в разных частях программы:

```

<Window x:Class="DataApp.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:DataApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="250" Width="300">
    <Window.Resources>
        <DataTemplate x:Key="listTemplate">
            <StackPanel Margin="5">
                <Image Width="100" Height="75" Source="{Binding Path=ImagePath}" />
                <TextBlock FontSize="16" Text="{Binding Path=Title}" HorizontalAlignment="Center" />
                <TextBlock FontSize="16" Text="{Binding Path=Company}" HorizontalAlignment="Center" />
            </StackPanel>
        </DataTemplate>
    </Window.Resources>
    <Grid>
        <ListBox x:Name="phonesList" ItemTemplate="{StaticResource listTemplate}"
            SelectionChanged="phonesList_SelectionChanged" />
    </Grid>
</Window>

```

## Триггеры данных

С помощью триггеров данных (data triggers) можно задать дополнительную логику визуализации, которая срабатывает, если свойство привязанного объекта принимает то или иное значение. Например, в случае с примером выше, допустим, мы хотим, если у отображаемого телефона свойство Company равно Microsoft, то этот элемент выделяется жирным или выделяется красным цветом. И для этого добавим в шаблон данных DataTemplate триггер данных:

```

<Window x:Class="DataApp.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:DataApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="250" Width="300">
<Window.Resources>
    <DataTemplate x:Key="listTemplate">
        <StackPanel Margin="5">
            <Border x:Name="phoneImageBorder">
                <Image Width="100" Name="phoneImage" Height="75" Source="{Binding Path=Image"
            </Border>
            <TextBlock FontSize="16" Name="phoneTitle" Text="{Binding Path=Title}" HorizontalAlignt
            <TextBlock FontSize="16" Text="{Binding Path=Company}" HorizontalAlignment="Cent
        </StackPanel>
        <DataTemplate.Triggers>
            <DataTrigger Binding="{Binding Company}" Value="Microsoft">
                <Setter TargetName="phoneTitle" Property="FontWeight" Value="Bold" />
                <Setter TargetName="phoneImageBorder" Property="BorderBrush" Value="Red" />
                <Setter TargetName="phoneImageBorder" Property="BorderThickness" Value="3" /
            </DataTrigger>
        </DataTemplate.Triggers>
    </DataTemplate>
</Window.Resources>
<Grid>
    <ListBox x:Name="phonesList" ItemTemplate="{StaticResource listTemplate}"
        SelectionChanged="phonesList_SelectionChanged" />
</Grid>
</Window>

```

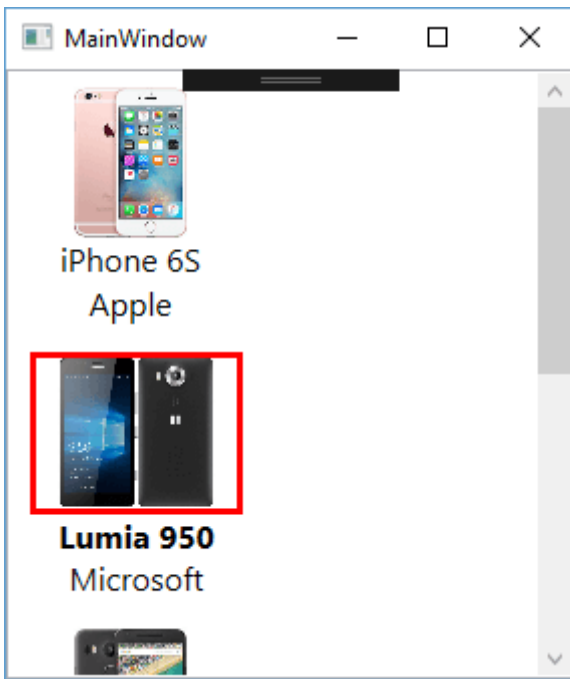
Для визуализации изображение помещено в элемент Border, который имеет имя "phoneImageBorder". С помощью триггера данных у этого элемента изменяется ширина и цвет границы. Кроме того, меняется жирность шрифта, который выводит название модели:

```

<DataTrigger Binding="{Binding Company}" Value="Microsoft">
    <Setter TargetName="phoneTitle" Property="FontWeight" Value="Bold" />
    <Setter TargetName="phoneImageBorder" Property="BorderBrush" Value="Red" />
    <Setter TargetName="phoneImageBorder" Property="BorderThickness" Value="3" />
</DataTrigger>

```

В итоге мы получим следующий визуальный эффект:



## ItemsPanel. Установка панели элементов

В примере из прошлой темы про шаблоны данных `DataTemplate` была одна проблема - по умолчанию `ListBox` размещает все элементы в один вертикальный столбик, поэтому если мы, к примеру, раздвинем окно приложения по ширине, то у нас появится много незаполненного пространства справа:

Такой дизайн сложно назвать удачным. Ведь гораздо удобнее было бы в данном случае, если бы элементы при растяжении по ширине автоматически заполняли новое пространство.

Дело в том, что такие элементы как `ListBox` и `ListView` по умолчанию инкапсулируют все элементы списка в специальную панель `VirtualizingStackPanel`, которая располагает все элементы по вертикали. Но с помощью свойства **`ItemsPanel`** можно переопределить панель элементов внутри списка.

