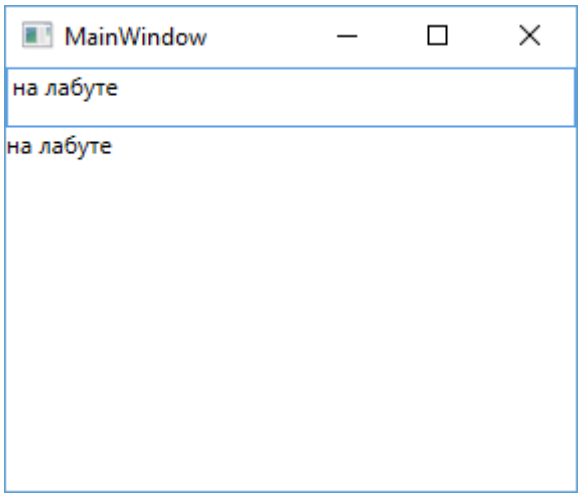
Привязка

Введение в привязку данных

В WPF привязка (binding) является мощным инструментом программирования, без которого не обходится ни одно серьезное приложение.

Привязка подразумевает взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника. В случае модификации объекта-источника, объект-приемник также будет модифицирован. Например, простейшая форма с использованием привязки:

```
<Window x:Class="BindingApp.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:BindingApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300">
    <StackPanel>
        <TextBox x:Name="myTextBox" Height="30" />
        <TextBlock x:Name="myTextBlock" Text="{Binding ElementName=myTextBox,Path=Text}" Height=
    </StackPanel>
</Window>
```



Для определения привязки используется выражение типа:

```
{Binding ElementName=Имя_объекта-источника, Path=Свойство_объекта-источника}
```

То есть в данном случае у нас элемент TextBox является источником, а TextBlock - приемником привязки. Свойство Text элемента TextBlock привязывается к свойству Text элемента TextBox. В итоге при осуществлении ввода в текстовое поле синхронно будут происходить изменения в текстовом блоке.

Работа с привязкой в C#

Ключевым объектом при создании привязки является объект **System.Windows.Data.Binding**. Используя этот объект мы можем получить уже имеющуюся привязку для элемента:

```
Binding binding = BindingOperations.GetBinding(myTextBlock, TextBlock.TextProperty);
```

В данном случае получаем привязку для свойства зависимостей TextProperty элемента myTextBlock.

Также можно полностью установить привязку в коде C#:

```
public MainWindow()
{
    InitializeComponent();


    Binding binding = new Binding();

    binding.ElementName = "myTextBox"; // элемент-источник
    binding.Path = new PropertyPath("Text"); // свойство элемента-источника
    myTextBlock.SetBinding(TextBlock.TextProperty, binding); // установка привязки для элемента-
}
```

Если в дальнейшем нам станет не нужна привязка, то мы можем воспользоваться классом **BindingOperations** и его методами **ClearBinding()** (удаляет одну привязку) и **ClearAllBindings()** (удаляет все привязки для данного элемента)

```
BindingOperations.ClearBinding(myTextBlock, TextBlock.TextProperty);
```

или

```
BindingOperations.ClearAllBindings(myTextBlock);
```

Некоторые свойства класса **Binding**:

- **ElementName**: имя элемента, к которому создается привязка
- **IsAsync**: если установлено в True, то использует асинхронный режим получения данных из объекта. По умолчанию равно False
- **Mode**: режим привязки
- **Path**: ссылка на свойство объекта, к которому идет привязка
- **TargetNullValue**: устанавливает значение по умолчанию, если привязанное свойство источника привязки имеет значение null
- **RelativeSource**: создает привязку относительно текущего объекта
- **Source**: указывает на объект-источник, если он не является элементом управления.
- **XPath**: используется вместо свойства path для указания пути к xml-данным

Режимы привязки

Свойство **Mode** объекта Binding, которое представляет режим привязки, может принимать следующие значения:

- **OneWay**: свойство объекта-приемника изменяется после модификации свойства объекта-источника.
- **OneTime**: свойство объекта-приемника устанавливается по свойству объекта-источника только один раз. В дальнейшем изменения в источнике никак не влияют на объект-приемник.
- **TwoWay**: оба объекта - применки и источник могут изменять привязанные свойства друг друга.
- **OneWayToSource**: объект-приемник, в котором объявлена привязка, меняет объект-источник.
- **Default**: по умолчанию (если меняется свойство TextBox.Text, то имеет значение TwoWay, в остальных случаях OneWay).

