Windows Presentation Foundation

Урок №3

Отображение документов нефиксированного формата.

Соединение данных и элементов управления.

Содержание

1. Элементы управления
    1. TabControl
    2. ToolBar
    3. ToolBarTray
    4. Практические примеры использования
2. Отображение документов нефиксированного формата
    1. Введение в документы нефиксированного формата
    2. Отображение документов:
    3. Компоновка документа
    4. Практический пример использования
3. Соединение данных и элементов управления
    1. Общие принципы.
    2. Схема взаимодействия источника и приёмника данных
    3. Режимы связывания данных
    4. Класс Binding
    5. Примеры связывания данных
        - Связывание со свойством класса
        - Связывание с объектом класса
        - Связывание с XML данными
        - Связывание с элементом управления
4. Управление стилями и ресурсами:
    1. Стили
    2. Шаблоны
    3. Скины
    4. Темы
    5. Практические примеры использования

1. Элементы управления

Элемент управления содержимым TabControl

TabControl представляет элемент управления, содержащий несколько элементов, которые совместно используют одно пространство на экране. Объект **TabControl** можно использовать, чтобы сократить использование пространства экрана и в то же время сделать так, чтобы приложение отображало большой объем данных. Объект **TabControl** состоит из нескольких объектов **TabItem**, которые совместно используют одно пространство на экране. Одновременно отображается только один элемент **TabItem** из **TabControl**. Когда пользователь выбирает вкладку элемента **TabItem**, содержимое данного объекта **TabItem** становится видимым, а содержимое других объектов **TabItem** скрывается.

Модель содержимого: элемент управления **TabControl** относится к типу **ItemsControl**. Его свойствами содержимого являются коллекция **Items** и **ItemsSource**.

```
[StyleTypedPropertyAttribute(Property = L"ItemContainerStyle",
                             StyleTargetType = typeof(TabItem))]
[TemplatePartAttribute(Name = L"PART_SelectedContentHost",
                       Type = typeof(ContentPresenter))]
public ref class TabControl : public Selector
```

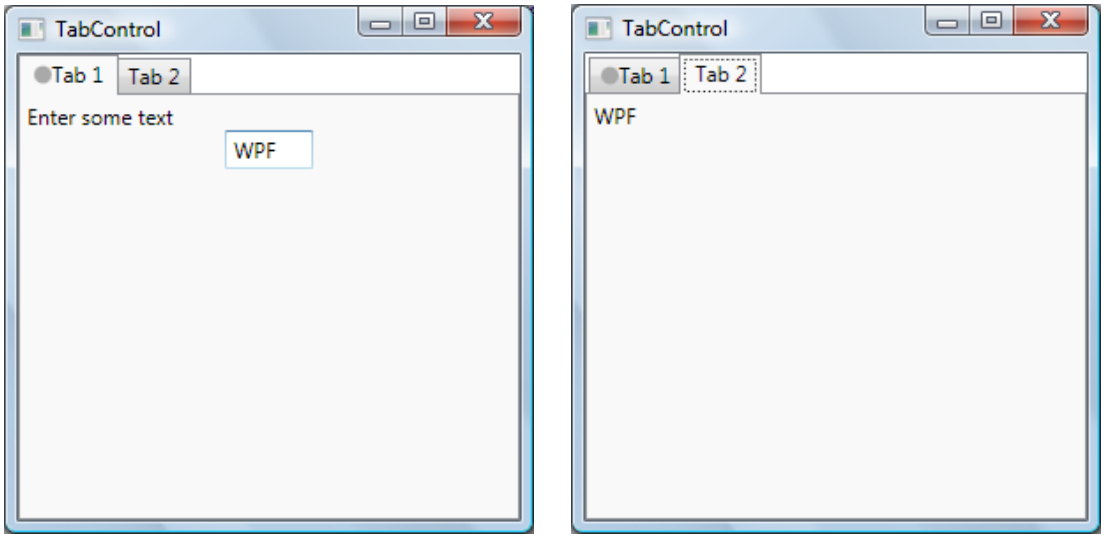
Или в виде разметки элемента объекта XAML:

```
<TabControl>
       Items
</TabControl>
```

Рассмотрим следующий пример: создается объект **TabControl** и объект **TextBlock** во втором элементе **TabItem** привязывается к объекту **TextBox** в первом элементе **TabItem**. Более подробно о привязке будет рассказано в следующих частях урока:

```
<TabControl>
            <TabItem>
                <TabItem.Header>
                    <StackPanel Orientation="Horizontal">
                        <Ellipse Width="10" Height="10" Fill="DarkGray"/>
                        <TextBlock>Tab 1</TextBlock>
                    </StackPanel>
                </TabItem.Header>
                <StackPanel>
                    <TextBlock>Enter some text</TextBlock>
                    <TextBox Name="textBox1" Width="50"/>
                </StackPanel>
            </TabItem>
            <TabItem Header="Tab 2">
                <TextBlock Text="{Binding ElementName=textBox1, Path=Text}"/>
            </TabItem>
</TabControl>
```

Вот результат выполнения примера – ввод текста на первой вкладке и его отображение на второй:



TabControl содержит коллекцию элементов **TabItem**.

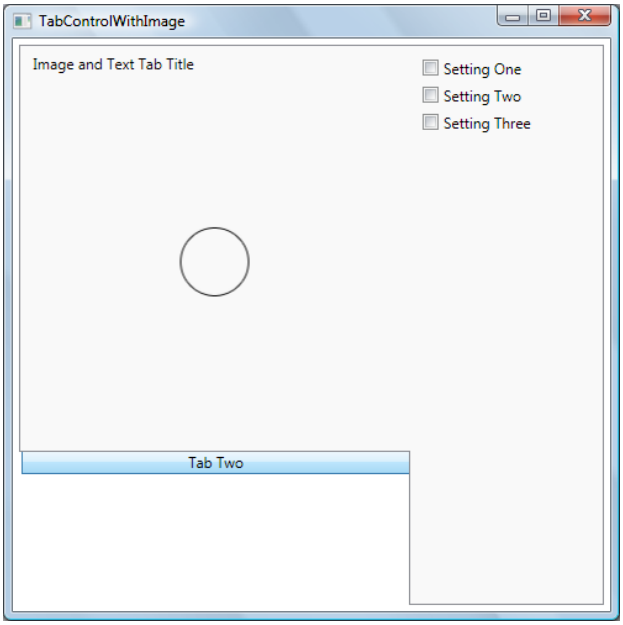
Одним из наследников класса **ContentControl** является класс **HeaderedContentControl**. Его роль простая - он представляет контейнер, который может иметь как одноэлементное содержимое (хранится в свойстве **Content**), так и одноэлементный заголовок

(хранится в свойстве **Header**). У класса **HeaderedContentControl** есть три наследника: **GroupBox**. **TabItem** и **Expander**. **TabItem** представляет страницу в элементе управления **TabControl**. Класс **TabItem** добавляет одно важное свойство **IsSelected**, которое показывает, отображается ли в данный момент вкладка в элементе управления **TabControl**.

Свойство **TabStripPlacement** можно использовать для того, чтобы вкладки отображались сбоку элемента **TabControl**, а не сверху, как это обычно бывает. Как и свойство **Content**, свойство **Header** может принимать любой тип объекта. Оно отображает классы-наследники **UIElement**, визуализируя их и используя метод **ToString()** для внутристрочного текста и всех других объектов. Это означает, что можно создать групповое окно или вкладку с графическим содержимым или произвольными элементами в заголовке. Рассмотрим пример:

```
<TabControl Margin="5" TabStripPlacement="Left" >
            <TabItem>
                <TabItem.Header>
                    <StackPanel>
                      <TextBlock Margin="3" >
                              Image and Text Tab Title
                      </TextBlock>
                        <Image Source="pic.gif" Stretch="None" />
                    </StackPanel>
                    </TabItem.Header>
                    <StackPanel Margin="3">
                        <CheckBox Margin="3">Setting One</CheckBox>
                        <CheckBox Margin="3">Setting Two</CheckBox>
                        <CheckBox Margin="3">Setting Three</CheckBox>
                    </StackPanel>
            </TabItem>
            <TabItem Header="Tab Two"></TabItem>
        </TabControl>
```

Результат будет выглядеть следующим образом:



Панель инструментов **ToolBar**

Панели инструментов являются одним из главных компонентов в Windows. Они представляют собой специализированные контейнеры, способные хранить коллекции элементов. Панели инструментов обычно состоят из кнопок, однако могут использоваться с множеством различных элементов управления.

В **Windows Forms** у панелей инструментов имеется своя собственная модель содержимого. И хотя размещать внутри них произвольные элементы управления с помощью упаковщика допускается, простым этот процесс никак не назовешь. Новая модель содержимого в WPF значительно улучшает эту ситуацию. WPF-класс **ToolBar** поддерживает все элементы WPF. тем самым предоставляя разработчиками ни с чем несравнимую гибкость. На самом деле, никаких элементов, предназначенных специально для панелей инструментов, не существует. Все необходимое уже доступно в базовой коллекции WPF элементов.

Обычно WPF-класс **ToolBar** заполняется объектами **Button**, **ComboBox, CheckBox, RadioButton** и **Separator**. Поскольку все эти элементы (кроме **Separator**) являются элементами управления содержимым, внутри них можно размещать текстовое и графическое

содержимое. Хотя допускается использовать и другие элементы, подобные **Label** и **Image**, для добавления в **ToolBar** неинтерактивных элементов, эффект зачастую оказывается неоднозначным.

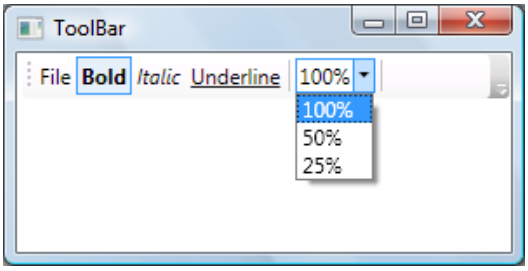
С точки зрения WPF. кнопка в панели инструментов практически ничем не отличается от кнопки в окне: обе представляют собой активизируемые щелчком области, которые разработчик может использовать для выполнения какого-нибудь действия. Разница состоит лишь в визуальном внешнем виде. Поэтому идеальное решение — использовать существующий класс **Button** и просто настроить должным образом различные свойства или изменить шаблон элемента управления. Именно это как раз и делает класс **ToolBar**: он переопределяет стиль, используемый по умолчанию для потомков некоторых типов, включая кнопки. Разработчик все равно может "сказать свое последнее слово", вручную установив свойство **ButtonStyle**, если хочет создать свою собственную специальную кнопку для панели инструментов, но обычно добиться необходимого результата можно и просто за счет установки содержимого кнопки.

Класс ToolBar изменяет не только внешний вид многих из размещаемых в нем элементов управления, но и также поведение унаследованных от него классов **ToggleButton**, **Checkbox** и **RadioButton**. Элементы **ToggleButton** и **CheckBox** в **ToolBar** визуализируются как обычные кнопки, но при выполнении на них щелчка остаются выделенными (до тех пор, пока на них не щелкнут снова). **RadioButton** имеет похожий внешний вид, но снять выделение с этого элемента можно, только щелкнув на другом таком же элементе в группе. (Во избежание путаницы, группу объектов **RadioButton** в панели инструментов всегда лучше отделять с помощью **Separator**.) Продемонстрируем это на следующем примере:

```
<ToolBar VerticalAlignment="Top">
            <Button Content="File"/>
```

```
            <CheckBox FontWeight="Bold">Bold</CheckBox>
            <CheckBox FontStyle="Italic">Italic</CheckBox>
            <CheckBox>
                <TextBlock TextDecorations="Underline">Underline</TextBlock>
            </CheckBox>
            <Separator/>
            <ComboBox SelectedIndex="0">
                <ComboBoxItem>100%</ComboBoxItem>
                <ComboBoxItem>50%</ComboBoxItem>
                <ComboBoxItem>25%</ComboBoxItem>
            </ComboBox>
            <Separator/>
</ToolBar>
```

Результат будет иметь следующий вид:



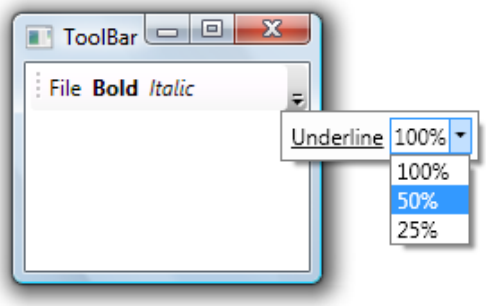
Обычно кнопки в **ToolBar** включают графическое содержимое. Текстовое и графическое содержимое также можно комбинировать, упаковав элементы **Image** и **TextBlock** или **Label** в элемент **StackPanel** с горизонтальной ориентацией. В качестве графического содержимого могут использоваться растровые изображения, пиктограммы и векторные изображения.