Например, в ситуации выше было бы более эффективно использовать панель `WrapPanel`:

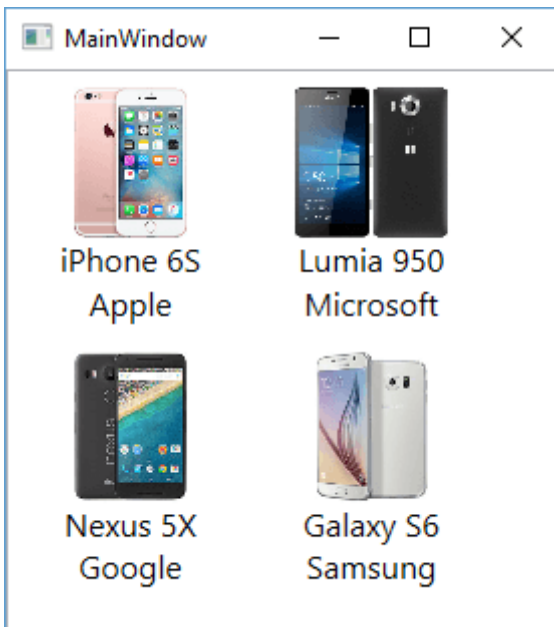
```

<Window x:Class="DataApp.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:DataApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="250" Width="300">
<Window.Resources>
    <DataTemplate x:Key="listTemplate">
        <StackPanel Margin="5">
            <Image Width="100" Height="75" Source="{Binding Path=ImagePath}" />
            <TextBlock FontSize="16" Text="{Binding Path=Title}" HorizontalAlignment="Center" />
            <TextBlock FontSize="16" Text="{Binding Path=Company}" HorizontalAlignment="Center" />
        </StackPanel>
    </DataTemplate>
</Window.Resources>
<Grid>
    <ListBox x:Name="phonesList" ItemTemplate="{StaticResource listTemplate}"
        SelectionChanged="phonesList_SelectionChanged"
        ScrollViewer.HorizontalScrollBarVisibility="Disabled">
        <ListBox.ItemsPanel>
            <ItemsPanelTemplate>
                <WrapPanel />
            </ItemsPanelTemplate>
        </ListBox.ItemsPanel>
    </ListBox>
</Grid>
</Window>

```

Для установки панели применяется элемент **ItemsPanelTemplate**, который собственно и устанавливает нужную нам панель. И также в данном случае с помощью свойства `ScrollViewer.HorizontalScrollBarVisibility="Disabled"` убирается горизонтальная прокрутка, чтобы она нам не мешала. В итоге окно приложения будет выглядеть следующим образом:





И также, как и в случае с шаблонами данных, шаблон панели мы также можем вынести во внешний ресурс и затем этот ресурс подключать:

```
<Window x:Class="DataApp.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:DataApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300">
    <Window.Resources>
        <DataTemplate x:Key="listTemplate">
            <StackPanel Margin="5">
                <Image Width="100" Height="75" Source="{Binding Path=ImagePath}" />
                <TextBlock FontSize="16" Text="{Binding Path=Title}" HorizontalAlignment="Center" />
                <TextBlock FontSize="16" Text="{Binding Path=Company}" HorizontalAlignment="Center" />
            </StackPanel>
        </DataTemplate>
        <ItemsPanelTemplate x:Key="listPanelTemplate">
            <WrapPanel />
        </ItemsPanelTemplate>
    </Window.Resources>
    <Grid>
        <ListBox x:Name="phonesList"
                ItemTemplate="{StaticResource listTemplate}"
                ItemsPanel="{StaticResource listPanelTemplate}"
                SelectionChanged="phonesList_SelectionChanged"
                ScrollViewer.HorizontalScrollBarVisibility="Disabled" />
    </Grid>
</Window>
```

# Виртуализация

Данные, которыми оперирует программа, могут насчитывать сотни и даже тысячи объектов. Возможно, нам потребуется все эти объекты выводить в списочные элементы - `ListBox`, `ListView`, `DataGrid`. Однако если мы разом загрузим все тысячи объектов в эти элементы управления, то можем столкнуться с проблемой падения производительности. И в этом случае нам надо будет воспользоваться механизмом виртуализации.

Виртуализация позволяет создавать элементу управления контейнер только для непосредственно отображаемых объектов списка. Только для них выделяется память, при этом элемент хранит схему структуры данных, чтобы при прокрутке или изменении видимых объектов соответственно изменить содержимое контейнера. То есть, если в `ListView` загружено 10000 объектов, однако в реальности элемент отображает только 10 объектов, то `ListView` создает только 10 объектов `ListViewItem`. Остальные объекты существуют для `ListView` только потенциально, в реальности они начинают использоваться только тогда, когда попадают в область видимости по мере прокрутки. В итоге приложение использует меньше памяти, работает быстрее, что повышает производительность.

По умолчанию виртуализация включена для элементов `ListView`, `ListBox` и `DataGrid`, когда они используют привязку к данным в коллекциях.

Для элемента `TreeView` виртуализацию можно включить, присвоив вложенному свойству `VirtualizingStackPanel.IsVirtualizing` значение `true`:

```
<TreeView VirtualizingStackPanel.IsVirtualizing="True" ... >
```

Если же надо использовать виртуализацию для каких-то своих элементов, производных от `ItemsControl`, или для уже существующих элементов управления, которые используют `StackPanel` (например, `ComboBox`), то в этих случаях 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>
```

Однако при наличии некоторых условий виртуализация отключается:

- если контейнеры элементов добавляются напрямую к элементу управления `ItemsControl`. Например, если объекты `ListBoxItem` добавляются к `ListBox`, то `ListBox` не будет виртуализировать эти `ListBoxItem`
- если объект `ItemsControl` содержит контейнеры элементов различных типов. Например, объект `Menu`, может содержать объекты типа `Separator` и `MenuItem`
- если для прикрепленного свойства `VirtualizingStackPanel.IsVirtualizing` установлено значение `false`
- если для прикрепленного свойства `ScrollViewer.CanContentScroll` установлено значение `false`

## Повторное использование контейнера

При заполнении объекта `ItemsControl` создается контейнер элементов для каждого отображаемого при прокрутке элемента. Когда элемент в результате прокрутки перестает отображаться, его контейнер удаляется. Повторное использование контейнера позволяет элементу управления повторно использовать существующие контейнеры элементов для различных элементов данных, чтобы контейнеры элементов не создавались и не удалялись постоянно при прокрутке. Для этого надо использовать свойство `VirtualizingStackPanel.VirtualizationMode="Recycling"`:

```
<ListBox Height="150" ItemsSource="{StaticResource data}"
        VirtualizingStackPanel.VirtualizationMode="Recycling" />
```

По умолчанию данное свойство отключено для все элементов управления списками за исключением элемента `DataGrid`. Подобная техника снижает потребление памяти и необходимость утилизировать ненужные объекты сборщиком мусора каждый раз, когда они станут не нужны.

## Кэширование

Для большей оптимизации производительности `VirtualizingStackPanel` производит кэширование объектов. С помощью свойств **`CacheLength`** и **`CacheLengthUnit`** класса `VirtualizingStackPanel` мы можем настроить кэширование видимых объектов в списках.

`CacheLengthUnit` позволяет указать тип кэша - это может быть отдельные объекты списка, страницы списка (страницы представляет набор объектов, одновременно отображаемых в списке) и пиксели (используется преимущественно для изображений).

Свойство `CacheLength` устанавливает количество единиц (объектов, страниц или пикселей), которые будут кэшироваться виртуальной панелью.

Например, по умолчанию используются следующие настройки:

```
<ListBox VirtualizingStackPanel.CacheLength="1" VirtualizingStackPanel.CacheLengthUnit="Page" ..
```

То есть `VirtualizingStackPanel` будет кэшировать одну страницу объектов до и одну страницу после тех, которые отображаются в `ListBox`. В итоге при прокрутке вверх или вниз все новые элементы будут появляться плавно и без задержек, так как они берутся из кэша `VirtualizingStackPanel`. Но мы можем изменить стандартные настройки, например, следующим образом:

```
<ListBox VirtualizingStackPanel.CacheLength="30" VirtualizingStackPanel.CacheLengthUnit="Item" .
```

В данном случае будет кэшировать 30 объектов до и после тех, которые отображаются в `ListBox`.

Также можно отдельно настроить кэширование тех объектов, которые находятся до отображаемых, и тех объектов, которые после:

```
<ListBox VirtualizingStackPanel.CacheLength="30, 50" VirtualizingStackPanel.CacheLengthUnit="Item"
```

В данном случае будет кэшировать 30 объектов до тех, которые отображаются, и 50 объектов, которые в списке после тех, которые отображаются в `ListBox`.

## Отложенная прокрутка

Отложенная прокрутка (deferred scrolling) представляет технику оптимизации, при которой обновление данных в окне элемента управления происходит только тогда, когда пользователь отпускает скрол. Чтобы реализовать отложенную прокрутку, надо установить для свойства **`IsDeferredScrollingEnabled`** значение `true`:

```
<ListBox ScrollViewer.IsDeferredScrollingEnabled="True" ... />
```

`IsDeferredScrollingEnabled` является вложенным свойством и может быть задано в объекте `ScrollViewer` и в любом элементе управления с `ScrollViewer` в его шаблоне элементов управления.

По умолчанию прокрутка в элементах-списках осуществляет по объектам: при перемещении скрола мы перемещаемся на один объект в списке вниз или вверх. Однако мы можем использовать более плавную прокрутку, установив для свойства

**VirtualizingStackPanel.ScrollUnit** значение Pixel:

```
<ListBox VirtualizingStackPanel.ScrollUnit="Pixel" ... />
```

# Провайдеры данных. ObjectDataProvider

Для упрощения работы с источниками данных в WPF есть такая функциональность, как **провайдеры данных**. Они позволяют связывать источники данных и элементы интерфейса. По умолчанию доступно два провайдера данных: **ObjectDataProvider** (для работы с объектами) и **XmlDataProvider** (для работы с данными из xml-файлов).

**ObjectDataProvider** позволяет использовать в качестве базы данных какой-нибудь локальный объект. Для установки подключения к объекту провайдер применяет следующие свойства:

- **ObjectType**: указывает на тип объекта, который будет выполнять роль источника
- **MethodName**: указывает на метод объекта из свойства **ObjectType**
- **ObjectInstance**: непосредственно сам объект-источник. В приложение можно одновременно использовать только одно из свойств: **ObjectType**, либо **ObjectInstance**
- **IsAsynchronous**: при значении true получает данные от источника в асинхронном режиме

К примеру, определим в файле кода класс, который представляет модель данных, и класс, который будет поставлять данные:

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
}

public class PhoneRepository
{
    private ObservableCollection<Phone> phones;

    public PhoneRepository()
    {
        phones = new ObservableCollection<Phone>
        {
            new Phone {Id=1, Title="iPhone 6S", Company="Apple" },
            new Phone {Id=2, Title="Lumia 950", Company="Microsoft" },
            new Phone {Id=3, Title="Nexus 5X", Company="Google" },
            new Phone {Id=4, Title="Galaxy S6", Company="Samsung"}
        };
    }
    public ObservableCollection<Phone> GetPhones()
    {
        return phones;
    }
}

public class Phone
{
    public int Id { get; set; }
    public string Title { get; set; } // модель телефона
    public string Company { get; set; } // производитель
}

```

Итак, класс окна MainWindow здесь абсолютно пустой, в нем не устанавливается никаких привязок или источников данных.

В качестве модели данных применяется класс Phone, а всю работу по поставке данных берет на себя класс PhoneRepository. Это стандартный репозиторий, усеченный до одного метода GetPhones(). Данный метод фактически выполняет роль источника данных.