Применение режима привязки:

```
<StackPanel>
    <TextBox x:Name="textBox1" Height="30" />
    <TextBox x:Name="textBox2" Height="30" Text="{Binding ElementName=textBox1, Path=Text, Mode=
</StackPanel>
```

Обновление привязки. UpdateSourceTrigger

Односторонняя привязка от источника к приемнику практически мгновенно изменяет свойство приемника. Но если мы используем двустороннюю привязку в случае с текстовыми полями (как в примере выше), то при изменении приемника свойство источника не изменяется мгновенно. Так, в примере выше, чтобы текстовое поле-источник изменилось, нам надо перевести фокус с текстового поля-приемника. И в данном случае в дело вступает свойство **UpdateSourceTrigger** класса Binding, которое задает, как будет присходить обновление. Это свойство в качестве принимает одно из значений перечисления **UpdateSourceTrigger**:

- PropertyChanged: источник привязки обновляется сразу после обновления свойства в приемнике
- LostFocus: источник привязки обновляется только после потери фокуса приемником
- Explicit: источник не обновляется до тех пор, пока не будет вызван метод BindingExpression.UpdateSource()
- Default: значение по умолчанию. Для большинства свойств это значение PropertyChanged. А для свойства Text элемента TextBox это значение LostFocus

В данном случае речь идет об обновлении источника привязки после изменения приемника в режимах OneWayToSource или TwoWay. То есть чтобы у нас оба текстовых поля, которые связаны режимом TwoWay, моментально обновлялись после изменения одного из них, надо использовать значение UpdateSourceTrigger.PropertyChanged:

```
<StackPanel>
    <TextBox x:Name="textBox1" Height="30" />
    <TextBox x:Name="textBox2" Height="30"
        Text="{Binding ElementName=textBox1, Path=Text, Mode=TwoWay, UpdateSourceTrigger=Propert
</StackPanel>
```

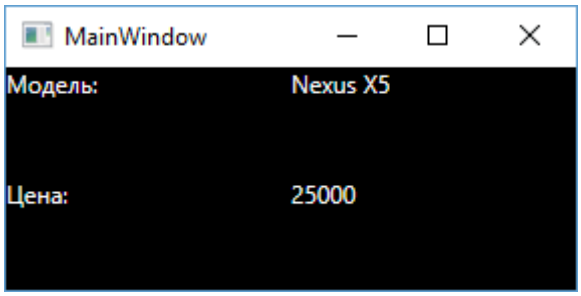
Свойство Source

Свойство Source позволяет установить привязку даже к тем объектам, которые не являются элементами управления WPF. Например, определим класс Phone:

```
class Phone
{
    public string Title { get; set; }
    public string Company { get; set; }
    public int Price { get; set; }
}
```

Теперь создадим объект этого класса и определим к нему привязку:

```
<Window x:Class="BindingApp.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:BindingApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="150" Width="300">
    <Window.Resources>
        <local:Phone x:Key="nexusPhone" Title="Nexus X5" Company="Google" Price="25000" />
    </Window.Resources>
    <Grid Background="Black">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <TextBlock Text="Модель:" Foreground="White"/>
        <TextBlock x:Name="titleTextBlock" Text="{Binding Source={StaticResource nexusPhone}, Pa
                   Foreground="White" Grid.Column="1"/>
        <TextBlock Text="Цена:" Foreground="White" Grid.Row="1"/>
        <TextBlock x:Name="priceTextBlock" Text="{Binding Source={StaticResource nexusPhone}, Pa
                   Foreground="White" Grid.Column="1" Grid.Row="1"/>
    </Grid>
</Window>
```



Свойство TargetNullValue

На случай, если свойство в источнике привязки вдруг имеет значение null, то есть оно не установлено, мы можем задать некоторое значение по умолчанию. Например:

```
<Window.Resources>
    <local:Phone x:Key="nexusPhone" Company="Google" Price="25000" />
</Window.Resources>
<StackPanel>
    <TextBlock x:Name="titleTextBlock"
        Text="{Binding Source={StaticResource nexusPhone}, Path=Title, TargetNullValue=Текст по
</StackPanel>
```

В данном случае у ресурса nexusPhone не установлено свойство Title, поэтому текстовый блок будет выводить значение по умолчанию, указанное в параметре TargetNullValue.