У элемента управления **ToolBar** имеется несколько странностей. Во-первых, в отличие от других элементов управления, которые наследуются от **ItemsControl**. он не предоставляет отдельного класса упаковщика. **ToolBar** просто не требуется такой упаковщик для управления элементами, отслеживания выбора и т.д., как другим списковым элементам управления. Другой странностью **ToolBar** является то, что он унаследован от **HeaderedItemsControl**. хотя свойство **Header**

не имеет никакого эффекта. Как использовать это свойство, должен решать разработчик.

У **ToolBar** есть еще одно интересное свойство: **Orientation**. Путем установки для свойства **ToolBar.Orientation** значения **Vertical** можно создать панель инструментов, располагаемую сверху вниз и пристыковываемую к одной из сторон окна. Однако все элементы в этой панели все равно будут иметь горизонтальную ориентацию (например, текст не будет развернут соответствующим образом), если только для их переворачивания не использовать **LayoutTransform**.

Если в панели инструментов находится больше содержимого, чем может уместиться в окне, лишние элементы удаляются и помещаются в дополнительное меню (**overflow menu**), увидеть которое можно, щелкнув на указывающей вниз стрелке в конце панели инструментов:



Элемент управления **ToolBar** добавляет элементы в дополнительное меню автоматически, начиная с конца (т.е. с последнего элемента). Однако это поведение можно настраивать, применяя к элементам в панели инструментов прикрепленное свойство **ToolBar.OverflowMode**. Значение **OverflowMode.Never** позволяет сделать так, чтобы элемент вообще никогда не размещался в дополнительном меню. **OverflowMode.AsNeeded** — так, чтобы элемент размещался в дополнительном меню только в случае нехватки места, а **OverflowMode.Always** — так, чтобы он всегда оставался в этом меню. В случае если контейнер панели инструментов (каковым обычно является окно) окажется меньше пространства, необходимого для отображения всех элементов **OverflowMode.Always**, не уместившиеся элементы

будут усечены по краям контейнера и станут недоступными для пользователя. Если в панели инструментов содержится более одного элемента **OverflowMode.AsNeeded**, тогда **ToolBar** сначала удаляет те элементы, что находятся в конце этой панели. Возможности назначения элементам панели инструментов относительных приоритетов не существует. Например, нельзя создать элемент, допускающий размещение в дополнительном меню, но не помещаемый туда до тех пор, пока туда не будут перемещены все остальные передвигаемые элементы. Также нельзя и создать кнопки, способные корректировать свой размер в соответствии с количеством доступного пространства.

Элемент управления ToolBarTray

Хотя можно добавлять в окно множество элементов управления **Toolbar** и управлять ими с помощью контейнера компоновки, в **WPF** есть класс, предназначенный специально для избавления разработчика от части работы, и называется он **ToolBarTray**. По сути. **ToolBarTray** удерживает коллекцию объектов **ToolBar** (которые предоставляются через свойство **ToolBars**).

Класс **ToolBarTray** упрощает размещение панелей инструментов на одной строке, также называемой полосой (**band**). Его можно сконфигурировать так, чтобы одни панели инструментов размещались на одной и той же полосе, а другие - на совершенно отдельных полосах. **ToolBarTray** отображает по всей площади **ToolBar** затененный фон. Но самым важным является то, что **ToolBarTray** дополнительно предоставляет поддержку для функции перетаскивания панелей инструментов. Если только для свойства **ToolBarTray.IsLocked** не устанавливается значение **true**, пользователь может переупорядочивать находящиеся в **ToolBarTray** панели инструментов, щелкая на специальном значке захвата с левой стороны. Панели инструментов можно передвигать как в пределах одной и той же полосы, так и перемещать на другие полосы. Однако перетаскивать панель

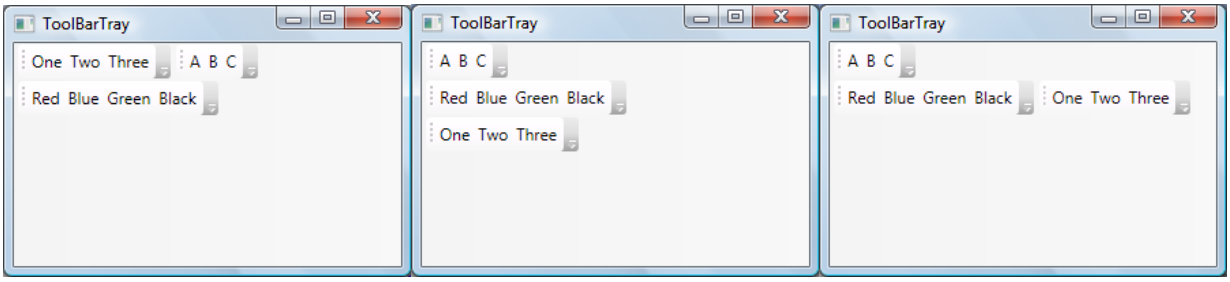
инструментов из одного элемента управления **ToolBarTray** в другой нельзя. При желании заблокировать возможность перемещения панелей инструментов нужно просто установить для соответствующих объектов **ToolBar** прикрепленное свойство **ToolBarTray.IsLocked**.

В **ToolBarTray** разрешено размещать столько объектов **ToolBar**, сколько нужно. По умолчанию все добавленные панели инструментов располагаются в самой верхней полосе в порядке слева направо. Первоначально каждая панель инструментов занимает всю необходимую ей ширину. Можно указать, какую именно полосу должна занимать данная панель инструментов, путем установки для свойства **Band** числового индекса (где 0 соответствует самой верхней полосе). Можно явно указать нужную позицию расположения внутри полосы с помощью свойства **BandIndex**. В случае установки для свойства **BandIndex** значения 0 панель инструментов размещается в начале полосы.

Рассмотрим вышесказанное на примере:

```
<ToolBarTray>
            <ToolBar>
                <Button>One</Button>
                <Button>Two</Button>
                <Button>Three</Button>
            </ToolBar>
            <ToolBar>
                <Button>A</Button>
                <Button>B</Button>
                <Button>C</Button>
            </ToolBar>
            <ToolBar Band="1">
                <Button>Red</Button>
                <Button>Blue</Button>
                <Button>Green</Button>
                <Button>Black</Button>
            </ToolBar>
        </ToolBarTray>
```

Вот результат группирования панелей инструментов в ToolBarTray:



2. Отображение документов нефиксированного формата

Введение в документы нефиксированного формата

Документы

Когда нам нужно вывести небольшой текст, мы используем **TextBlock** или **Label**. Но для показа большого объема информации, когда нужно, чтобы информация заняла все окно, эти элементы управления не очень подходят, и тогда разработчик использует документы.

При использовании документов появляются дополнительные возможности работы с содержимым, например масштабирование содержимого, различные режимы отображения или поиск по тексту.

В WPF документы бывают двух категорий:
- документы фиксированного формата (**fixed documents**);
- документы нефиксированного формата (**flow documents**).

Документы фиксированного формата

Документы фиксированного формата используются лишь в том случае, когда информация должна отображаться и на экране, и при печати в одном и том же виде. Например, есть строгая форма какого-то бланка и нельзя, чтобы слова переносились на другую строку. Такой режим отображения содержимого называется WYSIWYG («what you see is what you get» – «что видишь, то и получаешь»). Ярким примером этого

режима является отображение PDF-файлов – при печати мы получим то же самое, что видим на экране.

Для реализации документов фиксированного образца в WPF используется тип документа **FixedDocument**.

Для показа содержимого типа **FixedDocument** необходимо использовать контейнер **DocumentViewer**.

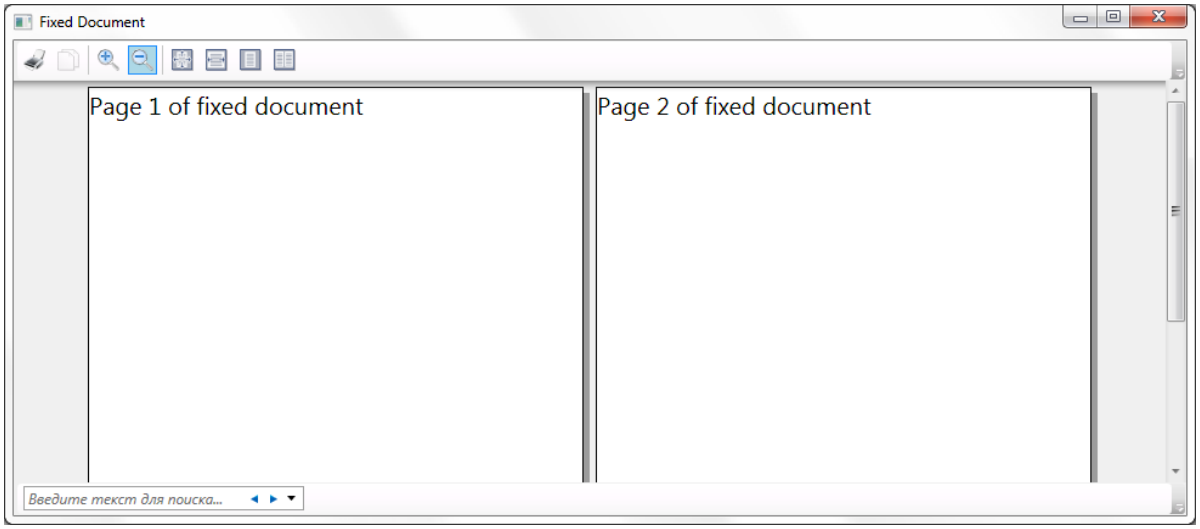
PageContent является единственным разрешенным дочерним элементом **FixedDocument**, т.е. кроме него ничего другого использовать в **FixedDocument** не получится. А для **PageContent** единственным разрешенным дочерним элементом является **FixedPage**, причем в одном элементе **PageContent** может содержаться только один элемент **FixedPage**. **FixedPage** может содержать в себе элементы, классы которых наследуются от **UIElement**.

Приведем небольшой пример работы **FixedDocument**.

Код на XAML:

```
<Window x:Class="Document.FixedDocumentWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="400"
    Height="200" Title="Fixed Document">

    <DocumentViewer>
        <FixedDocument>

            <PageContent>
                <FixedPage>
                   <TextBlock Text="Page 1 of fixed document" FontSize="40"/>
                </FixedPage>
            </PageContent>

            <PageContent>
                <FixedPage>
                   <TextBlock Text="Page 2 of fixed document" FontSize="40"/>
                </FixedPage>
            </PageContent>

        </FixedDocument>
    </DocumentViewer>
</Window>
```

Результат будет выглядеть следующим образом:



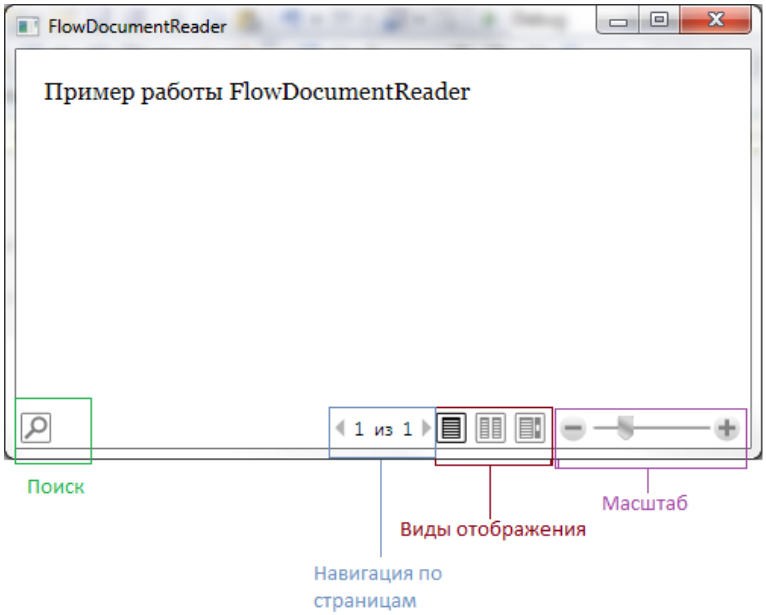
Документы нефиксированного формата.

