

ПРИВЯЗКА И РАСШИРЕНИЕ РАЗМЕТКИ

Расширение разметки – это специальная конструкция **XAML**, позволяющая вместо конкретных значений указывать ссылку на объект значения.

Расширение разметки подставляется при установке свойства элемента и оформляется между символами **{** и **}**.

```
<TextBlock Text="{x:Null}"/>
```

В данном примере, текстовому полю присваивается значение **null**.

Синтаксис расширений

Синтаксис расширений выглядит следующим образом.

```
<object Property="{ExtensionObj ExProperty=value}"/>
```

Рисунок 1

Object – элемент разметки

Property – одно из свойств Object, которому можно задать значение

ExtensionObj – специальный объект расширения

ExProperty – свойство объекта расширения

Value – значение свойства объекта расширения

Если объект сложный, то ему можно задавать аргументы или свойства. Например, для объекта расширения **Binding** можно задать как сокращённую запись, так и полную.

```
<TextBlock Text="{Binding MyTextProperty}"/>
```

Рисунок 2

В этой сокращённой записи значение аргумента расширения **Binding** (*MyTextProperty*) присваивается его свойству **Path**. Ниже указана та же запись, но в полном формате.

```
<TextBlock Text="{Binding Path=MyTextProperty}"/>
```

Если объект расширения сложный и требует установки нескольких свойств, они перечисляются через запятую.

```
<TextBlock Text="{Binding Path=MyTextProperty, Mode=Default}"/>
```

Рисунок 3

Свойства расширения так же могут представлять объект с несколькими свойствами. Создание объекта для свойства расширения задаётся так же фигурными скобками { и }.

```
<TextBlock Text="{Binding RelativeSource={RelativeSource AncestorType=Window},  
Path=Title}"/>
```

Рисунок 4

Расширения разметки, определённые в XAML

В пространстве имён с базовым префиксом x предопределены следующие объекты расширенной разметки:

x:Null – представляет значение **null**

```
<object property="{x:Null}" .../>
```

Рисунок 5

x:Static – создаёт статическое значение

```
<object property="{x:Static prefix:typeName.staticMemberName}" .../>
```

Рисунок 6

x:Array – задаёт список значений. Не используется в специальной разметке.

Расширения разметки для WPF

Пространство имён **WPF** предоставляет возможность привязывать ресурсы и другие данные.

● Привязка ресурсов

За привязку ресурсов отвечает расширение **StaticResource** и **DynamicResource**. Их разница в том, что **StaticResource** объявляется и не может быть изменён во время выполнения программы, в то время как **DynamicResource** поддерживает такую возможность.

Binding предоставляет для свойства привязанное к данным значение, используя контекст данных, который применяется к родительскому объекту во время выполнения. Это расширение разметки довольно сложное, поскольку разрешает использовать важный встроенный синтаксис для указания привязки данных.

RelativeSource предоставляет исходные сведения для класса **Binding**, который может перемещаться по нескольким возможным связям в дереве объектов времени выполнения. Это обеспечивает специализированные источники для привязок, которые создаются в шаблонах для многократного использования или в коде без полных сведений об окружающем дереве объектов.

Расширение Binding

● Схема привязки

Общая схема привязки представлена на следующей схеме.

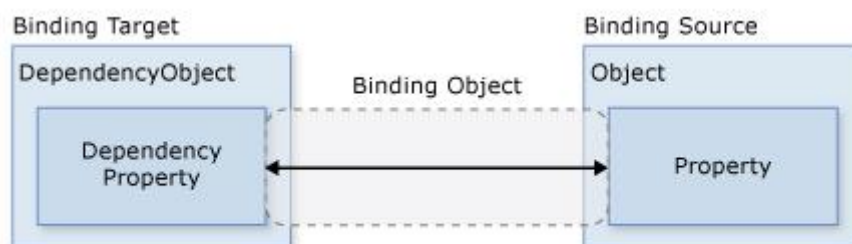


Рисунок 7

BindingTarget – Цель привязки (к чему будут привязываться данные).

BindingSource – Источник привязки (откуда берутся данные).

DependencyObject – специальный объект, который реализует зависимые свойства. От класса **DependencyProperty** наследуются все элементы **WPF**.

DependencyProperty – это зарегистрированное свойство, которое называют свойством зависимости. Оно поддерживает привязку (зависимость от источника).

● Направление привязки

Обновление свойства привязки может быть определено в одном из трёх направлений.