Теперь в коде разметки xaml определим ObjectDataProvider, который свяжем с PhoneRepository:

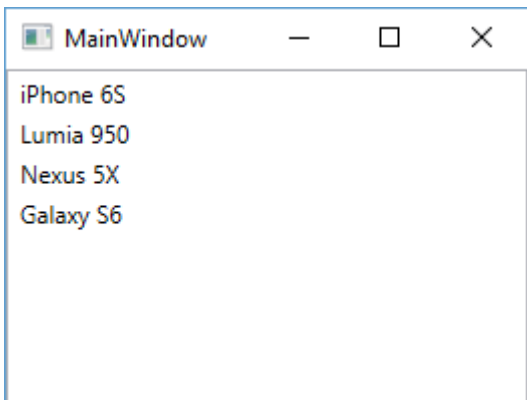
```

<Window x:Class="DataProvidersApp.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:DataProvidersApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <ObjectDataProvider x:Key="phonesProvider"
            ObjectType="{x:Type local:PhoneRepository}" MethodName="GetPhones" />
    </Window.Resources>
    <Grid>
        <ListBox x:Name="phonesList" DisplayMemberPath="Title"
            ItemsSource="{Binding Source={StaticResource phonesProvider}}" />
    </Grid>
</Window>

```

ObjectDataProvider определяется как обычный ресурс. В качестве типа объекта указываем тип PhoneRepository, а в качестве метода - GetPhones. Таким образом, ObjectDataProvider будет связан с той коллекцией объектов Phone, которая возвращается методом GetPhones.

Затем этот ресурс мы можем использовать как источник данных в элементе ListBox.



Подобным образом можно использовать свойство ObjectInstance класса ObjectDataProvider для определения источника. Для этого для ObjectInstance надо установить ссылку на ресурс - коллекцию объектов:

```

<Window x:Class="DataProvidersApp.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:DataProvidersApp"
    xmlns:col="clr-namespace:System.Collections;assembly=mscorlib"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <col:ArrayList x:Key="phones">
            <local:Phone Id="1" Title="Lumia 950" Company="Microsoft" />
            <local:Phone Id="2" Title="Elite X3" Company="HP" />
            <local:Phone Id="3" Title="Fierce XL" Company="Alcatel" />
        </col:ArrayList>
        <ObjectDataProvider x:Key="phonesProvider" ObjectInstance="{StaticResource phones}" />
    </Window.Resources>
    <Grid>
        <ListBox x:Name="phonesList" DisplayMemberPath="Title"
            ItemsSource="{Binding Source={StaticResource phonesProvider}}" />
    </Grid>
</Window>

```

# XmlDataProvider

XmlDataProvider используется для подключения к xml-документам. Причем xml-документом в данном случае может быть и локальный xml-файл, и xml из интернета. Для организации связи с источником данных XmlDataProvider использует следующие свойства:

- **Source:** устанавливает источник данных
- **XPath:** задает путь внутри документа xml к целевому набору данных
- **Document:** устанавливает ссылку на xml-документ
- **IsAsynchronous:** при значении true загружает данные асинхронно

К примеру, определим в проекте файл phones.xml:



```
<?xml version="1.0" encoding="utf-8" ?>
<phones>
  <phone>
    <id>1</id>
    <title>Lumia 950</title>
    <company>Microsoft</company>
  </phone>
  <phone>
    <id>2</id>
    <title>Elite X3</title>
    <company>HP</company>
  </phone>
  <phone>
    <id>3</id>
    <title>Fierce XL</title>
    <company>Alcatel</company>
  </phone>
</phones>
```

Теперь обратимся к этому файлу через XmlDataProvider:

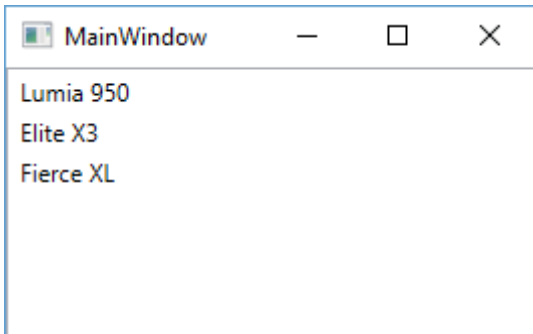
```
<Window x:Class="DataProvidersApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:DataProvidersApp"
  mc:Ignorable="d"
  Title="MainWindow" Height="350" Width="525">
  <Window.Resources>
    <XmlDataProvider x:Key="phonesProvider" Source="phones.xml" XPath="phones" />
  </Window.Resources>
  <Grid>
    <ListBox x:Name="phonesList" DisplayMemberPath="title"
      ItemsSource="{Binding Source={StaticResource phonesProvider}, XPath=phone}" />
  </Grid>
</Window>
```

XmlDataProvider определяется как ресурс. Его свойству Source присваивается название xml-файла. Свойство XPath указывает, что все данные для провайдера в файле будут находиться в рамках элемента .

В элементе ListBox как обычно устанавливаем свойство ItemsSource, при этом используя параметр привязки XPath. В данном случае он указывает на элемент из xml, на основе которого будут создаваться элементы в ListBox, то есть элемент .

Здесь важно отметить, что в XPath используются регистрозависимые строки. То есть в файле xml определен элемент - с маленькой буквы, поэтому в XPath также используется название с маленькой буквы: XPath="phones". То же самое касается и свойства DisplayMemberPath="title".

В итоге у нас получится следующий список:



## Иерархические данные и HierarchicalDataTemplate

В отличие от простых списковых элементов типа ListBox или ComboBox, элементы TreeView и Menu способны отображать иерархические данные, построенные по образцу дерева. Как у дерева могут быть ветви, у которых, в свою очередь, также могут быть ветви, так и у TreeView могут быть узлы высшего уровня, которые могут содержать подузлы, а в подузлах также могут храниться подузлы.