Если документы фиксированного формата нацелены на точность отображения и печати, то документы нефиксированного формата направлены в первую очередь на удобство работы с информацией на экране. Документы нефиксированного формата поддерживают перестроение информации при изменении размеров окна, масштабирование.

Для обозначения документов нефиксированного формата используется класс **FlowDocument**. Так как у него нет визуального отображения, то при определении **FlowDocument** нужно завернуть в один из четырех типов контейнеров документов нефиксированного формата:

- **FlowDocumentReader**;
- **FlowDocumentPageViewer**;
- **FlowDocumentScrollViewer**;
- **RichTextBox**.

Использование контейнера **FlowDocumentReader** даст нам следующий результат:



Код на XAML:

```
<Window x:Class="Documents.FlowDocumentReaderWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="400"
    Height="200" Title="FlowDocumentReader">
    <FlowDocumentReader>
        <FlowDocument>
            <Paragraph>
                Пример работы FlowDocumentReader
            </Paragraph>
        </FlowDocument>
    </ FlowDocumentReader >
</Window>
```

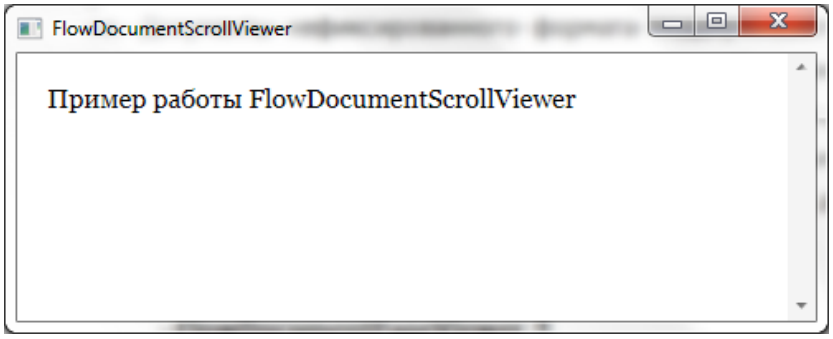
Как видим, у нас есть и возможность поиска, и навигация по страницам, и варианты видов, и масштабирование содержимого.

FlowDocumentPageViewer – урезанная версия предыдущего контейнера. У этого контейнера отсутствуют варианты видов (можно просматривать только одну страницу), а также нет поиска по содержимому.

Код на XAML:

```
<Window x:Class="Documents.FlowDocumentPageViewerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="400"
    Height="200" Title="FlowDocumentPageViewer">
    <FlowDocumentPageViewer>
        <FlowDocument>
            <Paragraph>
                Пример работы FlowDocumentPageViewer
            </Paragraph>
        </FlowDocument>
    </FlowDocumentPageViewer>
</Window>
```

Третий вариант контейнера – **FlowDocumentScrollViewer –** самая простая версия из перечисленных. В этом контейнере документ нефиксированного формата будет отображен как непрерывное полотно с вертикальным скроллингом.



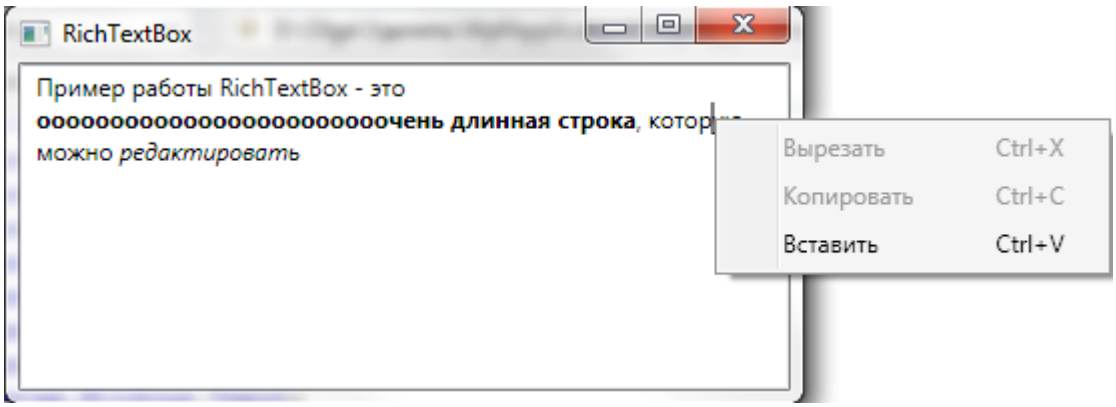
Код на XAML:

```
<Window x:Class="Documents.FlowDocumentScrollViewerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="400"
    Height="200" Title="FlowDocumentScrollViewer">
    <FlowDocumentScrollViewer>
        <FlowDocument>
            <Paragraph>
```

```
                Пример работы FlowDocumentScrollViewer
            </Paragraph>
        </FlowDocument>
    </FlowDocumentScrollViewer>
</Window>
```

И последний вариант контейнера для документов нефиксированного формата, который предлагает WPF, - это **RichTextBox**. Среди перечисленных контейнеров это единственный контейнер, который подразумевает редактирование содержимого пользователем. Кроме того, у него отсутствуют возможности предыдущих контейнеров, но есть свои собственные. У него нет масштабирования, навигации по страницам, поиска, но в нем сразу есть контекстное меню, есть возможность форматирования текста командами, например, такими как **ToggleBold** (Ctrl+B) или **AlignCenter** (Ctrl+E), а также есть возможность включить проверку орфографии в режиме реального времени. В него можно вставлять параграфы, таблицы и картинки, то есть все то, что может быть в документе нефиксированного формата.

Этот контейнер является самым подходящим элементом управления при создании своего редактора. Выглядит он следующим образом (вид шрифта изменен при помощи сочетаний клавиш для демонстрации возможностей форматирования):



Код на XAML:

```
<Window x:Class="Documents.RichTextBoxWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="400"
```

```
    Height="200" Title="RichTextBox">

    <RichTextBox>
        <FlowDocument>
            <Paragraph>
                Пример работы RichTextBox - это ооооооооооооооооооооооочень
длинная строка, которую можно редактировать
            </Paragraph>
        </FlowDocument>
    </RichTextBox>

</Window>
```

Отображение документов

Определив тип контейнера, можно переходить к наполнению документа нефиксированного формата содержимым. В качестве дочерних элементов **FlowDocument** могут использоваться только определенные элементы, главными из которых являются 2 группы элементов-потомков абстрактных классов **Block** (элементы блокового типа) и **Inline** (встраиваемые элементы в блоки), причем на верхнем уровне заполнения могут использоваться только блок-элементы**.**

Элементы управления – классы, производные от **Block**:

Paragraph – применяется для создания абзаца. Технически, **Paragraph** не может содержать просто текст. Он может содержать только inline-элементы. Но если посмотреть на предыдущие примеры, то мы спокойно писали таким образом:

```
<Paragraph>  Какой-либо текст  </Paragraph>
```

У нас есть такая возможность, потому что класс **Paragraph** в таких случаях самостоятельно неявно создает inline-элемент **Run** для текста. Элемент **Run** будет описан ниже.

Рассмотрим некоторые свойства, характерные для **Paragraph**.

Свойство	Описание

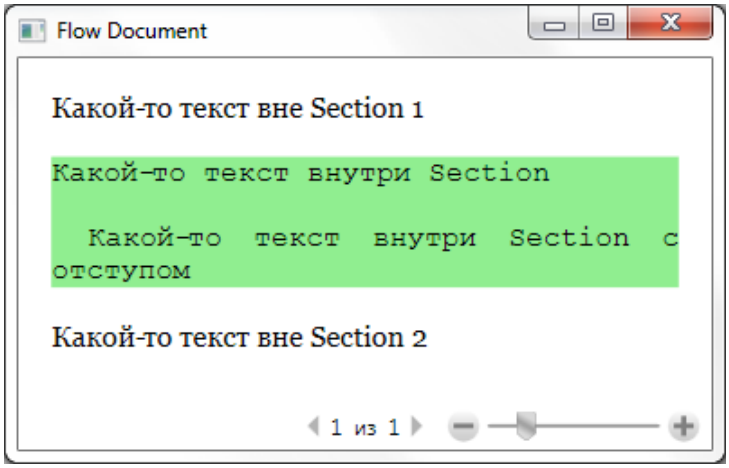
KeepTogether	Можно ли разрывать содержимое абзаца на разные страницы или нет. Значение по умолчанию – можно (false).
KeepWithNext	Не разрывать со следующим абзацем. Значение по умолчанию – разрывать (false).
MinOrphanLines	Минимальное число строк, которые должны остаться, если абзац разделяется на несколько страниц или колонок. Значение по умолчанию – 0.
MinWidowLines	Минимальное количество строк, которые перенесутся, если абзац делится на несколько страниц или колонок. Значение по умолчанию – 0.
TextIntend	Отступ первой строки абзаца.

Section – применяется только для объединения других элементов, производных от Block, в группу, например для присвоения сразу всей группе элементов каких-то параметров.

Пример:

```
<Paragraph>
  Какой-то текст вне Section 1
</Paragraph>

<Section FontFamily="Courier new" Background="LightGreen">
   <Paragraph>
     Какой-то текст внутри Section
   </Paragraph>
   <Paragraph TextIndent="20">
     Какой-то текст внутри Section с отступом
   </Paragraph>
</Section>

<Paragraph>
  Какой-то текст вне Section 2
</Paragraph>
```

В результате получим:



List – используется для создания маркированного или нумерованного списка. Для создания пункта списка используется **ListItem**, в которые необходимо вписать какой-либо block-элемент, например, **Paragraph**.

Рассмотрим свойства, характерные для **List**.

Свойство	Описание
MarkerOffset	Задает расстояние между пунктом меню и его маркером. В XAML может быть задано следующим образом: 1. Число double (в пикселях) 2. Число double с указанием единицы измерения (px, in, cm, pt) 3. Auto (подбор в зависимости от настроек шрифта)
MarkerStyle	Какой значок (маркер) будет в начале каждого пункта. Возможны следующие варианты: None, Disc (используется по умолчанию), Circle, Square, Box, LowerRoman, UpperRoman, LowerLatin, UpperLatin, Decimal

Перечислим в примере все виды маркеров:

```
<Paragraph>Примеры маркеров в списке: </Paragraph>
  <List MarkerStyle="None">
     <ListItem><Paragraph>None</Paragraph></ListItem>
  </List>
  <List>
     <ListItem><Paragraph>Disc</Paragraph></ListItem>
  </List>
```

```
<List MarkerStyle="Circle">
   <ListItem><Paragraph>Circle</Paragraph></ListItem>
</List>
<List MarkerStyle="Box">
   <ListItem><Paragraph>Box</Paragraph></ListItem>
</List>
<List MarkerStyle="Square">
   <ListItem><Paragraph>Square</Paragraph></ListItem>
</List>
<List MarkerStyle="LowerRoman">
   <ListItem><Paragraph>LowerRoman 1</Paragraph></ListItem>
   <ListItem><Paragraph>LowerRoman 2</Paragraph></ListItem>
</List>
<List MarkerStyle="UpperRoman">
   <ListItem><Paragraph>UpperRoman 1</Paragraph></ListItem>
   <ListItem><Paragraph>UpperRoman 2</Paragraph></ListItem>
</List>
<List MarkerStyle="UpperLatin">
   <ListItem><Paragraph>UpperLatin 1</Paragraph></ListItem>
   <ListItem>
      <Paragraph>UpperLatin 2</Paragraph>
      <List MarkerStyle="LowerLatin">
        <ListItem><Paragraph>LowerLatin 1</Paragraph></ListItem>
        <ListItem><Paragraph>LowerLatin 2</Paragraph></ListItem>
      </List>
   </ListItem>
</List>
<List MarkerStyle="Decimal">
   <ListItem><Paragraph>Decimal 1</Paragraph></ListItem>
   <ListItem><Paragraph>Decimal 2</Paragraph></ListItem>
</List>
```

В результате получим следующие списки:

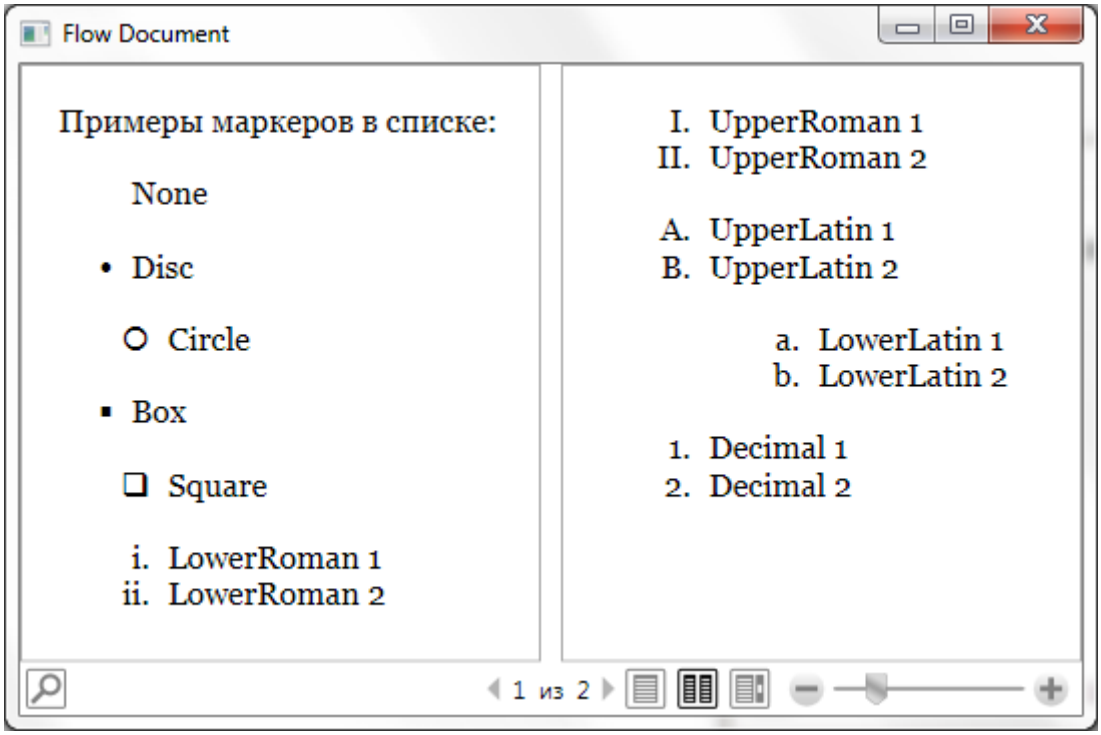
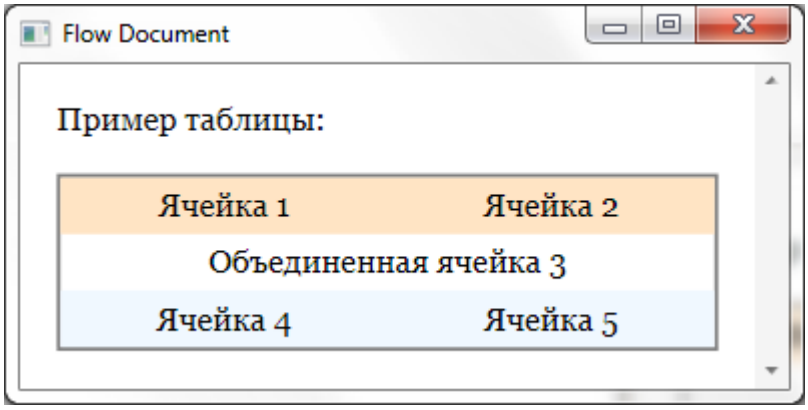


Table – создание таблицы в документе нефиксированного формата. Класс **Table** обеспечивает сеточное представление, состоящее из столбцов (представленных элементами **TableColumn**) и строк (представленных элементами **TableRow**). Элементы **TableColumn** не содержат данных; они просто определяют столбцы и их характеристики. Элементы **TableRow** должны быть расположены в элементе **TableRowGroup**, определяющем группировку строк для таблицы. Элементы **TableCell**, содержащие фактические данные, отображаемые в таблице, должны быть расположены в элементе **TableRow**. **TableCell** может содержать только элементы, производные от **Block**.

Пример таблицы:

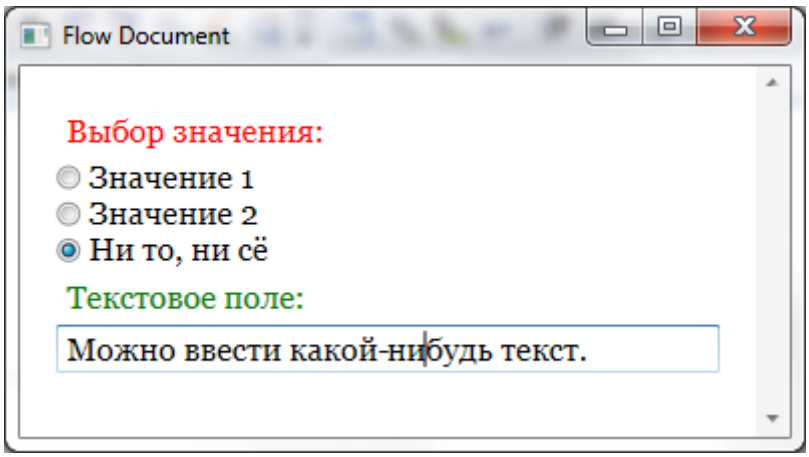
```
<Section TextAlignment="Center">
    <Table  CellSpacing="10" BorderBrush="Gray" BorderThickness="2">
        <TableRowGroup >
            <TableRow Background="Bisque">
                <TableCell><Paragraph>Ячейка 1</Paragraph></TableCell>
                <TableCell><Paragraph>Ячейка 2</Paragraph></TableCell>
            </TableRow>
            <TableRow>
                <TableCell ColumnSpan="2">
                    <Paragraph>Объединенная ячейка 3</Paragraph>
                </TableCell>
            </TableRow>
            <TableRow Background="AliceBlue">
                <TableCell><Paragraph>Ячейка 4</Paragraph></TableCell>
                <TableCell><Paragraph>Ячейка 5</Paragraph></TableCell>
            </TableRow>
        </TableRowGroup>
    </Table>
</Section>
```



BlockUIContainer – элемент управления для встраивания в документ типа **FlowDocument** элементов, производных от **UIElement**. К ним относятся кнопки, чекбоксы и так далее. Следует заметить, что нет никакого другого способа добавить **UIElement** в документ нефиксированного формата. Кроме этого есть особенность использования **BlockUIContainer**, которую надо учитывать, - в него можно поместить только один **UIElement.** Но если **UIElement** является контейнером (например, **Grid**), то в сам элемент можно встраивать столько, сколько нужно элементов.

Пример использования **BlockUIContainer**

```
<BlockUIContainer>
    <StackPanel>
        <Label Foreground ="Red">Выбор значения:</Label>
        <StackPanel>
            <RadioButton>Значение 1</RadioButton>
            <RadioButton>Значение 2</RadioButton>
            <RadioButton>Ни то, ни сё</RadioButton>
        </StackPanel>

        <Label Foreground="Green">Текстовое поле:</Label>
        <TextBox>Можно ввести какой-нибудь текст.</TextBox>
    </StackPanel>
</BlockUIContainer>
```



Элементы управления – классы, производные от **Inline**:

Run – просто содержит текст без какого-либо форматирования. Если нужно вставить текст, обычно принято использовать **Span**, а не

Run. С другой стороны **Run** часто создается неявно, например, когда текст добавляется в **Paragraph**.

Span – применяется для объединения в группу других Inline-элементов.

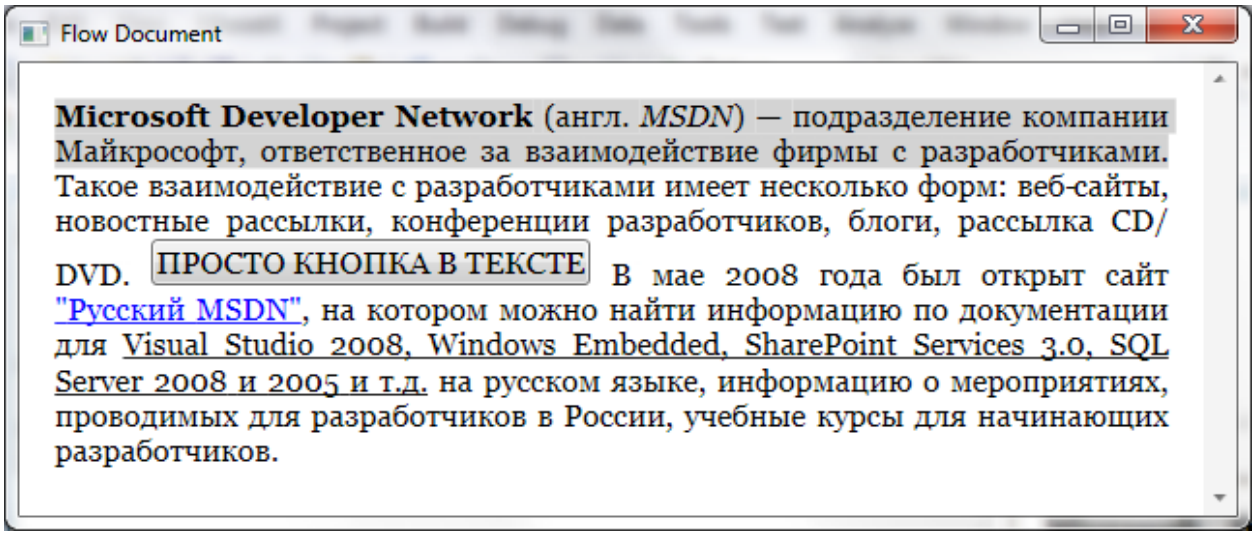
Bold, Italic, Underline – форматирование текста (полужирный шрифт, курсив и подчеркнутый соответственно).

Hyperlink – создание гиперссылки в документе.

LineBreak – переход на новую строку.

InlineUIContainer – встраивает **UIElement** в Inline-элемент. Если им пренебречь, то будет создан неявно.

Покажем, как работают перечисленные inline-элементы на примере.



Код примера на XAML:

```
<FlowDocument>

    <Paragraph>
        <Span Background="LightGray"><Bold>Microsoft Developer Network</Bold>
(англ. <Italic>MSDN</Italic>) — подразделение компании Майкрософт,
ответственное за взаимодействие фирмы с разработчиками.</Span> Такое
взаимодействие с разработчиками имеет несколько форм: веб-сайты, новостные
рассылки, конференции разработчиков, блоги, рассылка CD/DVD.
        <InlineUIContainer>
            <Button>ПРОСТО КНОПКА В ТЕКСТЕ</Button>
        </InlineUIContainer>
        <Run>В мае 2008 года был открыт сайт </Run> <Hyperlink
NavigateUri="http://msdn.microsoft.com/ru-ru/">"Русский
MSDN"</Hyperlink><Run>, на котором можно найти информацию по документации для
</Run>
        <Underline>Visual Studio 2008, Windows Embedded, SharePoint Services
```

```
3.0, SQL Server 2008 и 2005 и т.д.</Underline>
        <Run>на русском языке, информацию о мероприятиях, проводимых для
разработчиков в России, учебные курсы для начинающих разработчиков.</Run>
    </Paragraph>

</FlowDocument>
```

Floater и **Figure** – элементы для встраивания в документ с размещением на форме, отличным от остального. Хотя функционал у них практически одинаков, перечислим отличия **Figure** и **Floater** (список отличий взят из MSDN).

Figure:

- Для элемента можно определить местоположение.

Элемент можно разместить в нескольких столбцах. Можно установить высоту и ширину **Figure** в значения, кратные странице, содержимому, высоте или ширине столбца.

Элемент не разбивается постранично. Если содержимое в **Figure** не помещается внутри **Figure**, будет отображаться все поместившееся содержимое, а не помещающаяся часть будет утеряна.