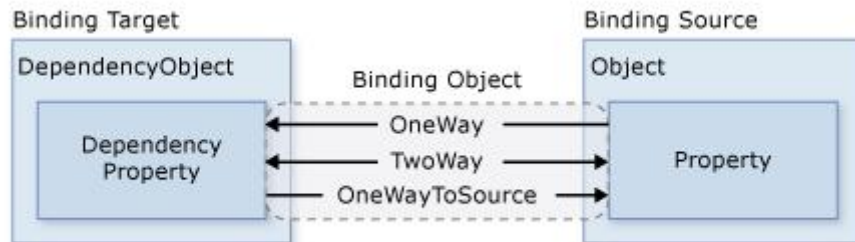


Рисунок 8

OneWay – целевое свойство обновляется исходя из обновления источника.

TwoWay – для обновления учитываются как изменения целевого свойства, так и свойства источника.

OneWayToSource – свойство источника обновляется исходя из обновления целевого свойства.

● Оповещение об изменениях свойств источника

Чтобы обнаружить изменения источника (применимые к привязкам *OneWay* и *TwoWay*), источник должен реализовать подходящий механизм уведомления об изменении свойств, например **INotifyPropertyChanged** – это интерфейс, который реализует источник данных для оповещения об изменениях своих внутренних свойств.

Интерфейс имеет только одно публичное событие **PropertyChanged**. Система подписывается на него. Так же, чтобы источник смог оповестить об изменениях (вызвать событие *PropertyChanged*) необходимо описать специальный метод. Обычно его называют **OnPropertyChanged**.

Интерфейс **INotifyPropertyChanged** (сокращённо называют *INPC*) описан в пространстве имён **System.ComponentModel**.

```

using System.ComponentModel;
//ссылка: 1
public class Class1 : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
}

```

Рисунок 9

Для того, чтобы уведомить об изменении конкретного свойства источника, необходимо выполнить некоторую проверку. Легче описать её в методе.

```

using System.ComponentModel;
//ссылка: 0
public class Class1 : INotifyPropertyChanged
{
    //Публичное событие, на которое подписывается WPF для обновления
    public event PropertyChangedEventHandler PropertyChanged;

    //Получает на вход имя свойства, которое изменило значение
    //ссылка: 0
    public void OnPropertyChanged (string property)
    {
        //Если событие не пустое, его вызывают
        //Делегат события требует передачи источника, поэтому передают this
        //И аргументы изменяемого свойства, по которому WPF может получить
        //новое значение
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(property));
    }
}

```

Рисунок 10

В данном примере источником является **Class1**. Теперь можно описать свойство, и при изменении вызвать метод **OnPropertyChanged**.

```

using System.ComponentModel;
...
Ссылка: 0
public class Class1 : INotifyPropertyChanged
{
    private string _myProperty;
    Ссылка: 0
    public string MyProperty
    {
        get { return _myProperty; }
        set { _myProperty = value; OnPropertyChanged("MyProperty"); }
    }

    //Публичное событие, на которое подписывается WPF для обновления
    public event PropertyChangedEventHandler PropertyChanged;

    //Получает на вход имя свойства, которое изменило значение
    Ссылка: 0
    public void OnPropertyChanged_(string property)
    {
        //Если событие не пустое, его вызывают
        //Делегат события требует передачи источника, поэтому передают this
        //И аргументы изменяемого свойства, по которому WPF может получить
        //новое значение
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(property));
    }
}

```

Рисунок 11

Причём, если случаются ситуации, когда при изменении одного свойства надо оповестить об изменении другого, в первом свойстве можно вызвать ещё один метод **OnPropertyChanged**, в который передаётся имя дополнительного свойства.

Можно пойти ещё на одну хитрость. При написании свойств в параметре **OnPropertyChanged** может возникнуть опечатка. Можно доверить передачу имени, вызвавшего метод **OnPropertyChanged** свойства рантайму, где во время выполнения среда .NET автоматически подставляет имя свойства. Возможность указать другое имя всё ещё остаётся и его так же можно задать вручную реализуется это с помощью атрибута **CallerMemberName** из пространства имён **System.Runtime.CompilerServices**.

```

public void OnPropertyChanged([CallerMemberName] string property=null)