Свойство RelativeSource

Свойство **RelativeSource** позволяет установить привязку относительно элемента-источника, который связан какими-нибудь отношениями с элементом-приемником. Например, элемент-источник может быть одним из внешних контейнеров для элемента-приемника. Либо источником и приемником может быть один и тот же элемент.

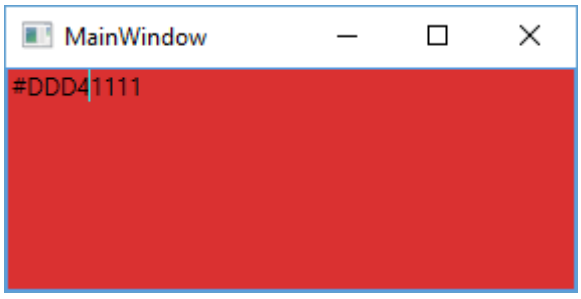
Для установки этого свойства используется одноименный объект **RelativeSource**. У этого объекта есть свойство **Mode**, которое задает способ привязки. Оно принимает одно из значений перечисления **RelativeSourceMode**:

Self: привязка осуществляется к свойству этого же элемента. То есть элемент-источник привязки в то же время является и приемником привязки.

FindAncestor: привязка осуществляется к свойству элемента-контейнера.

Например, совместим источник и приемник привязке в самом элементе:

```
<TextBox Text="{Binding RelativeSource={RelativeSource Mode=Self}, Path=Background, Mode=TwoWay,
```



Здесь текст и фоновый цвет текстового поля связаны двусторонней привязкой. В итоге мы можем увидеть в поле числовое значение цвета, поменять его, и вместе с ним изменится и фон поля.

Привязка к свойствам контейнера:

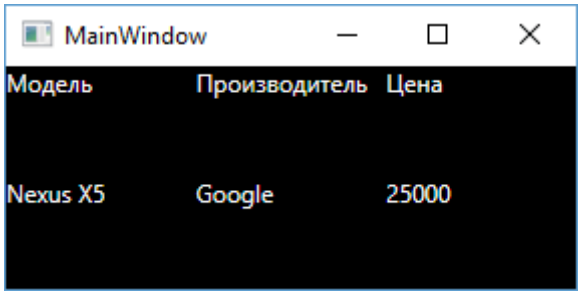
```
<Grid Background="Black">
    <TextBlock Foreground="White"
        Text="{Binding RelativeSource={RelativeSource Mode=FindAncestor,AncestorType={x:Type Gri
</Grid>
```

При использовании режима FindAncestor, то есть привязке к контейнеру, необходимо еще указывать параметр **AncestorType** и передавать ему тип контейнера в виде выражения AncestorType={x:Type Тип_элемента-контейнера}. При этом в качестве контейнера мы могли бы выбрать любой контейнер в дереве элементов, в частности, в данном случае кроме Grid таким контейнером также является элемент Window.

Свойство DataContext

У объекта FrameworkElement, от которого наследуются элементы управления, есть интересное свойство **DataContext**. Оно позволяет задавать для элемента и вложенных в него элементов некоторый контекст данных. Тогда вложенные элементы могут использовать объект Binding для привязки к конкретным свойствам этого контекста. Например, используем ранее определенный класс Phone и создадим контекст данных из объекта этого класса:

```
<Window x:Class="BindingApp.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:BindingApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="150" Width="300">
    <Window.Resources>
        <local:Phone x:Key="nexusPhone" Title="Nexus X5" Company="Google" Price="25000" />
    </Window.Resources>
    <Grid Background="Black" DataContext="{StaticResource nexusPhone}" TextBlock.Foreground="Whi
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <TextBlock Text="Модель" />
        <TextBlock Text="{Binding Title}" Grid.Row="1" />
        <TextBlock Text="Производитель" Grid.Column="1"/>
        <TextBlock Text="{Binding Company}" Grid.Column="1" Grid.Row="1" />
        <TextBlock Text="Цена" Grid.Column="2" />
        <TextBlock Text="{Binding Price}" Grid.Column="2" Grid.Row="1" />
    </Grid>
</Window>
```