Для работы именно с иерархическими данными в WPF имеется специальный тип шаблонов данных - **HierarchicalDataTemplate**. Этот шаблон задает формат отображения уровня данных. Рассмотрим на примере.

Пусть у нас есть следующий класс:

```
public class Node
{
    public string Name { get; set; }
    public ObservableCollection<Node> Nodes { get; set; }
}
```

Этот класс представляет иерархические данные - объект Node может содержать несколько других объектов Node.

Теперь в коде xaml определим элемент TreeView для отображения данных:

```

<Window x:Class="DataApp.DataWindow"
    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:DataApp"
    mc:Ignorable="d"
    Title="DataWindow" Height="300" Width="300">
    <Grid>
        <TreeView x:Name="treeView1">
            <TreeView.ItemTemplate>
                <HierarchicalDataTemplate ItemsSource="{Binding Path=Nodes}">
                    <TextBlock Text="{Binding Name}" />
                </HierarchicalDataTemplate>
            </TreeView.ItemTemplate>
        </TreeView>
    </Grid>
</Window>

```

С помощью свойства `ItemTemplate` у `TreeView` задается шаблон отображения для каждого объекта `Node` в виде `HierarchicalDataTemplate`.

Атрибут `ItemsSource="{Binding Path=Nodes}"` осуществляет привязку к внутренней коллекции `Nodes`, которая имеется в объекте `Node`.

Далее элемент устанавливает привязку к свойству `Name` объекта `Node`.

Теперь в файле кода окна определим коллекцию объектов `Node`, которая будет отображаться в `TreeView`:

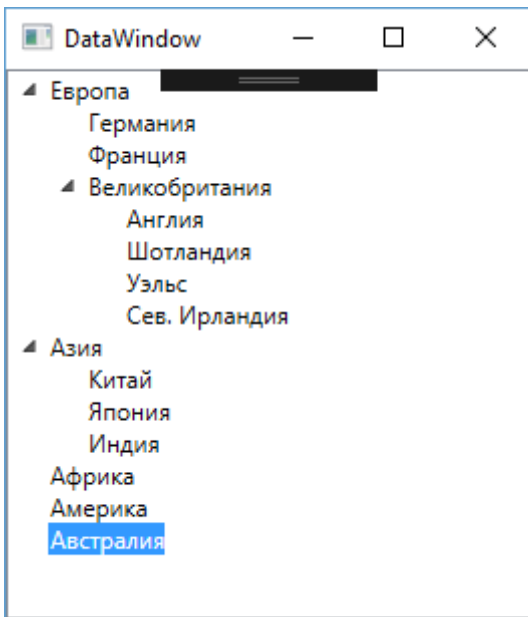
```

public partial class DataWindow : Window
{
    ObservableCollection<Node> nodes;
    public DataWindow()
    {
        InitializeComponent();

        nodes = new ObservableCollection<Node>
        {
            new Node
            {
                Name = "Европа",
                Nodes = new ObservableCollection<Node>
                {
                    new Node {Name="Германия" },
                    new Node {Name="Франция" },
                    new Node
                    {
                        Name = "Великобритания",
                        Nodes = new ObservableCollection<Node>
                        {
                            new Node {Name="Англия" },
                            new Node {Name="Шотландия" },
                            new Node {Name="Уэльс" },
                            new Node {Name="Сев. Ирландия" },
                        }
                    }
                }
            },
            new Node
            {
                Name = "Азия",
                Nodes = new ObservableCollection<Node>
                {
                    new Node {Name="Китай" },
                    new Node {Name="Япония" },
                    new Node { Name = "Индия" }
                }
            },
            new Node { Name="Африка" },
            new Node { Name="Америка" },
            new Node { Name="Австралия" }
        };
        treeView1.ItemsSource = nodes;
    }
}

```

Здесь просто создается список, который привязывается к TreeView. В итоге у нас получится следующее отображение:



Рассмотрим другой пример. Пусть объекты узлой будут отличаться по типу. Для этого определим следующие классы:

```
public class Company
{
    public string Name { get; set; }
    public ObservableCollection<Smartphone> Phones { get; set; }
    public Company()
    {
        Phones = new ObservableCollection<Smartphone>();
    }
}

public class Smartphone
{
    public string Title { get; set; }
}
```

Компания-производитель содержит список произведенных моделей телефонов.

Теперь изменим код TreeView в xaml:

```
<TreeView x:Name="treeView1">
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate ItemsSource="{Binding Path=Phones}">
      <TextBlock Text="{Binding Name}" />
      <HierarchicalDataTemplate.ItemTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Title}" />
        </DataTemplate>
      </HierarchicalDataTemplate.ItemTemplate>
    </HierarchicalDataTemplate>
  </TreeView.ItemTemplate>
</TreeView>
```

Теперь для отображения внутренних данных, то есть объектов Phone задается дополнительный шаблон:

```
<HierarchicalDataTemplate.ItemTemplate>
  <DataTemplate>
    <TextBlock Text="{Binding Title}" />
  </DataTemplate>
</HierarchicalDataTemplate.ItemTemplate>
```