```

Рисунок 12

Источников привязки может быть большое количество. Чтобы не нагромождать код реализацией интерфейса в каждом источнике, его реализацию выносят в отдельный класс, от которого потом наследуются.


```

using System.ComponentModel;
namespace myNamespace
{
    Ссылка: 0
    public class Class1 : NotifyClass
    {
        private string _myProperty;
        Ссылка: 0
        public string MyProperty
        {
            get { return _myProperty; }
            set { _myProperty = value; OnPropertyChanged("MyProperty"); }
        }
    }
    ссылка: 1
    public class NotifyClass : INotifyPropertyChanged
    {
        //Публичное событие, на которое подписывается WPF для обновления
        public event PropertyChangedEventHandler PropertyChanged;

        //Получает на вход имя свойства, которое изменило значение
        ссылка: 1
        public void OnPropertyChanged(string property)
        {
            //Если событие не пустое, его вызывают
            //Делегат события требует передачи источника, поэтому передают this
            //И аргументы изменяемого свойства, по которому WPF может получить
            //новое значение
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(property));
        }
    }
}

```

Рисунок 13

● Источник данных

Расширение **Binding** отвечает за привязку данных из источника. Источник представляется объектом и записывается в специальное свойство элемента **DataContext**. Это свойство существует у любого элемента, который отображает данные.

```

<Window x:Class="WpfApp1.MainWindow"
    xmlns="http://schemas.microsoft.com/w
    xmlns:x="http://schemas.microsoft.com
    xmlns:d="http://schemas.microsoft.com
    xmlns:mc="http://schemas.openxmlforma
    xmlns:local="clr-namespace:WpfApp1"
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width
    x:Name="mw"
    Data>
    <Grid
        DataContext
        DataContextChanged
    </Gr {} DataGridCell
    ...

```

Рисунок 14

DataContext для привязки выбирается исходя из дерева элементов. Свойство зависимости, для которого описывается привязка, выбирает ближайший по иерархии **DataContext**.

```

<Window x:Class="WpfApp1.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:WpfApp1"
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800"
    x:Name="mw">
    <Window.DataContext>
        <x:Array Type="{x:Type SolidColorBrush}">
            <SolidColorBrush Color="AliceBlue"/>
            <SolidColorBrush Color="BurlyWood"/>
            <SolidColorBrush Color="Coral"/>
        </x:Array>
    </Window.DataContext>
    <Grid>
        <Grid.DataContext>
            <x:Array Type="{x:Type SolidColorBrush}">
                <SolidColorBrush Color="Fuchsia"/>
                <SolidColorBrush Color="Ivory"/>
                <SolidColorBrush Color="Silver"/>
            </x:Array>
        </Grid.DataContext>
        <TextBlock Background="{Binding Path=[0]}"
            Text="{Binding ElementName=mw, Path=Title}"/>
    </Grid>
</Window>

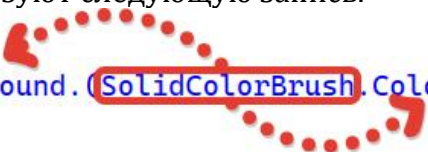
```

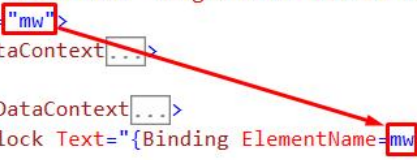
Рисунок 15

В данном примере **DataContext** задаётся как сложное свойство для двух элементов – **Window** и **Grid**. **TextBlock**, который использует зависимость от **DataContext** возьмёт ближайший к нему **DataContext** вверх по дереву, и это будет контекст элемента **Grid**. Поэтому фон **TextBlock'a** окрасится в цвет **Fuchsia**.

Нужно учесть, что привязку поддерживают только свойства зависимости (*dependency property*) – те свойства, которые зарегистрированы для отслеживания в **WPF** с помощью специальной конструкции.

Перечисление основных свойств класса **Binding**

Название атрибута	Описание	Синтаксис
Source	Задаёт объект привязки.	Используется для указания того, из какого объекта необходимо выбирать свойство (которое далее указывается с помощью атрибута Path). Вложенность указывается через точку. Binding Source=Object.Property
Path	Задаёт свойство привязки.	Используется для указания имени свойства данных из Source. Path=ValueProperty Если имеет место быть вложенность, то разрешается указывать только один уровень вложенности (одна точка). Path=Object.Value Так же можно указывать индекс записи, если свойство представляет из себя список. 1) Если свойство списка указано в пути Source Binding Path=[0] Или 2) Если свойство не указано в Source Binding Path=ListProperty[0] Некоторые объекты позволяют хранить значения разных типов (например, Background поддерживает множества типов кистей). Если необходимо выбрать конкретное свойство из объекта, представляющий абстрактный тип, необходимо его привести к конкретному. На примере с Background и выборки из него Color зная, что заливка представляет из себя SolidColorBrush, используют следующую запись.  {Binding Path=Background.(SolidColorBrush.Color)}