Floater:

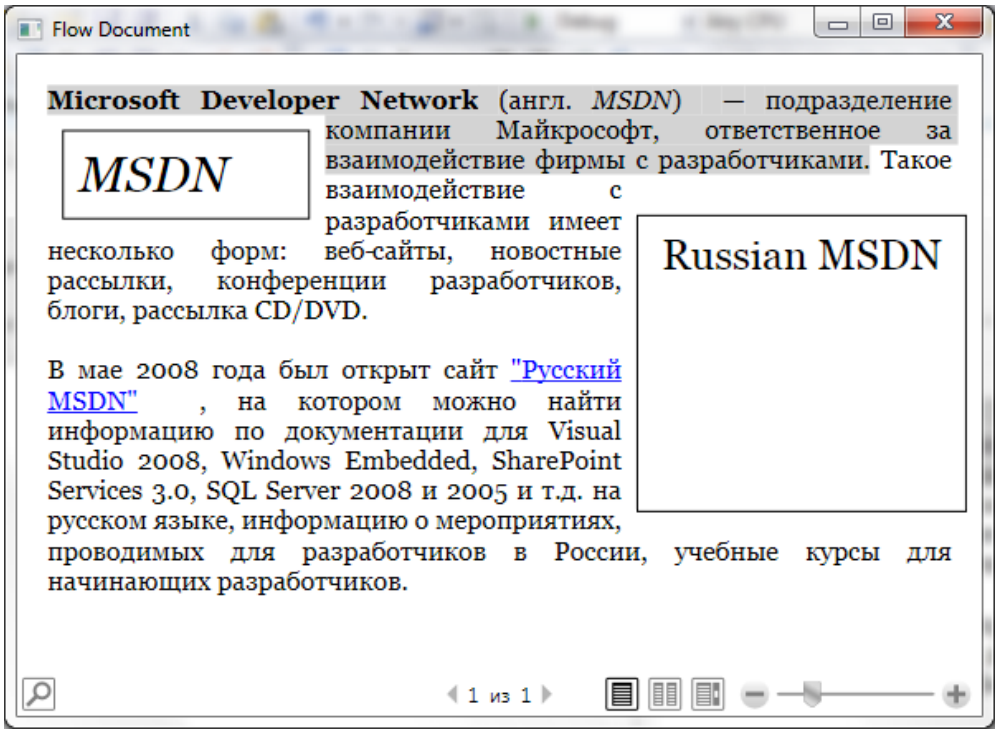
- Для элемента невозможно определить местоположение, и он будет отображаться в любом доступном для него месте.

Не может изменяться по размеру до размера большего, чем один столбец. По умолчанию **Floater** имеет размер, равный одному столбцу. Свойство элемента **Width** может быть задано в абсолютных точках, но в случае, если это значение превышает ширину одного столбца, оно будет игнорироваться, а размеры плавающего объекта изменятся до размера одного столбца. Размеры можно уменьшить до размеров, меньших чем один столбец, задав правильную ширину точки. **Floater** не имеет

свойства высоты, т.е. его высота не может быть задана, так как зависит от содержимого.

Floater переносится постранично. Если его содержимое в заданной ширине расширяется на более чем 1 высоту столбца, плавающий объект прерывается и переходит в следующий столбец, страницу и т.д.

Посмотрим, как это выглядит на примере.



Код на XAML:

```
<FlowDocument>
  <Paragraph>
    <Span Background="LightGray"><Bold>Microsoft Developer Network</Bold>
(англ. <Italic>MSDN</Italic>)
    <Floater HorizontalAlignment="Left" Width="150" BorderBrush="Black"
BorderThickness="1">
        <Paragraph FontSize="30" FontStyle="Italic">MSDN</Paragraph>
    </Floater> — подразделение компании Майкрософт, ответственное за
взаимодействие фирмы с разработчиками.</Span>
    Такое взаимодействие с разработчиками имеет несколько форм: веб-сайты,
новостные рассылки, конференции разработчиков, блоги, рассылка CD/DVD.
  </Paragraph>
  <Paragraph>
    В мае 2008 года был открыт сайт <Hyperlink
NavigateUri="http://msdn.microsoft.com/ru-ru/">"Русский MSDN"</Hyperlink>
    <Figure Width="200" Height="180" HorizontalAnchor="PageRight"
VerticalAnchor="PageCenter" TextAlignment="Center" BorderBrush="Black"
BorderThickness="1">
        <Paragraph FontSize="25">Russian MSDN</Paragraph>
```

```
    </Figure>
     , на котором можно найти информацию по документации для Visual Studio
2008, Windows Embedded, SharePoint Services 3.0, SQL Server 2008 и 2005 и
т.д. на русском языке, информацию о мероприятиях, проводимых для
разработчиков в России, учебные курсы для начинающих разработчиков.
   </Paragraph>
</FlowDocument>
```

Подведем итоги:

Figure удобен для размещения отдельного содержимого, для которого необходимо контролировать размер и размещение, если содержимое помещается в заданные размеры.

Floater удобен для размещения более свободного содержимого, расположенного подобно содержимому главной страницы, но отделенного от него.

Пример приложения для отображения документа нефиксированного формата.

Код на XAML:

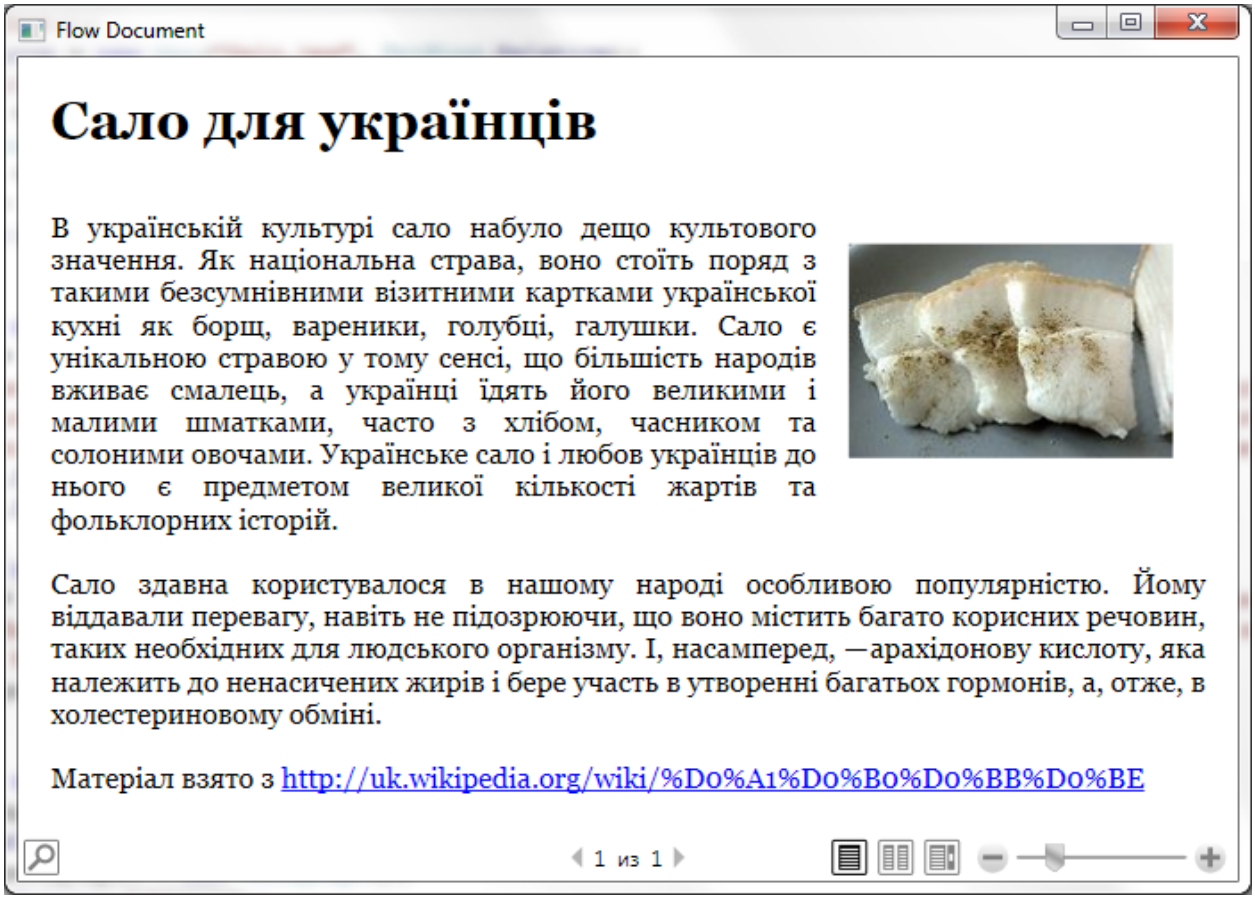
```
<Window x:Class="FlowDocuments.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="700"
    Height="500"   Title="Flow Document">

    <FlowDocumentReader Name="DocReader"></FlowDocumentReader>
</Window>
```

Код на C#:

```
public Window1()
{
    InitializeComponent();

    //Заголовок
    Run run1 = new Run("Сало для українців");
    Paragraph paragraph1 = new Paragraph(new Bold(run1));
    paragraph1.FontSize = 30;


    Paragraph paragraph2 = new Paragraph();
    //картинка
    Image img = new Image();
    BitmapImage bimg = new BitmapImage();
    bimg.BeginInit();
    bimg.UriSource = new Uri("Salo.jpg", UriKind.Relative);
    bimg.EndInit();
    img.Source = bimg;
    BlockUIContainer block = new BlockUIContainer(img);
```

```
    Figure fig = new Figure(block);
    fig.Width = new FigureLength(200);
    fig.HorizontalAnchor = FigureHorizontalAnchor.ContentRight;

    //текст
    Run run2 = new Run();
    run2.Text = "В українській …";
    paragraph2.Inlines.Add(fig);
    paragraph2.Inlines.Add(run2);

    Run run3 = new Run();
    run3.Text = "Сало ….";
    Paragraph paragraph3 = new Paragraph(run3);

    //ссылки
    Run run4 = new Run("Матеріал взято з ");
    Hyperlink hp = new Hyperlink(new
Run("http://uk.wikipedia.org/wiki/%D0%A1%D0%B0%D0%BB%D0%BE"));
    hp.NavigateUri = new
Uri("http://uk.wikipedia.org/wiki/%D0%A1%D0%B0%D0%BB%D0%BE");
    Paragraph paragraph4 = new Paragraph();
    paragraph4.Inlines.Add(run4);
    paragraph4.Inlines.Add(hp);

    FlowDocument doc = new FlowDocument();
    doc.Blocks.Add(paragraph1);
    doc.Blocks.Add(paragraph2);
    doc.Blocks.Add(paragraph3);
    doc.Blocks.Add(paragraph4);

    DocReader.Document = doc;
}
```

Результат получился следующий:



Данный пример называется **FlowDocuments** и располагается в папке **Source**.

3.Соединение данных и элементов управления

Общие принципы

Большинство приложений, с которыми вы работаете в повседневной жизни оперируют некоторыми данными. Например, заполняя регистрационную форму вы взаимодействуете с информацией о человеке, просматривая товары в веб-магазине вы неявно общаетесь с базой данных магазина. И в том и в другом случае налицо набор UI элементов взаимодействующих с источником данных.

Что может являться источником данных? Ответ на данный вопрос достаточно комплексный, например это может быть столбец базы данных, xml файл, объект класса, элемент управления.

Обмен информацией между элементами управления и источниками данных можно производить посредством написания кусков кода, как вы это уже делали, альтернативой является возможность использования механизмов связывания (соединения) данных и элементов управления.

В английской литературе это средство называется **Data Binding**.

Data Binding позволяет настроить автоматический обмен данными между элементами управления и источниками данных без написания больших фрагментов кода.

Схема взаимодействия источника и приёмника данных

Перейдем к рассмотрению схемы соединения данных и элементов управления.

На схеме изображено три объекта, участвующих в процессе: источник данных (source object), объект связывания (**Binding** object), приёмник данных (target object). На первом шаге необходимо определиться с источником данных, выбрать какое свойство источника данных мы будем использовать. Например, если источником данных является объект класса **Student**, в качестве свойства поставляющего данные можно выбрать его фамилию. На втором шаге используя объект связывания выбрать какое-то свойство зависимости (**dependency property**) в приёмнике данных. Это свойство зависимости будет обмениваться данными со свойством из источника данных. В нашем

примере мы можем в качестве приёмника выбрать **TextBlock**, а в качестве свойства зависимости **Text**.

Режимы связывания данных

При выполнении связывания нужно определиться с режимами связывания. Они бывают такие:

OneWay – обновление данных в источнике отображаются в приёмнике, при этом изменение данных в приёмнике не влияет на источник. Этот режим удобен для создания элементов управления только для чтения, например в статическом тексте можно отображать фамилию текущего менеджера некоторого отдела

TwoWay – обновление данных работает в обе стороны, то есть изменение данных в источнике обновляет приёмник и наоборот. Данный режим удобен для создания форм, предназначенных для редактирования данных

OneWayToSource – режим обратный **OneWay**. Изменение данных в приёмнике влияет на источник, а наоборот нет.