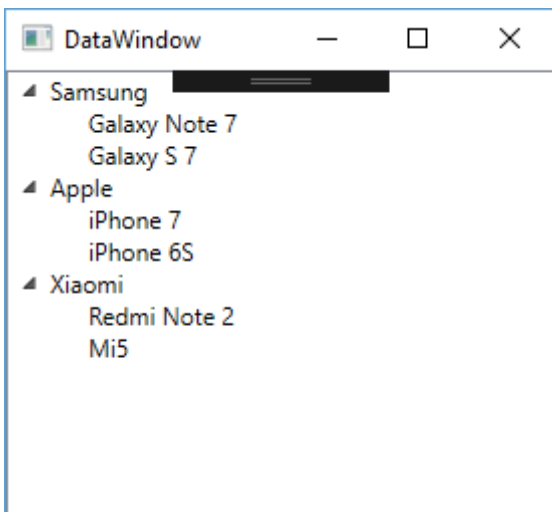
Опять же в коде C# можно создать и установить источник данных для TreeView:

```

ObservableCollection<Company> companies = new ObservableCollection<Company>()
{
    new Company
    {
        Name = "Samsung",
        Phones = new ObservableCollection<Smartphone>
        {
            new Smartphone {Title = "Galaxy Note 7" },
            new Smartphone {Title = "Galaxy S 7" }
        }
    },
    new Company
    {
        Name = "Apple",
        Phones = new ObservableCollection<Smartphone>
        {
            new Smartphone { Title="iPhone 7" },
            new Smartphone { Title="iPhone 6S"}
        }
    },
    new Company
    {
        Name="Xiaomi",
        Phones = new ObservableCollection<Smartphone>
        {
            new Smartphone {Title="Redmi Note 2" },
            new Smartphone {Title="Mi5" }
        }
    }
};
treeView1.ItemsSource = companies;

```

В итоге получится следующее приложение:

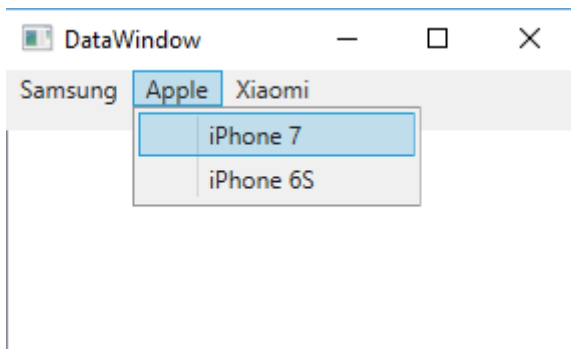


Аналогичным образом можно определить шаблон для элемента Menu:

```

<Menu x:Name="mainMenu" VerticalAlignment="Top" Height="30">
    <Menu.ItemTemplate>
        <HierarchicalDataTemplate ItemsSource="{Binding Path=Phones}">
            <TextBlock Text="{Binding Name}" />
            <HierarchicalDataTemplate.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding Title}" />
                </DataTemplate>
            </HierarchicalDataTemplate.ItemTemplate>
        </HierarchicalDataTemplate>
    </Menu.ItemTemplate>
</Menu>

```



## Получение данных из xml

Для представления иерархических данных очень удобен формат xml. И в WPF мы можем легко связать данные из xml с иерархическим элементом - TreeView или Menu. Допустим, у нас есть в проекте следующий файл nodes.xml:



```

<?xml version="1.0" encoding="utf-8" ?>
<nodes>
  <node title="Европа">
    <node title="Германия" />
    <node title="Франция" />
    <node title="Великобритания">
      <node title="Англия" />
      <node title="Шотландия" />
      <node title="Уэльс" />
      <node title="Сев. Ирландия" />
    </node>
  </node>
  <node title="Азия">
    <node title="Китай" />
    <node title="Япония" />
    <node title="Индия" />
  </node>
</nodes>

```

С помощью XmlDataProvider мы можем подключить этот файл к TreeView:

```

<Window x:Class="DataApp.DataWindow"
  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:DataApp"
  mc:Ignorable="d"
  Title="DataWindow" Height="300" Width="300">
  <Window.Resources>
    <XmlDataProvider x:Key="nodesProvider" Source="nodes.xml" XPath="nodes/node" />
  </Window.Resources>
  <Grid>

    <TreeView ItemsSource="{Binding Source={StaticResource nodesProvider}}">
      <TreeView.ItemTemplate>
        <HierarchicalDataTemplate ItemsSource="{Binding XPath=node}">
          <TextBlock Text="{Binding XPath=@title}" />
        </HierarchicalDataTemplate>
      </TreeView.ItemTemplate>
    </TreeView>
  </Grid>
</Window>

```

## Валидация данных

При работе с данными важную роль играет валидация данных. Прежде чем использовать полученные от пользователя данные, нам надо убедиться, что они введены правильно и представляют корректные значения. Один из встроенных способов проверки введенных данных в WPF представлен классом `ExceptionValidationRule`. Этот класс обозначает введенные данные как некорректные, если в процессе ввода возникает какое-либо исключение, например, исключение преобразования типов.

Итак, допустим, у нас определен следующий класс:

```
public class PersonModel
{
    public string Name { get; set; }
    public int Age { get;set;}
    public string Position { get; set; }
}
```

Этот класс представляет человека и предполагает три свойства: имя, возраст и должность. Понятно, что возраст должен представлять числовое значение. Однако пользователи могут ввести что угодно. Мы можем обрабатывать ввод с клавиатуры, а можем воспользоваться классом `ExceptionValidationRule`, который в случае неудачи преобразования строки в число установит красную границу вокруг текстового поля.