		<p>В примере объект Background приводят к типу данных SolidColorBrush и выбирают его свойство Color.</p> <p>Подобным синтаксисом пользуются ещё и тогда, когда пытаются узнать значение прикреплённого к элементу свойства (например DockPanel.Dock).</p> <p>Ещё больше про синтаксис свойства Path можно узнать из документации PropertyPath XML.</p>
ElementName	Задаёт объект элемента интерфейса для привязки.	<p>Заменяет свойство Source, но в качестве источника используется элемент разметки. С помощью этого атрибута можно привязаться к конкретному элементу. Для привязки элемент-источник именуют с помощью x:Name, а после указывают это имя.</p> <pre> <Window x:Class="WpfApp1.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:WpfApp1" mc:Ignorable="d" Title="MainWindow" Height="450" Width="800" x:Name="mw"> <Window.DataContext...> <Grid> <Grid.DataContext...> <TextBlock Text="{Binding ElementName=mw, Path=Title}"/> </Grid> </Window> </pre>  <p>В данном примере элемент TextBlock привязывается к элементу Window, из которого выбирается свойство Title.</p>
RelativeSource	Задаёт объект привязки относительно дерева элементов.	<p>Привязывается к одному из родительских элементов в иерархии/ Это свойство является сложным. Для того, чтобы указать ему значение так же указывают расширение разметки.</p> <pre> <TextBlock Text="{Binding RelativeSource={RelativeSource AncestorType=Window}, Path=Title}"/> </pre> <p>Объект RelativeSource имеет 3 атрибута: FindMode – способ нахождения ресурса. Если необходимо определить свойства, описанные ниже, подставляют значение FindAncestor.</p>

		<p>PreviousData – предшествующий в дереве объект данных, Self – привязка к самому себе, TemplatedParent – указывается для элементов в шаблоне.</p> <p>AncestorType – тип элемента родителя AncestorLevel – уровень элемента привязки. Для ближайшего элемента в дереве указывают 1.</p>
TemplateBinding	Используется для привязки к источнику шаблона.	<p>Шаблоны используются для описания разметки элемента управления, элемента списка, типа данных. Для того, чтобы указать на то, что какой-то элемент шаблона должен брать данные из свойства «базового» элемента, используют этот объект.</p> <p>TemplateBinding Property=Context</p>

Ресурсы WPF

Ресурсы WPF – это объекты, которые можно использовать многократно. Тем самым уменьшается объём кода, а также унифицируются настройки. Так, если вы хотите объявить один цвет для всех кнопок приложения, можно указать его в качестве ресурса и при желании заменить его только в одной точке кода.

В роли ресурсов в **WPF** могут выступать любые объекты, поддерживаемые в **WPF**.

```
<Window.Resources>
  <x:Array x:Key="tt" Type="{x:Type SolidColorBrush}">
    <SolidColorBrush Color="AliceBlue"/>
    <SolidColorBrush Color="BurlyWood"/>
    <SolidColorBrush Color="Coral"/>
  </x:Array>
  <Button x:Key="ff"/>
</Window.Resources>
```

Рисунок 16

В данном случае были объявлены два ресурса: **ff** –кнопка и **tt** – массив цветов. Теперь к ним можно обращаться из любой вложенной точки разметки.

```
<Grid>
  <ContentPresenter Content="{StaticResource ff}"/>
  <TextBlock Background="{Binding Source={StaticResource tt},
    Path=[2]}"
    Height="50"/>
</Grid>
```

Рисунок 17

В данном примере были привязаны ресурсы по их уникальному ключу.

Каждое свойство должно иметь ключ **x:Key**. Это очевидное правило и нужно для того, чтобы обращаться к ресурсу по его имени.

Ресурс может использоваться как статический или динамический. Ссылки создаются с помощью расширения разметки **StaticResource** или **DynamicResource** соответственно.

Статичная ссылка на ресурс присваивает значение ресурса, которое изначально определено в разметке в блоке **Resources** элемента. Изменения, возникшие в ресурсах, не повлекут за собой изменений в отрисовке. Используйте статическую ссылку, если ресурс определён изначально.

Динамическая ссылка на ресурс представляет из себя уже действительно ссылку на ресурс. При этом ресурс может быть не определён при запуске (*отсутствовать в борке Resources элемента*), так же он не будет загружен, пока на него никто не ссылается. Это позволяет ему реагировать на изменение значения во время исполнения программы.