OneTime - при данном режиме источник заполняет данными приёмник только один раз, после чего любые изменения источника и приёмника не синхронизируются.

Класс Binding

От теоретических познаний перейдем к суровым практическим реалиям. За связывание отвечает класс **Binding**. Он располагается в пространстве **System.Windows.Data**.

```
public class Binding : BindingBase
```

Важнейшими свойствами данного класса являются: **Source** – позволяет задать источник данных, **Path** – позволяет задать имя

свойства в источнике, откуда будут черпаться данные, **Mode -** задает режим связывания данных (это могут быть значения **OneWay**,**TwoWay**, **OneWayToSource**, **OneTime**,**Default**).

Примеры связывания данных

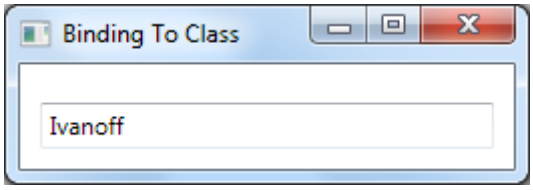
Рассмотрим ***пример связывания элемента управления со свойством класса***. Свяжем свойство **Text** класса **TextBox** со свойством **Surname**. Код **XAML** файла:

```
<Window x:Class="BindingToClass.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Binding To Class" Height="81" Width="190">
    <Grid xmlns:c="clr-namespace:BindingToClass">
          <Grid.Resources>
            <c:Person x:Key="PersonSource"/>
          </Grid.Resources>
          <TextBox Height="23" Margin="10,10,10,10" Name="textBox1"
VerticalAlignment="Bottom" Text="{Binding Path=Surname, Mode=OneTime,
          Source={StaticResource PersonSource}}"/>
    </Grid>
</Window>
```

Код **cs** файла:

```
class Person
{
        public string Surname{
            get{
                return "Ivanoff";
            }
        }
}
```

Для задания связи мы используем объект класса **Binding** и устанавливаем свойства **Path**,**Mode**,**Source**. Результат работы программы:



Данный пример называется **BindingToClass** и находиться в папке **Source** текущего урока.

Рассмотрим ***пример связывания элемента управления с объектом классом***. Если при связывании элемента управления не указать свойство **Path**, то по умолчанию связывание производится со всем объектом. Для получения значения у связанного объекта будет вызываться метод **ToString**.

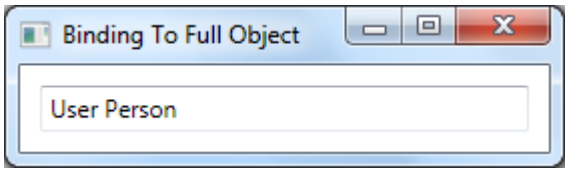
Код **XAML** файла:

```
<Window x:Class="BindingToFullObject.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Binding To Full Object" Height="81" Width="190">
    <Grid xmlns:c="clr-namespace:BindingToFullObject">
        <Grid.Resources>
            <c:Person x:Key="PersonSource"/>
        </Grid.Resources>
        <TextBox Height="23" Margin="10,10,10,10" Name="textBox1"
VerticalAlignment="Bottom" Text="{Binding Mode=OneTime,
          Source={StaticResource PersonSource}}"/>
    </Grid>
</Window>
```

Код **cs** файла:

```
class Person
{
    public override string ToString()
    {
       return "User Person";
    }
}
```

Результат работы программы:



Данный пример называется **BindingToFullObject** и находиться в папке **Source** текущего урока.

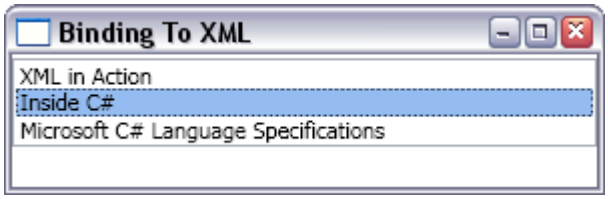
Рассмотрим ***пример связывания элемента управления с XML данными***. В этом примере источником будут служить XML данные.

Код **XAML** файла:

```
<Window x:Class="BindingToXml.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Binding To XML" Height="95" Width="300">
    <StackPanel Height="62">
        <StackPanel.Resources>
            <XmlDataProvider x:Key="inventoryData" XPath="Inventory/Books">
                <x:XData>
                    <Inventory xmlns="">
                        <Books>
                            <Book ISBN="0-7356-0562-9" Stock="in" Number="9">
                                <Title>XML in Action</Title>
                                <Summary>XML Web Technology</Summary>
                            </Book>
                            <Book ISBN="0-7356-1288-9" Stock="out"
Number="7">
                                <Title>Inside C#</Title>
                                <Summary>C# Language Programming</Summary>
                            </Book>
                            <Book ISBN="0-7356-1448-2" Stock="out"
Number="4">
                                <Title>Microsoft C# Language
Specifications</Title>
                                <Summary>The C# language definition</Summary>
                            </Book>
                        </Books>
                    </Inventory>
                </x:XData>
            </XmlDataProvider>
        </StackPanel.Resources>
        <ListBox>
            <ListBox.ItemsSource>
                <Binding Source="{StaticResource inventoryData}"
XPath="*[@Stock='out'] | *[@Number>=5 or @Number=3]"/>
            </ListBox.ItemsSource>
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding XPath=Title}">
                    </TextBlock>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </StackPanel>
</Window>
```

Как вы видите в данном пример источником данных служит **xml**, содержащий информацию о книгах, встроенный в **XAML** файл.

Результат работы программы:



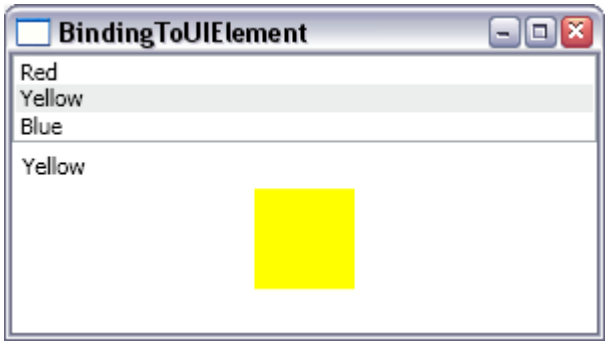
Данный пример называется **BindingToXml** и находиться в папке **Source** текущего урока.

Рассмотрим ***пример связывания элемента управления с другим элементом управления***. Предположим, что вы разрабатываете приложение отображающее информацию о заказчиках в списке, и хотите демонстрировать пользователю текущего выбранного заказчика в отдельном элементе управления. Для решения этой задачи удобно использовать такой тип связывания. Ниже приведенный пример отображает список цветов, и два элемента управления:**Label** – содержит название текущего выбранного цвета, **Canvas** – использует текущий цвет в качестве фона.

Код **XAML** файла:

```
<Window x:Class="BindingToUIElement.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BindingToUIElement" Height="300" Width="300">
    <StackPanel>
        <ListBox x:Name="colorList" SelectedIndex="1">
            <ListBoxItem>Red</ListBoxItem>
            <ListBoxItem>Yellow</ListBoxItem>
            <ListBoxItem>Blue</ListBoxItem>
        </ListBox>
        <Label Content="{Binding ElementName=colorList,
Path=SelectedItem.Content}"/>
        <Canvas Background="{Binding ElementName=colorList,
Path=SelectedItem.Content}" Height="50" Width="50" />
    </StackPanel>
</Window>
```

Обратите внимание, что для связывания используется свойство **ElementName**. Оно позволяет установить элемент, с которым будет связан приёмник. Результат работы программы:



Данный пример называется **BindingToUIElement** и находиться в папке **Source** текущего урока.

При использовании связывания и режимов обмена данными необходимо понимать, когда произойдет обмен. Для задания режима синхронизации используется свойство **UpdateSourceTrigger** класса **Binding**. Оно может принимать такие значения:

PropertyChanged – обновить значение в источнике, как только оно изменилось в приёмнике

LostFocus – обновить значение в источнике, как только приёмник потерял фокус

Explicit - обновить значение в источнике, только после явного вызова метода **UpdateSource**

Default – использовать поведение по умолчанию **UpdateSourceTrigger** связанного свойства приёмника. Для большинства свойств зависимости по умолчанию стоит **PropertyChanged**, и только для свойства **Text** по умолчанию стоит **LostFocus**

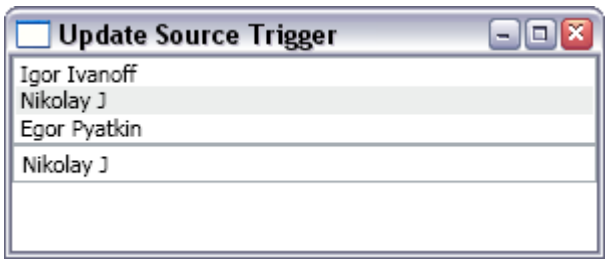
Приведенная выше информация важна только для режимов **TwoWay**, **OneWayToSource**. В качестве примера рассмотрим приложение со списком людей и текстовым полем, позволяющим править значение в списке.

Код **XAML** файла:

```
<Window x:Class="BindingUpdateTrigger.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Update Source Trigger" Height="126" Width="300">
    <StackPanel>
        <ListBox x:Name="peopleList" SelectedIndex="1">
            <ListBoxItem>Igor Ivanoff</ListBoxItem>
            <ListBoxItem>Nikolay Sidoroff</ListBoxItem>
            <ListBoxItem>Egor Pyatkin</ListBoxItem>
        </ListBox>
        <TextBox Text="{Binding ElementName=peopleList,
Path=SelectedItem.Content, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"/>
    </StackPanel>
</Window>
```

При изменение значения в текстовом поле автоматически меняется значение в активной строке списка. Это происходит за счет того, что мы указали режимы **TwoWay** и **PropertyChanged**.

Результат работы программы:



Данный пример называется **BindingUpdateTrigger** и находиться в папке **Source** текущего урока.

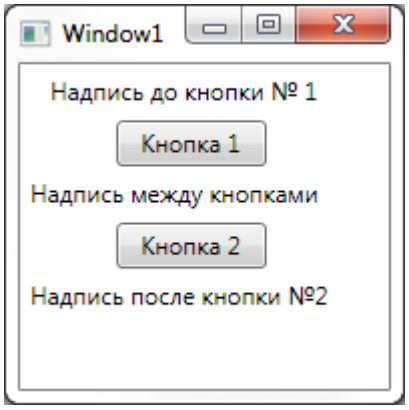
4. Управление стилями и ресурсами

Ресурсы

Если в приложении WPF есть повторяющиеся значения или объекты, например, шрифт или цвет элементов управления, можно использовать **ресурсы WPF.** Ресурсы WPF очень сильно напоминают

таблицы стилей (CSS), только обладают особенностями, настроены на работу с WPF.

Приведем пример, когда имеет смысл использовать ресурсы WPF. У нас есть приложение с двумя кнопками и каким-то текстом между ними.



Код XAML:

```
<Window x:Class="ResourcesDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="200" Width="200">
    <StackPanel Name="StackPanel1" HorizontalAlignment="Left">
        <Label Height="28" Name="label1" Width="150">Надпись до кнопки №
        1</Label>
        <Button Height="23" Name="button1" Width="75">Кнопка 1</Button>
        <Label Height="28" Name="label2" Width="170">Надпись между
        кнопками</Label>
        <Button Height="23" Name="button2" Width="75">Кнопка 2</Button>
        <Label Height="28" Name="label3" Width="170">Надпись после кнопки
        №2</Label>
     </StackPanel>
</Window>
```

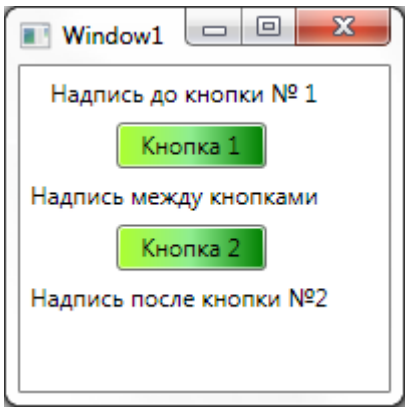
Мы хотим, чтобы кнопки были с градиентным фоном.