Сначала создадим в файле кода объект нашего класса `PersonModel` и установим контекст данных окна:

```
public partial class MainWindow : Window
{
    PersonModel Tom;
    public MainWindow()
    {
        InitializeComponent();
        Tom=new PersonModel();
        this.DataContext = Tom;
    }
}
```

Теперь установим привязку в xaml-коде:

```

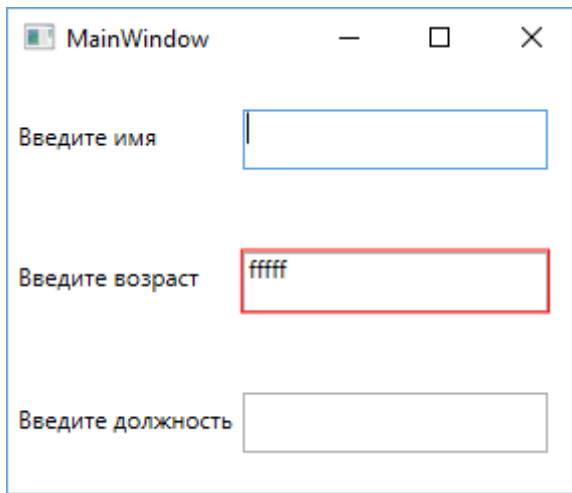
<Window x:Class="DataValidationApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:DataValidationApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="250" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <TextBox Grid.Column="1" Height="30" Margin="0 0 15 0"/>

        <TextBox Grid.Column="1" Grid.Row="1" Height="30" Margin="0 0 15 0">
            <TextBox.Text>
                <Binding Path="Age">
                    <Binding.ValidationRules>
                        <ExceptionValidationRule />
                    </Binding.ValidationRules>
                </Binding>
            </TextBox.Text>
        </TextBox>

        <TextBox Grid.Column="1" Grid.Row="2" Height="30" Margin="0 0 15 0" />
        <Label Content="Введите имя" Height="30" />
        <Label Grid.Row="1" Content="Введите возраст" Height="30" />
        <Label Grid.Row="2" Content="Введите должность" Height="30" />
    </Grid>
</Window>

```

В данном случае мы задаем объект Binding для свойства Text. Данный объект имеет коллекцию правил валидации вводимых данных - **ValidationRules**. Эта коллекция принимает только одно правило валидации, представленное классом **ExceptionValidationRule**. Запустим приложение на выполнение и попробуем ввести в текстовое поле какое-нибудь нечисловое значение. В этом случае текстовое поле будет обведено красным цветом, указывая на то, что в вводимых данных имеются ошибки.



Мы также можем реализовать свою логику валидации для класса модели. Для этого модель должна реализовать интерфейс `IDataErrorInfo`. Этот интерфейс имеет следующий синтаксис:

```
public interface IDataErrorInfo
{
    string Error {get;}
    string this[string columnName] { get;}
}
```

Допустим, мы хотим ограничить возраст человека только положительными значениями от 0 до 100. Тогда валидация модели будет выглядеть следующим образом:

```

public class PersonModel : IDataErrorInfo
{
    public string Name { get; set; }
    public int Age {get;set;}
    public string Position { get; set; }
    public string this[string columnName]
    {
        get
        {
            string error=String.Empty;
            switch (columnName)
            {
                case "Age" :
                    if ((Age < 0) || (Age > 100))
                    {
                        error = "Возраст должен быть больше 0 и меньше 100";
                    }
                    break;
                case "Name" :
                    //Обработка ошибок для свойства Name
                    break;
                case "Position" :
                    //Обработка ошибок для свойства Position
                    break;
            }
            return error;
        }
    }
    public string Error
    {
        get { throw new NotImplementedException(); }
    }
}

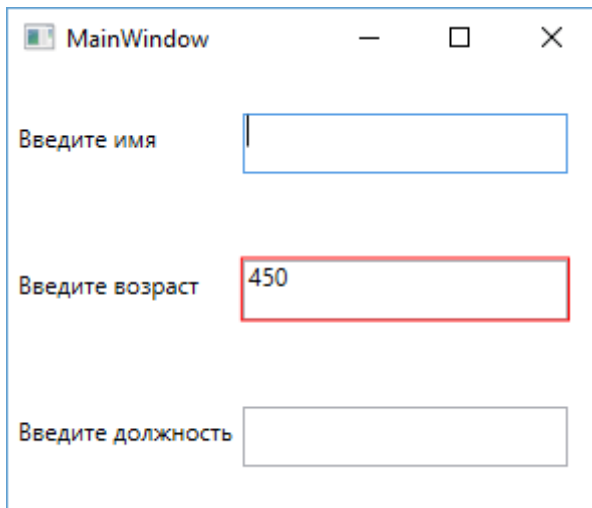
```

И последнее - нам осталось немного подкорректировать xaml-код. Теперь нам надо использовать в качестве правила валидации класс **DataErrorValidationRule**:

```

<TextBox Grid.Column="1" Grid.Row="1" Height="30" Margin="0 0 15 0">
    <TextBox.Text>
        <Binding Path="Age">
            <Binding.ValidationRules>
                <DataErrorValidationRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>

```



MainWindow

Введите имя

Введите возраст 450

Введите должность

Так как число 450 больше 100 и поэтому не является валидным, то текстовое поле выделяется красным.