StaticResource – определён изначально, **DynamicResource** – отложенное определение во время выполнения программы.

● Словарь ресурсов

Словарь ресурсов представляет коллекцию ресурсов. Несколько ресурсов, указанные в **Element.Resources** неявно оборачиваются в объект **ResourceDictionary**.

```
<ResourceDictionary>
  <!--Ресурсы-->
</ResourceDictionary>
```

Для **ResourceDictionary** так же, как и для любых других элементов, можно создать отдельный файл **XAML**.

Для словаря ресурсов есть отдельный шаблон файла.

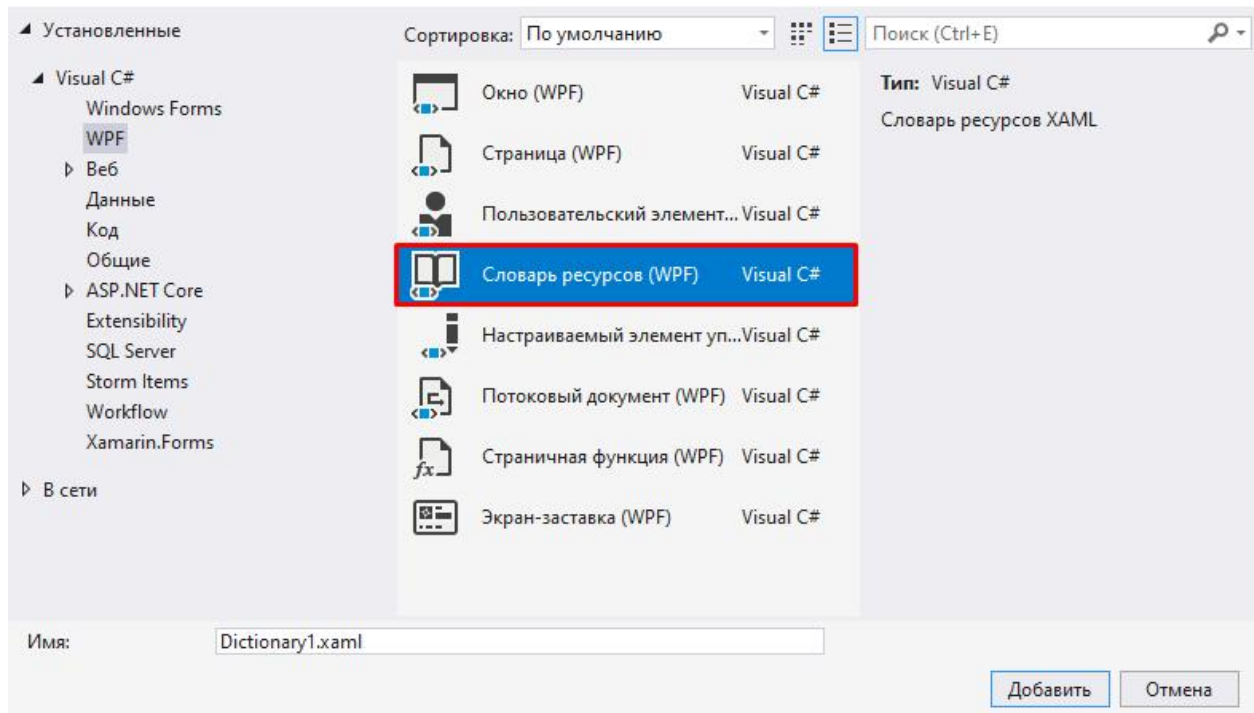


Рисунок 18

Если в системе будет несколько словарей, которые необходимо использовать одновременно, для словаря можно установить объединение с помощью **MergedDictionary**.

```
<ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary>
      <!--Ресурсы-->
    </ResourceDictionary>
    <ResourceDictionary>
      <!--Ресурсы-->
    </ResourceDictionary>
    <ResourceDictionary>
      <!--Ресурсы-->
    </ResourceDictionary>
  </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

Рисунок 19

Такой подход обычно используют для объявления словарей в **App.xaml**, так как этот файл является главной точкой для получения зависимостей (*корневой элемент приложения*).

Если словарь ресурсов находится вне файла, для него объявляется словарь ресурсов в текущем файле с указанием пути в свойстве **Source**.

Например, **App.xaml** использует внешний словарь ресурсов **Style.xaml**.