Вариант первый – для каждой кнопки можно прописать свойство **Background**. Изменения в примере выделены жирным шрифтом:

```
<Window x:Class="ResourcesDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="200" Width="200">
    <StackPanel Name="StackPanel1" HorizontalAlignment="Left">
```

```
        <Label Height="28" Name="label1" Width="150">Надпись до кнопки №
        1</Label>
        <Button Height="23" Name="button1" Width="75">
         <Button.Background>
          <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
            <GradientStop Color="GreenYellow" Offset="0.0" />
            <GradientStop Color="LightGreen" Offset="0.5" />
            <GradientStop Color="Green" Offset="1.0" />
          </LinearGradientBrush>
        </Button.Background>
            Кнопка 1
        </Button>
        <Label Height="28" Name="label2" Width="170">Надпись между
        кнопками</Label>
        <Button Height="23" Name="button2" Width="75">
         <Button.Background>
          <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
            <GradientStop Color="GreenYellow" Offset="0.0" />
            <GradientStop Color="LightGreen" Offset="0.5" />
            <GradientStop Color="Green" Offset="1.0" />
          </LinearGradientBrush>
        </Button.Background>
            Кнопка 2
        </Button>
        <Label Height="28" Name="label3" Width="170">Надпись после кнопки
        №2</Label>
      </StackPanel>
</Window>
```

В результате приложение будет выглядеть следующим образом:



Цель покрасить кнопки градиентом достигнута, но в случае, если захочется поменять градиент или вообще изменить вид заливки кнопок, придется изменять свойства каждой кнопки. Если бы в приложении была

только одна кнопка, либо вид каждой кнопки отличался бы от остальных, тогда такой вариант является оптимальным, но при повторении лучше использовать второй вариант.

Рассмотрим второй вариант – использование ресурсов. В этом случае код XAML будет выглядеть следующим образом:

```
<Window x:Class="ResourcesDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="200" Width="200">
    <StackPanel Name="StackPanel1" HorizontalAlignment="Left">
        <StackPanel.Resources>
         <LinearGradientBrush x:Key ="GradientBrushForButton"
           StartPoint="0,0" EndPoint="1,0">
           <GradientStop Color="GreenYellow" Offset="0.0" />
           <GradientStop Color="LightGreen" Offset="0.5" />
           <GradientStop Color="Green" Offset="1.0" />
         </LinearGradientBrush>
        </StackPanel.Resources>
         <Label Height="28" Name="label1" Width="150">Надпись до кнопки №
          1</Label>
       <Button Height="23" Name="button1" Width="75"
       Background="{DynamicResource GradientBrushForButton}">
       Кнопка 1</Button>
       <Label Height="28" Name="label2" Width="170">Надпись между
        кнопками</Label>
       <Button Height="23" Name="button2" Width="75"
       Background="{StaticResource GradientBrushForButton}">
       Кнопка 2</Button>
       <Label Height="28" Name="label3" Width="170">Надпись после кнопки
         №2</Label>
    </StackPanel>
</Window>
```

В результате выполнения данного кода мы получим такой же вид кнопок, как и в первом варианте. При этом у нас получился более компактный и наглядный код. Кроме того в этом варианте появилась возможность изменения кода в одном месте (**<StackPanel.Resources>**), в результате чего обе кнопки изменятся.

Теперь рассмотрим подробнее данный пример для того, чтобы понять, что же там написано. Первое, что мы сделали, - это определили свойство **Resources** для элемента управления **StackPanel**. Это свойство есть у всех элементов классов, унаследованных от **FrameworkElement** или **FrameworkContentElement**. Поэтому можно было определить ресурсы как для **StackPanel**, так и для **Window** или **Button**.

При обращении к ресурсам сначала проверяются ресурсы элемента, который обращается. После этого идет поиск вверх по визуальному дереву элементов. В нашем случае сначала проверялись ресурсы **Button**, потом **StackPanel**, далее **Window.** Если искомый ресурс не найден, то ресурсы ищутся в файле приложения. То есть если мы определили бы ресурс **GradientBrushForButton** в ресурсах одной из кнопок, то вторая кнопка этот ресурс уже не увидела бы. Так как обе кнопки находятся в **StackPanel**, то обе кнопки нашли ресурс и использовали его значения. В то же время, если создать кнопку, не входящую в **StackPanel**, то она опять-таки не увидит данный ресурс. В таком случае лучше объявить **GradientBrushForButton** в ресурсах окна. А в случае, если этот ресурс используют несколько окон, тогда можно объявить ресурс в ресурсах приложения (**App.xaml**).

Каждый ресурс должен иметь уникальный ключ, который объявляется при помощи **x:Key**. По этому ключу к нему будут в дальнейшем обращаться.

Теперь перейдем к рассмотрению вызова ресурса:

```
…
        <Button Height="23" Name="button1" Width="75"
        Background="{DynamicResource GradientBrushForButton}">
        Кнопка 1</Button>
…
        <Button Height="23" Name="button2" Width="75"
        Background="{StaticResource GradientBrushForButton}">
        Кнопка 2</Button>
…
```

Свойству **Background** кнопки мы присваиваем значение ресурса **GradientBrushForButton.** Для того, чтобы это сделать, нужно использовать определенное **расширение разметки** (**markup extension**), то есть конструкцию, при которой обработчик XAML поймет, что в качестве значения свойства не просто строка, а определенный объект (в строке выделяется фигурными скобками). Чтобы определить ресурс используются следующие расширения разметки: **StaticResource** (**статический ресурс**)

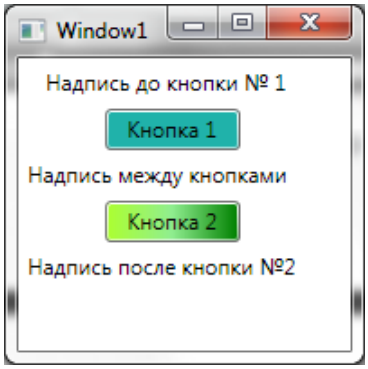
или **DynamicResource** (**динамический ресурс**).

При вызове статического ресурса (**StaticResource**) его значение определяется только раз на этапе загрузки и остается таким на протяжении выполнения программы. В отличие от статического ресурса, динамический ресурс определяется только в момент обращения к нему и реагирует на изменения значений ресурса.

В предыдущем примере первая кнопка использовала динамический ресурс, а вторая – статический. Для того, чтобы понять разницу между ними, в код программы добавим строчку изменения ресурса.

```
StackPanel1.Resources["GradientBrushForButton"] = new
                        SolidColorBrush(Colors.LightSeaGreen);
```

В результате первая кнопка поменяет цвет, так как в процессе выполнения ресурс **GradientBrushForButton** изменился. А вторая кнопка, так как использовала статический ресурс, останется такой же, как была определена в момент загрузки.



Для того, чтобы определиться какой тип ресурса использовать (динамический или статический), нужно выяснить, будет ли меняться значение ресурса во время выполнения программы. Если значение ресурса может меняться, тогда нужно использовать динамический ресурс.

Стили

Давайте еще чуть-чуть изменим наше приложение, а точнее нужно изменить шрифт в надписях (тип шрифта и курсив).

У нас есть несколько вариантов действий из известных нам:

1. Задать изменение шрифта для **Window**. Преимущество – задается один раз. Недостаток – шрифт поменяется и на кнопках.
2. Задать свойства **FontSize** и **FontStyle** для каждой надписи. Недостатки – много повторяющегося кода, при изменении придется менять свойства каждой надписи.
3. Вариант использования ресурсов является также неудачным, так как для его осуществления понадобится прописать сначала ресурсы, а потом все равно прописывать свойства для каждой надписи. Преимущество – переопределение свойств происходит в одном месте, недостатки – очень громоздкий код. (см.

пример), если понадобится изменить какой-то еще параметр, придется прописывать ссылку на ресурс для всех надписей.

```
<StackPanel Name="StackPanel1" HorizontalAlignment="Left" >
  <StackPanel.Resources>
   <LinearGradientBrush x:Key ="GradientBrushForButton" StartPoint="0,0"
EndPoint="1,0">
            …
   </LinearGradientBrush>
   <FontStyle x:Key="LabelFontStyle">Italic</FontStyle>
   <FontFamily x:Key="LabelFontFamily">Times New
Roman</FontFamily>
  </StackPanel.Resources>
    <Label Height="28" Name="label1" Width="150"
   FontStyle="{StaticResource LabelFontStyle}"
   FontFamily="{StaticResource LabelFontFamily}">Надпись до
кнопки № 1</Label>
    <Button Height="23" Name="button1" Width="75"
Background="{DynamicResource GradientBrushForButton}">Кнопка 1</Button>
    <Label Height="28" Name="label2" Width="170"
    FontStyle="{StaticResource LabelFontStyle}"
    FontFamily="{StaticResource LabelFontFamily}">Надпись между
кнопками</Label>
    <Button Height="23" Name="button2" Width="75" Background="{StaticResource
GradientBrushForButton}">Кнопка 2</Button>
    <Label Height="28" Name="label3" Width="170"
    FontStyle="{StaticResource LabelFontStyle}"
    FontFamily="{StaticResource LabelFontFamily}">Надпись после
кнопки №2</Label>
</StackPanel>
```

Ни один из приведенных примеров не является оптимальным. Поэтому в **WPF** есть более элегантное решение – стили. Стили являются разновидностью ресурсов и соответственно задаются в свойстве **Resources** элемента. Стиль является своеобразным контейнером для набора значений свойств (в нашем случае – **FontStyle** и **FontFamily**). В следующем примере показано, как при помощи стилей указать свойства для всех надписей, входящих в **StackPanel**.

```
<Window …>
    <StackPanel Name="StackPanel1" HorizontalAlignment="Left" >
        <StackPanel.Resources>
            …
        <Style TargetType="Label">
          <Setter Property="FontStyle" Value="Italic"/>
          <Setter Property="FontFamily" Value="Times New
          Roman"/>
```

```
        </Style>
      </StackPanel.Resources>
      <Label Height="28" Name="label1" Width="150">Надпись до кнопки №
      1</Label>
      <Button Height="23" Name="button1" Width="75"
      Background="{DynamicResource GradientBrushForButton}">
      Кнопка 1</Button>
      <Label Height="28" Name="label2" Width="170">Надпись между
      кнопками</Label>
      <Button Height="23" Name="button2" Width="75"
      Background="{StaticResource GradientBrushForButton}">
      Кнопка 2</Button>
      <Label Height="28" Name="label3" Width="170">Надпись после кнопки
      №2</Label>
    </StackPanel>
</Window>
```

Рассмотрим подробнее данный пример. Мы в ресурсах **StackPanel** задали стиль для **Label**, указав **TargetType = ”Label”**. При этом для стиля мы не указывали **x:Key.** При таком объявлении стиля для всех элементов **Label**, входящих в **StackPanel**, будет применен данный стиль по умолчанию.

Для каждого типа элементов можно объявить только один стиль без ключа (**x:Key**). То есть, если для **Label** нужно объявить еще какие-то стили, то их нужно объявлять с ключом и в дальнейшем ссылаться на стиль при помощи ключа, но для **Button** мы можем задать стиль по умолчанию.

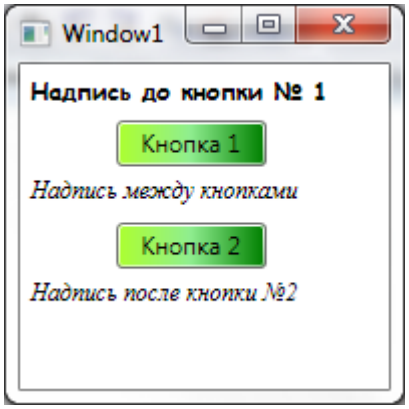
Далее, после объявления стиля, мы начинаем заполнять стиль значениями свойств при помощи **Setter**. Заполнять можно абсолютно любые свойства, которые относятся к типу, для которого создается стиль.