Таким образом мы задаем свойству DataContext некоторый динамический или статический ресурс. Затем осуществляем привязку к этому ресурсу.

Интерфейс INotifyPropertyChanged

В прошлой теме использовался объект Phone для привязки к текстовым блокам. Однако если мы изменим его, содержимое текстовых блоков не изменится. Например, добавим в окно

приложения кнопку:

```
<Window x:Class="BindingApp.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:BindingApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="150" Width="300">
    <Window.Resources>
        <local:Phone x:Key="nexusPhone" Title="Nexus X5" Company="Google" Price="25000" />
    </Window.Resources>
    <Grid Background="Black" DataContext="{StaticResource nexusPhone}" TextBlock.Foreground="Whi
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <TextBlock Text="Модель" />
        <TextBlock Text="{Binding Title}" Grid.Row="1" />
        <TextBlock Text="Производитель" Grid.Column="1"/>
        <TextBlock Text="{Binding Company}" Grid.Column="1" Grid.Row="1" />
        <TextBlock Text="Цена" Grid.Column="2" />
        <TextBlock Text="{Binding Price}" Grid.Column="2" Grid.Row="1" />

        <Button Foreground="White" Content="Изменить" Click="Button_Click" Background="Black"
            BorderBrush="Silver" Grid.Column="2" Grid.Row="2" />
    </Grid>
</Window>
```

И в файле кода для этой кнопки определим обработчик, в котором будет меняться свойства ресурса:

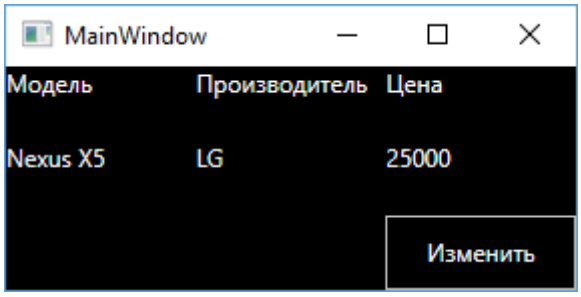
```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Phone phone = (Phone)this.Resources["nexusPhone"];
    phone.Company = "LG"; // Меняем с Google на LG
}
```

Сколько бы мы не нажимали на кнопку, текстовые блоки, привязанные к ресурсу, не изменятся. Чтобы объект мог полноценно реализовать механизм привязки, нам надо реализовать в его

классе интерфейс INotifyPropertyChanged. И для этого изменим класс Phone следующим образом:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

class Phone : INotifyPropertyChanged
{
    private string title;
    private string company;
    private int price;

    public string Title
    {
        get { return title; }
        set
        {
            title = value;
            OnPropertyChanged("Title");
        }
    }
    public string Company
    {
        get { return company; }
        set
        {
            company = value;
            OnPropertyChanged("Company");
        }
    }
    public int Price
    {
        get { return price; }
        set
        {
            price = value;
            OnPropertyChanged("Price");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged([CallerMemberName]string prop = "")
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(prop));
    }
}
```

Когда объект класса изменяет значение свойства, то он через событие PropertyChanged извещает систему об изменении свойства. А система обновляет все привязанные объекты.



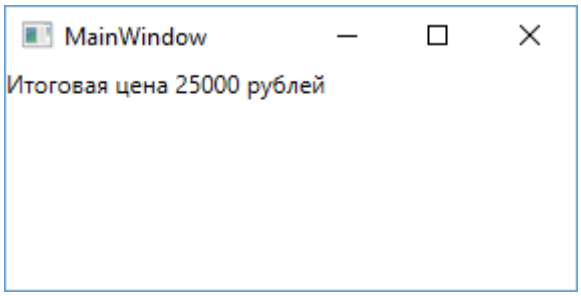
Форматирование значений привязки и конвертеры значений

Привязка представляет очень простой механизм, однако иногда этому механизму требуется некоторая кастомизация. Так, нам может потребоваться небольшое форматирование значение. Для примера возьмем класс Phone из прошлых тем:

```
class Phone
{
    public string Title { get; set; }
    public string Company { get; set; }
    public int Price { get; set; }
}
```

Допустим, нам надо в текстовый блок вывести не только цену, но и еще какой-нибудь текст:

```
<Window x:Class="ValueConventerApp.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:ValueConventerApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="150" Width="300">
    <Window.Resources>
        <local:Phone x:Key="nexusPhone" Title="Nexus X5" Company="Google" Price="25000" />
    </Window.Resources>
    <Grid>
        <TextBlock Text="{Binding StringFormat=Итоговая цена {0} рублей, Source={StaticResource
    </Grid>
</Window>
```



Свойство **StringFormat** получает набор параметров в фигурных скобках. Фигурные скобки ({0}) передают собственно то значение, к которому идет привязка. Можно сказать, что действие свойства StringFormat аналогично методу String.Format(), который выполняет форматирование строк.

При необходимости мы можем использовать дополнительные опции форматирования, например, {0:C} для вывода валюты, {0:P} для вывода процентов и т.д.:

```
<TextBlock Text="{Binding StringFormat={}{0:C}, Source={StaticResource nexusPhone}, Path=Price}"
```

При этом если у нас значение в StringFormat начинается с фигурных скобок, например, "{0:C}", то перед ними ставятся еще пара фигурных скобок, как в данном случае. По сути они ничего важно не несут, просто служат для корректной интерпретации строки.

Либо в этом случае нам надо экранировать скобки слешами:

```
<TextBlock Text="{Binding StringFormat=\{0:C\}, Source={StaticResource nexusPhone}, Path=Price}"
```

В зависимости от типа элемента доступны различные типы форматировщиков значений:

- **StringFormat**: используется для класса Binding
- **ContentStringFormat**: используется для классов ContentControl, ContentPresenter, TabControl
- **ItemStringFormat**: используется для класса ItemsControl
- **HeaderStringFormat**: используется для класса HeaderContentControl
- **ColumnHeaderStringFormat**: используется для классов GridView, GridViewHeaderRowPresenter
- **SelectionBoxItemStringFormat**: используется для классов ComboBox, RibbonComboBox

Их применение аналогично. Например, так как Button представляет ContentControl, то для этого элемента надо использовать ContentStringFormat:

```
<Button ContentStringFormat="{}{0:C}"
    Content="{Binding Source={StaticResource nexusPhone}, Path=Price}" />
```

Конвертеры значений

Конвертеры значений (value converter) также позволяют преобразовать значение из источника привязки к типу, который понятен приемнику привязки. Так как не всегда два связываемых привязкой свойства могут иметь совместимые типы. И в этом случае как раз и нужен конвертер значений.

Допустим, нам надо вывести дату в определенном формате. Для этой задачи создадим в проекте класс конвертера значений:

```
using System;
using System.ComponentModel;
using System.Globalization;
using System.Windows.Data;