## Настройка внешнего вида при ошибке валидации

Но это еще не все. Мы можем сами управлять через шаблоны отображением ошибки ввода. В предыдущем случае у нас граница текстового поля при ошибке окрашивалась в красный цвет. Попробуем настроить данное действие. Для этого нам нужно использовать элемент **AdornedElementPlaceholder**. Итак изменим разметку приложения следующим образом, добавив в нее шаблон элемента управления:

```

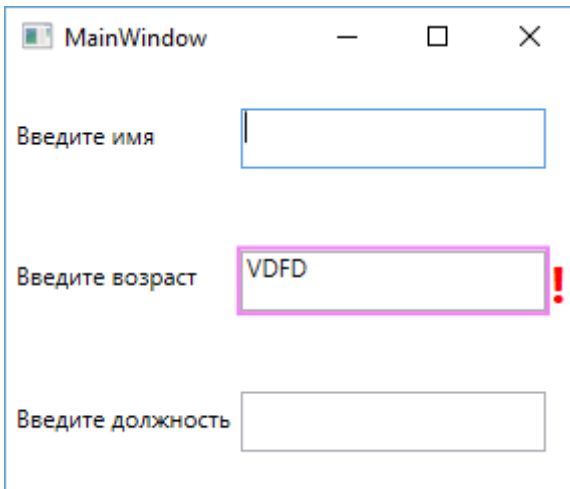
<Window x:Class="DataValidationApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:DataValidationApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="250" Width="300">
<Window.Resources>
    <ControlTemplate x:Key="validationFailed">
        <StackPanel Orientation="Horizontal">
            <Border BorderBrush="Violet" BorderThickness="2">
                <AdornedElementPlaceholder />
            </Border>
            <TextBlock Foreground="Red" FontSize="26" FontWeight="Bold">!</TextBlock>
        </StackPanel>
    </ControlTemplate>
</Window.Resources>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBox Grid.Column="1" Height="30" Margin="0 0 15 0"/>

    <TextBox Grid.Column="1" Grid.Row="1" Height="30" Margin="0 0 15 0"
        Validation.ErrorTemplate="{StaticResource validationFailed}" >
        <TextBox.Text>
            <Binding Path="Age">
                <Binding.ValidationRules>
                    <DataErrorValidationRule />
                </Binding.ValidationRules>
            </Binding>
        </TextBox.Text>
    </TextBox>

    <TextBox Grid.Column="1" Grid.Row="2" Height="30" Margin="0 0 15 0" />
    <Label Content="Введите имя" Height="30" />
    <Label Grid.Row="1" Content="Введите возраст" Height="30" />
    <Label Grid.Row="2" Content="Введите должность" Height="30" />
</Grid>
</Window>

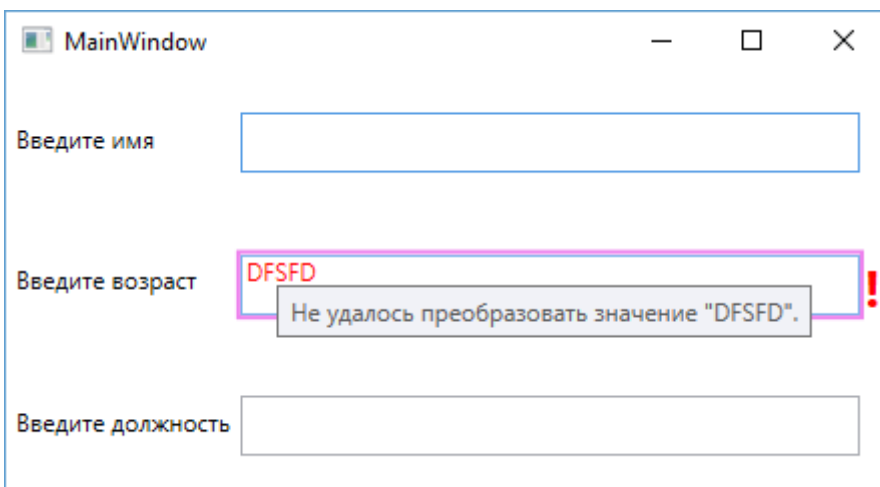
```

С помощью свойства **Validation.ErrorTemplate** мы получаем шаблон, который будет отрабатывать при ошибке валидации. Этот шаблон, определенный выше в ресурсах окна, определяет границу фиолетового цвета вокруг элемента ввода, а также отображает рядом с ним восклицательный знак красного цвета. Запустим приложение и попробуем ввести в текстовое поле какое-нибудь некорректное значение. В результате сработает наш шаблон:



Мы также можем определить поведение и визуализацию через триггер при установке свойства **Validation.HasError** в True. А с помощью свойства **ToolTip** можно создать привязку к сообщению ошибки:

```
<Style TargetType="TextBox">
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="True">
            <Setter Property="ToolTip"
                Value="{Binding RelativeSource={RelativeSource Self},
                    Path=(Validation.Errors)[0].ErrorContent}" />
            <Setter Property="Foreground" Value="Red" />
        </Trigger>
    </Style.Triggers>
</Style>
```





# Обработка событий валидации

WPF предоставляет механизм обработки ошибки валидации с помощью события `Validation.Error`. Данное событие можно использовать в любом элементе управления. Например, пусть при ошибке валидации при вводе в текстовое поле выскакивает сообщение с ошибкой. Для этого изменим текстовое поле следующим образом:

```
<TextBox Grid.Column="1" Grid.Row="1" Height="30" Margin="0 0 15 0" Validation.Error="TextBox_E
    <TextBox.Text>
        <Binding Path="Age" NotifyOnValidationError="True">
            <Binding.ValidationRules>
                <DataErrorValidationRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Здесь, во-первых, надо отметить установку свойства `NotifyOnValidationError="True"`:

```
<Binding Path="Age" NotifyOnValidationError="True">
```

Это позволит вызывать событие валидации.

И также устанавливается сам обработчик события валидации:

```
<TextBox Grid.Column="1" Grid.Row="1" Height="30" Margin="0 0 15 0"
    Validation.Error="TextBox_Error">
```

При этом следует отметить, что событие `Validation.Error` является поднимающимся (bubbling events), поэтому мы можем установить для него обработчик и в контейнере `Grid` или в любых других контейнерах, в которых находится это текстовое поле. И в случае ошибки событие также будет генерироваться и обрабатываться.

И в конце определим в файле кода `c#` сам обработчик:

```
private void TextBox_Error(object sender, ValidationErrorEventArgs e)
{
    MessageBox.Show(e.Error.ErrorContent.ToString());
}
```