```
<Application x:Class="TestApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:TestApp"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary Source="Style.xaml"/>
    </Application.Resources>
</Application>
```

Рисунок 20

Шаблоны и стили

Шаблон – это структура разметки и отображения элемента. Шаблон можно установить для типа данных (*DataTemplate*), для элемента управления (*ControlTemplate*) и для элемента списка (*ItemTemplate*).

Шаблоны могут быть определены как ресурс, и его можно использовать повторно.

● Шаблон элемента

У каждого элемента есть свойство шаблона, которое определяет шаблон отображения – **Template**. Это свойство принимает в качестве значения объект **ControlTemplate**. Этот объект имеет свойство **TargetType**, которое необходимо указать для согласованности типов (чтобы не присвоить шаблон, предназначенный кнопке, например, текстовому полю).

Например, для кнопки начало определения шаблона будет выглядеть следующим образом.

```

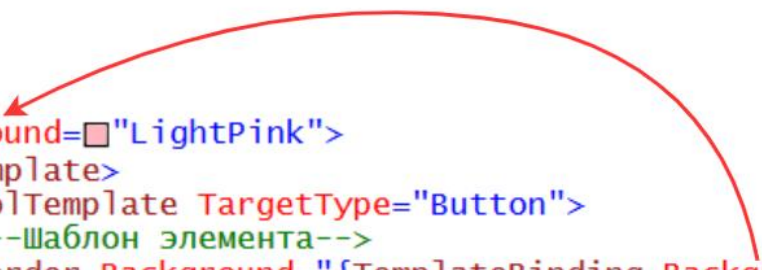
<Button>
  <Button.Template>
    <ControlTemplate TargetType="Button">
      <!--
        Шаблон элемента
      >
      <Border>
        <ContentPresenter/>
      </Border>-->
    </ControlTemplate>
  </Button.Template>
</Button>

```

Рисунок 21

Шаблон принимает в себя разметку, которая может иметь только один корневой элемент. Шаблон переопределяет базовый.

Для того, чтобы связаться со свойствами элементов из шаблона (например, *Content* или *Width*) необходимо использовать расширение разметки **TemplateBinding**. Например, сделаем так, чтобы цвет кнопки менялся в зависимости от установленного свойства **Background**.



```

<Button Background="LightPink">
  <Button.Template>
    <ControlTemplate TargetType="Button">
      <!--Шаблон элемента-->
      <Border Background="{TemplateBinding Background}">
        <ContentPresenter/>
      </Border>
    </ControlTemplate>
  </Button.Template>
</Button>

```

Рисунок 22

Если не установить цвет, то кнопка возьмёт базовый для кнопки серый цвет. Откуда он берётся – из базового стиля. Это следующая тема.

В шаблоне не зря применён элемент **ContentPresenter**. Этот элемент служит для отображения контентной части. Обычно к нему привязывают целевое свойство **Content**. Не путайте **ContentPresenter** и **ContentControl** – второй всего обозначает элемент управления с контентной частью. Из этого можно сделать предположение, что в **ContentControl** для отображения контента используется **ContentPresenter** как в примере, и это действительно так.

Чтобы несколько кнопок могли использовать шаблон, его стоит поместить в ресурсы. Например так, как показано далее.

● Шаблон данных

Шаблон данных правильнее шаблоном типа данных. Шаблоны данных чаще всего используются для определения шаблона элементов записей из элементов коллекций (таких как *ListBox*, *ListView*, *ItemsControl*). Использовать шаблон данных полезно, например, тогда, когда список имеет элементы общего типа, но каждый из них представляет более конкретный.

Шаблон данных определяется тегом **DataTemplate**.

Например, пусть есть класс **Person** с тремя свойствами.

```
class Person
{
    Ссылка: 0
    public string Name { get; set; }
    Ссылка: 0
    public int Age { get; set; }
    Ссылка: 0
    public string Description { get; set; }
}
```

Рисунок 23

Теперь определим типы, которые будут наследоваться от **Person** – **Student** и **Worker**.

```
class Student : Person
{
    Ссылка: 0
    public string Group { get; set; }
}

class Worker : Person
{
    Ссылка: 0
    public string WorkName { get; set; }
}
```

Рисунок 24

Для этих типов можно определить шаблон. Определим их в ресурсах окна в виде списка.


```

<x:Array Type="{x:Type local:Person}" x:Key="PersonList">
  <local:Student Name="Иван"
    Age="15"
    Description="Крутой парень"
    Group="123"/>
  <local:worker Name="Петя"
    Age="25"
    Description="Петя работает на работе."
    WorkName="Работа"/>
</x:Array>

```

Рисунок 25

Так же в ресурсах окна определяется **DataTemplate**. Если у него не определён атрибут **x:Key**, шаблон отображения будет проецироваться на каждый элемент с типом, определённым в свойстве **DataType**.