public class DateTimeToDateConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return ((DateTime)value).ToString("dd.MM.yyyy");
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo cultu
    {
        return DependencyProperty.UnsetValue;
    }
}
```

Конвертер значений должен реализовать интерфейс **System.Windows.Data.IValueConverter**. Этот интерфейс определяет два метода: Convert(), который преобразует пришедшее от привязки значение в тот тип, который понимается приемником привязки, и ConvertBack(), который выполняет противоположную операцию.

Оба метода принимают четыре параметра:

- object value: значение, которое надо преобразовать
- Type targetType: тип, к которому надо преобразовать значение value
- object parameter: вспомогательный параметр
- CultureInfo culture: текущая культура приложения

В данном случае метод Convert возвращает строковое представление даты в формате "dd.MM.yyyy". То есть мы ожидаем, что в качесве параметра value будет передаваться объект DateTime.

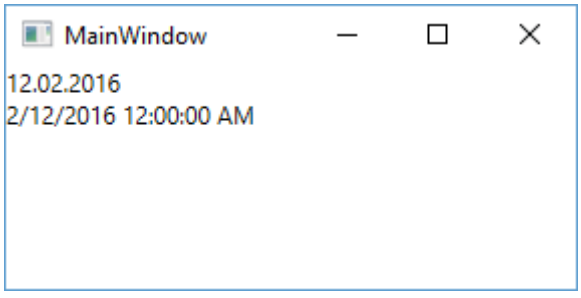
Метод ConvertBack в данном случае не имеет значения, поэтому он просто возвращает пустое значение для свойста. В другоим случае мы бы здесь получали строковое значение и преобразовывали его в DateTime.

Теперь применим этот конвертер в xaml:

```
<Window x:Class="ValueConventerApp.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:ValueConventerApp"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        mc:Ignorable="d"
        Title="MainWindow" Height="150" Width="300">
    <Window.Resources>
        <sys:DateTime x:Key="myDate">2/12/2016</sys:DateTime>
        <local:DateTimeToDateConverter x:Key="myDateConverter" />
    </Window.Resources>
    <StackPanel>
        <TextBlock Text="{Binding Source={StaticResource myDate},Converter={StaticResource myDat
        <TextBlock Text="{Binding Source={StaticResource myDate}}" />
    </StackPanel>
</Window>
```

Здесь искомая дата, которая выводится в текстовые блоки, задана в ресурсах. Также в ресурсах задан конвертер значений. Чтобы применить этот конвертер в конструкции привязки используется параметр **Converter** с указанием на ресурс: Converter={StaticResource myDateConverter}

Для сравнения я здесь определил два текстовых блока. Но поскольку к одному из них применяется конвертер, то отображение даты будет отличаться:



Немного изменим код конвертера и используем передаваемый параметр:

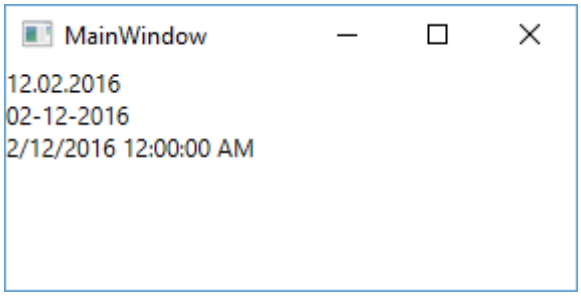
```
public class DateTimeToDateConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        if(parameter!=null && parameter.ToString()=="EN")
            return ((DateTime)value).ToString("MM-dd-yyyy");

        return ((DateTime)value).ToString("dd.MM.yyyy");
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo cultu
    {
        return DependencyProperty.UnsetValue;
    }
}
```

В качестве параметра может передаваться любой объект. Если параметр в xaml не используется, то передается null. В данном случае мы проверяем, равен ли параметр строке "EN", то есть мы ожидаем, что параметр будет передавать строковое значение. И если равен, то возвращаем дату немного в другом формате.

Для применения параметра изменим код xaml:

```
<StackPanel>
    <TextBlock Text="{Binding Source={StaticResource myDate},Converter={StaticResource myDateCor
    <TextBlock Text="{Binding Source={StaticResource myDate}, ConverterParameter=EN, Converter={
    <TextBlock Text="{Binding Source={StaticResource myDate}}" />
</StackPanel>
```

Параметр привязки задается с помощью свойства ConverterParameter. Итак, у нас тут три текстовых блока, и применяя конвертер, мы получим три разных отображения даты:



Также мы можем использовать передаваемые в конвертер параметры культуры и типа, к которому надо преобразовать. Например, мы можем смотреть на тип целевого значения и в зависимости от результатов производить определенные действия:

```
public class DateTimeToDateConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        if (targetType != typeof(Brush))
        {
            //....
        }
//..................
```

В данном случае предполагается, что тип объекта, к которому надо преобразовать, представляет тип Brush.