```

<!--Шаблон для типа Student-->
<DataTemplate DataType="{x:Type local:Student}">
  <StackPanel Background="LightCyan">
    <TextBlock Text="{Binding Name}"
      HorizontalAlignment="Center"
      FontSize="30"/>
    <TextBlock Text="{Binding Age}"
      HorizontalAlignment="Center"
      FontSize="50"/>
    <TextBlock Text="{Binding Description}"
      HorizontalAlignment="Center"/>
    <TextBlock Text="{Binding Group, StringFormat='Группа {0}'}"
      HorizontalAlignment="Center"/>
  </StackPanel>
</DataTemplate>

```

Рисунок 26

```

<!--Шаблон для типа Worker-->
<DataTemplate DataType="{x:Type local:Worker}">
    <Border CornerRadius="50"
        BorderThickness="3"
        BorderBrush="■"Black"
        Background="■"SteelBlue">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <TextBlock Text="{Binding Name}"
                HorizontalAlignment="Center"
                VerticalAlignment="Center"
                FontSize="40"/>

            <StackPanel Grid.Column="1">
                <TextBlock Text="{Binding WorkName}"
                    HorizontalAlignment="Center"
                    FontSize="25"
                    FontWeight="Bold"/>
                <TextBlock Text="{Binding Age}"
                    HorizontalAlignment="Center"
                    FontSize="50"/>
                <TextBlock Text="{Binding Description}"
                    HorizontalAlignment="Center"/>
            </StackPanel>
        </Grid>
    </Border>
</DataTemplate>

```

Рисунок 27

Далее определяем **ListView** в разметке окна, указывая источник записей в свойстве **ItemsSource**.

```

<ListView ItemsSource="{StaticResource PersonList}"
    HorizontalContentAlignment="Stretch"/>

```

Рисунок 28

В итоге, в одном **ListView** мы видим отображение элементов с разными шаблонами.



Рисунок 29

● Шаблон иерархических данных

Так же как и шаблоны данных, шаблон иерархических данных позволяет описать разметку для объекта, который представляет запись в иерархическом списке.

Объект, участвующий в иерархическом списке, имеет свойство типа коллекции себе подобных элементов.

За описание шаблона для иерархического объекта отвечает элемент **HierarchicalDataTemplate**. Шаблон **HierarchicalDataTemplate** обрабатывается в основном элементе управления **TreeView**, где каждый элемент может содержать в себе набор других элементов, которые так же могут представлять иерархический объект. Если **DataTemplate** описывает обычный объект, то **HierarchicalDataTemplate** определяет дополнительное свойство – **ItemsSource**, указывающее, что объект содержит свойство списка себе подобных. Он как бы ожидает это свойство списка от объекта, так как в дальнейшем используется в **TreeView**.

Для примера определим список папок и файлов, но для начала определим модель представления данных.

```

Ссылка: 5
interface INode //Интерфейс для записи в папке
{
    Ссылка: 2
    string Name { get; set; }
}

ссылка: 1
interface IFolder : INode //Интерфейс для папки
{
    Ссылка: 2
    List<INode> Nodes { get; set; }
}

Ссылка: 0
class FileNode : INode //Запись файла, реализующая общий интерфейс
{
    ссылка: 1
    public string Name { get; set; }
    Ссылка: 0
    public int Size { get; set; }
}

Ссылка: 0
class FolderNode : IFolder //Папка, содержащая записи
    //Этот класс представляет иерархический объект, потому что
    //содержит список себе подобных.
{
    //Список себе подобных записей
    Ссылка: 2
    public List<INode> Nodes { get; set; } = new List<INode>();
    ссылка: 1
    public string Name { get; set; }
    Ссылка: 0
    public int CountFile => Nodes.Count;
}

```

Рисунок 30

После можно определить список. Для наглядности, определим его в ресурсах окна.


```

<x:Array Type="{x:Type local:INode}" x:Key="FileList">
  <local:FileNode Name="Мой файл 1"
    Size="15"/>
  <local:FileNode Name="Мой файл 2"
    Size="15"/>
  <local:FolderNode Name="Моя папка1">
    <local:FolderNode.Nodes>
      <local:FileNode Name="Мой файл 3"
        Size="15"/>
      <local:FileNode Name="Мой файл 4"
        Size="15"/>
      <local:FolderNode Name="Моя папка2">
        <local:FolderNode.Nodes>
          <local:FileNode Name="Мой файл 5"
            Size="15"/>
          <local:FileNode Name="Мой файл 6"
            Size="15"/>
        </local:FolderNode.Nodes>
      </local:FolderNode>
    </local:FolderNode.Nodes>
  </local:FolderNode>
</x:Array>

```

Рисунок 31

Для оформления объекта типа **FileNode** используем **DataTemplate**, так как он не представляет иерархичность (отсутствует свойство коллекции себе подобных объектов).

```

<DataTemplate DataType="{x:Type local:FileNode}">
  <DockPanel Height="25">
    <Image DockPanel.Dock="Left"
      Source="files.png"/>
    <TextBlock Text="{Binding Path=Name}"/>
    <TextBlock xml:space="preserve"
      Text="{Binding Path=Size, StringFormat='{0} ({0} Mbyte)'}"/>
  </DockPanel>
</DataTemplate>

```

Рисунок 32

Для определения объекта иерархического типа, воспользуемся **HierarchicalDataTemplate**.


```

class FolderNode : IFolder //папка, содержащая записи
//Этот класс представляет иерархический объект, потому что
//содержит список себе подобных
{
    //Список себе подобных записей
    public List<INode> Nodes { get; set; } = new List<INode>();
    public string Name { get; set; }
    public int CountFile => Nodes.Count;
}

```

```

<HierarchicalDataTemplate DataType="{x:Type local:FolderNode}"
    ItemsSource="{Binding Path=Nodes}">
    <DockPanel Height="25">
        <Image DockPanel.Dock="Left"
            Source="folder.png"/>
        <TextBlock Text="{Binding Path=Name}"/>
        <TextBlock xml:space="preserve"
            Text="{Binding Path=CountFile, StringFormat='{ } ({0} файлов)}"/>
    </DockPanel>
</HierarchicalDataTemplate>

```

Рисунок 33

Теперь можно определить разметку **TreeView** на форме.

```

<TreeView ItemsSource="{Binding Source={StaticResource FileList}}"/>

```

Рисунок 34

После запуска отобразится следующее.

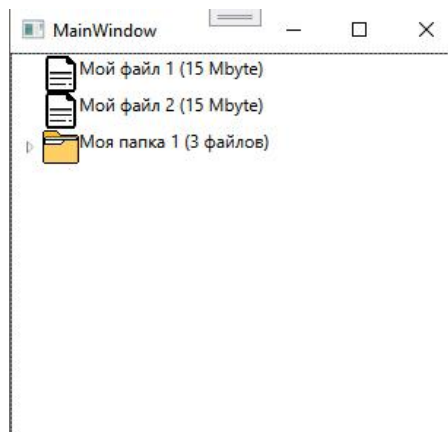


Рисунок 35

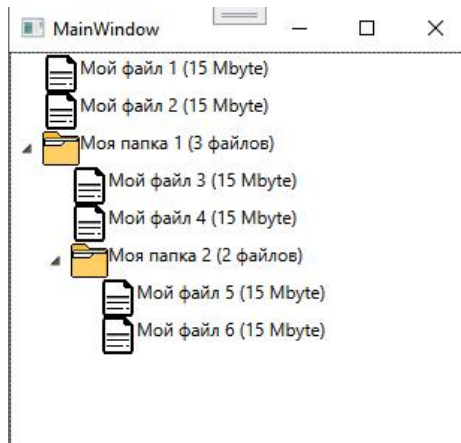


Рисунок 36

MultiBinding – привязка из нескольких источников.

Обычно, элемент **Binding** определяет только одно свойство в качестве источника данных. Это правильно, но иногда бывают ситуации, когда необходима привязка к нескольким источникам., однако используется крайне редко. Такое может происходить в случаях, когда используются конвертеры значений. Частный случай конвертера – свойство **StringFormat** у **Binding**. На самом деле он может принимать несколько параметров.

Рассмотрим на примере. Необходимо в одной строке определить несколько параметров.

```
<TextBlock>
  <TextBlock.Resources>
    <local:Student Name="Вася" Age="15"
      x:Key="Vasya"/>
  </TextBlock.Resources>
  <TextBlock.Text>
    <MultiBinding StringFormat="{0} Это {0}. И ему {1} лет">
      <Binding Source="{StaticResource Vasya}" Path="Name"/>
      <Binding Source="{StaticResource Vasya}" Path="Age"/>
    </MultiBinding>
  </TextBlock.Text>
</TextBlock>
```

Рисунок 37

Здесь можно заметить, что **MultiBinding** как будто образует список, к каждому элементу которого можно получить доступ по индексу.