Попробуем указать при помощи стилей, что первая надпись должна показываться жирным шрифтом и Comic Sans MS.

```
<StackPanel Name="StackPanel1" HorizontalAlignment="Left" >
    <StackPanel.Resources>
      …
     <Style TargetType="Label" x:Key="LabelStyle">
       <Setter Property="FontWeight" Value="Bold"/>
       <Setter Property="FontFamily" Value="Comic Sans MS"/>
      </Style>
```

```
    </StackPanel.Resources>
    <Label Height="28" Name="label1" Width="170"
    Style="{StaticResource LabelStyle}">
    Надпись до кнопки № 1</Label>
    …
  </StackPanel>
```

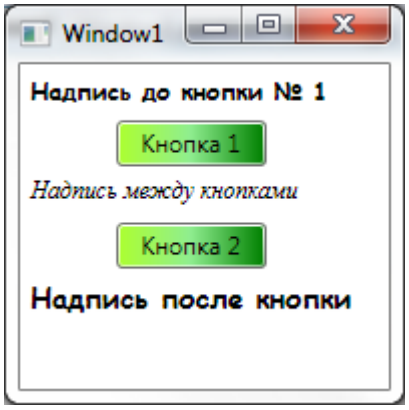
В результате получим:



Кроме всего прочего, стили могут наследоваться друг от друга. То есть если нам нужен стиль, не сильно отличающийся от уже существующего стиля, то можно использовать свойство стилей **BasedOn**. Например, последнюю надпись нужно сделать такую же, как первую, но с другим размером шрифта.

```
<StackPanel Name="StackPanel1" HorizontalAlignment="Left" >
    <StackPanel.Resources>
      …
     <Style TargetType="Label" x:Key="AnotherLabelStyle"
      BasedOn="{StaticResource LabelStyle}">
       <Setter Property="FontSize" Value="14"/>
     </Style>
    </StackPanel.Resources>
    <Label Height="28" Name="label3" Width="170"
    Style="{StaticResource AnotherLabelStyle}">
    Надпись после кнопки №2</Label>
    …
  </StackPanel>
```

Результат:



На данном этапе при наведении курсора на кнопки, они опять окрашиваются в системные цвета. В следующем уроке мы научимся изменять такое поведение при помощи триггеров.

Шаблоны

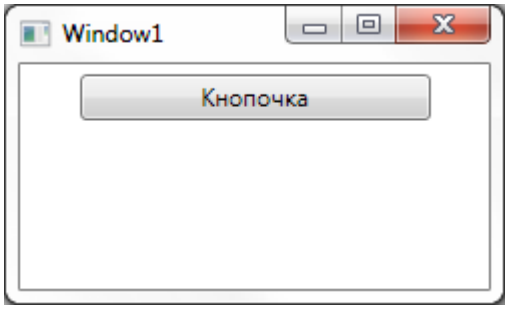
Когда внешний вид элемента нас совсем не устраивает, например, вместо обычной кнопки нам нужна круглая кнопка, используются шаблоны, так как возможностей стилей в данном случае не хватит.

Чтобы понять, как работают шаблоны, необходимо понять термины **логическое дерево** (**logical tree**) и **визуальное дерево** (**visual tree**).

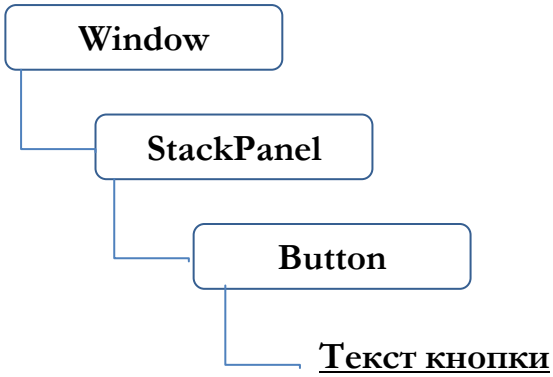
Рассмотрим простой пример с кнопкой на форме:

```
<Window x:Class="VisualTreeDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="150" Width="250">
    <StackPanel>
        <Button Height="23" Name="button1" Width="175"
         Margin="5">
           Кнопочка
        </Button>
    </StackPanel>
  </Window>
```

Выглядит это следующим образом.



Логическое дерево данного окна будет представлять собой следующее:

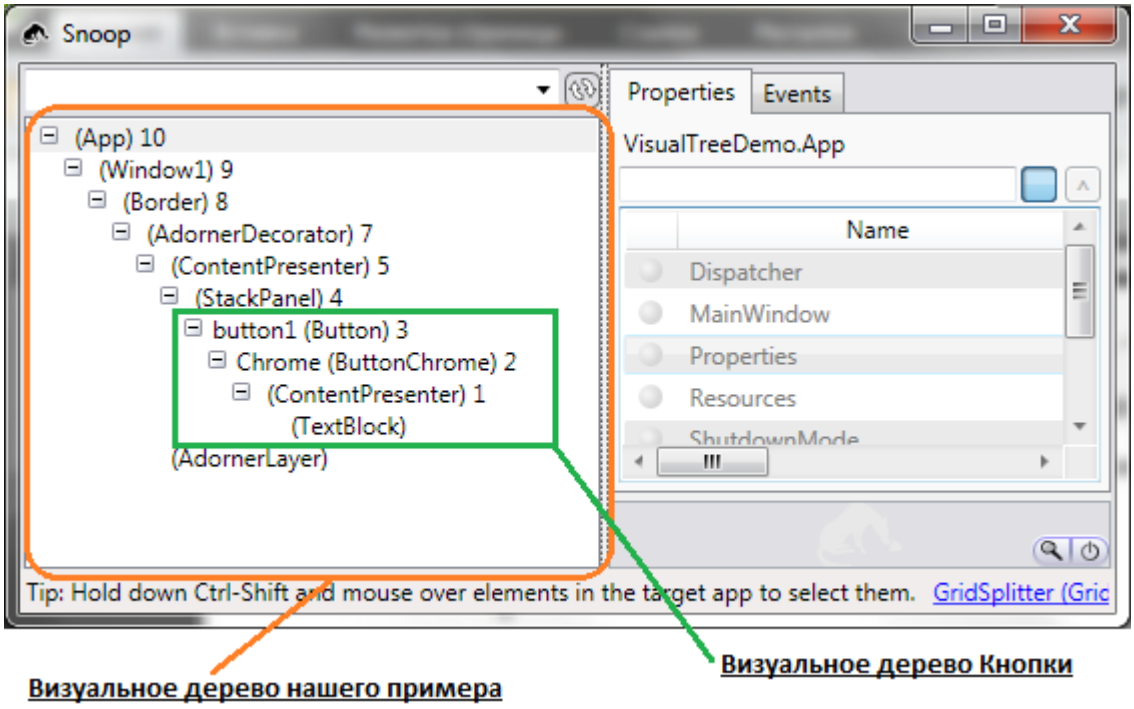


То есть логическое дерево – это порядок расположения элементов управления в окне или странице.

Визуальное дерево отличается от логического дерева тем, что показывает не только элементы окна, но и из чего эти элементы состоят. Оказывается, большинство элементов управления являются на самом деле набором более примитивных элементов, которые создают определенный вид элемента управления.

Посмотреть визуальное дерево приложения можно при помощи утилиты Snoop Utility, входящей в состав Microsoft Expression Blend, либо можно воспользоваться функцией Show the visual tree приложения XAMLPad, входящего в состав Microsoft SDK для Windows. В XAMLPad невозможно посмотреть дерево для Window только для Page.

Визуальное дерево нашего примера будет выглядеть следующим образом:



Как мы видим, исходя из рисунка, кнопка состоит из нескольких элементов, а именно: элемента **ButtonChrome**, который отвечает за внешний вид кнопки в соответствии с темой Windows, и элемента **ContentPresenter**, который может содержать в себе какое-либо наполнение – в данном случае элемент **TextBlock** с текстом «Кнопочка». Если бы у нас на форме был элемент **Label**, он бы тоже предстал в визуальном дереве набором элементов, а именно элементом **Border**, внутри которого будет находиться элемент **ContentPresenter**, содержащий в себе **TextBlock.**

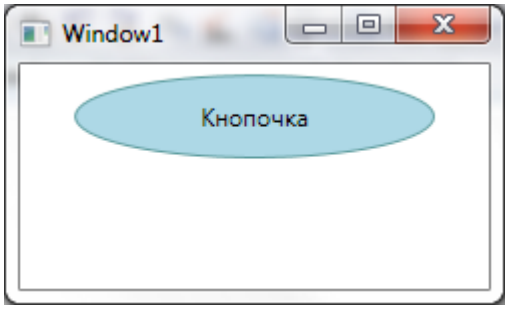
Когда мы собираемся переопределить внешний вид элемента при помощи шаблонов, мы меняем визуальное дерево элемента так, чтобы элемент выглядел желаемым образом. При этом шаблоны не могут изменить поведение элемента, то есть если изменяется внешний вид кнопки, то ее поведение (возможность щелкнуть по ней, возможность установить кнопку по умолчанию и т.д.) остается прежним.

Например, попробуем сделать кнопку в форме синего эллипса.

```
<Window.Resources>
  <ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
```

```
    <Grid>
      <Ellipse Height="42" Stroke="CadetBlue" Width="180" Fill="LightBlue"/>
        <ContentPresenter HorizontalAlignment="Center"
         VerticalAlignment="Center"></ContentPresenter>
    </Grid>
  </ControlTemplate>
</Window.Resources>

<StackPanel>
    <Button Name="button1" Margin="5"
    Template="{StaticResource ButtonTemplate}">Кнопочка</Button>
</StackPanel>
```

В результате получится:



Рассмотрим подробнее код примера.

В разделе **Window.Resources** мы объявили **ControlTemplate** с ключем **ButtonTemplate** и указали, что использоваться данный шаблон будет для кнопки.

Далее необходимо нарисовать кнопку. Чтобы текст был на эллипсе, мы использовали **Grid**, так как эллипс не является контейнером. Объект **ContentPresenter** обязателен, если хотим, чтобы можно было выводить информацию или картинки на кнопку.

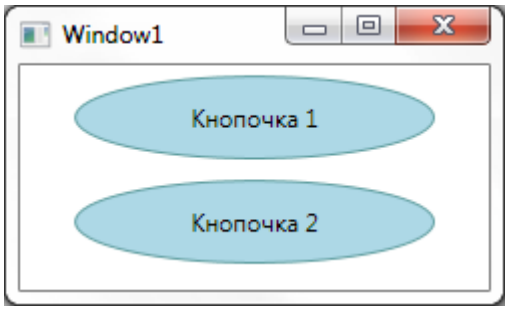
На данном этапе кнопка никак внешне не реагирует на наведение на нее курсора или клик. Это решается при помощи триггеров, которые будут рассмотрены в следующем уроке.

Чтобы шаблон действовал для всех кнопок, его нужно связать со стилем:

```
<Window.Resources>
    <ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
       <Grid>
            <Ellipse Height="42" Stroke="CadetBlue" Width="180"
             Fill="LightBlue"/>
```

```
            <ContentPresenter HorizontalAlignment="Center"
             VerticalAlignment="Center"></ContentPresenter>
        </Grid>
    </ControlTemplate>

    <Style TargetType="{x:Type Button}">
      <Setter Property="Template"
        Value="{StaticResource ButtonTemplate}"/>
     </Style>
</Window.Resources>

<StackPanel>
    <Button Name="button1" Margin="5">Кнопочка 1</Button>
    <Button Name="button2" Margin="5">Кнопочка 2</Button>
</StackPanel>
```

В результате получим:



Шаблоны, в отличие от стилей, не наследуются друг от друга. Поэтому, если нужно создать похожий шаблон, придется делать копию исходного шаблона и потом ее изменять.

Очень полезной возможностью шаблонов является расширение разметки **TemplateBinding**.

Если в предыдущем примере попытаться изменить цвет фона кнопки,

```
<Button Name="button1" Margin="5" Background="Red">Кнопочка 1</Button>
```

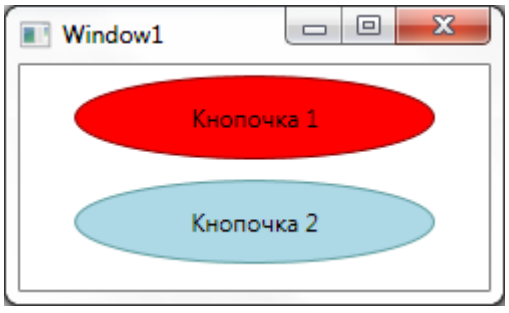
то ничего не произойдет, потому что WPF не знает где применять этот фон. Чтобы связать свойство элемента с шаблоном, применяется **TemplateBinding**.

Изменим наш пример так, чтобы можно было менять свойства **Background** и **BorderBrush** для кнопки. Для этого переопределим в

шаблоне для эллипса свойства **Stroke** и **Fill** при помощи **TemplateBinding** и привяжем их к свойствам кнопки (**Background** и **BorderBrush**). После этого в стиле укажем цвета по умолчанию для свойств **Background** и **BorderBrush**, иначе цвета будут системными. После этого можно смело переопределять их в свойствах кнопки

```
<Window.Resources>
  <ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
    <Grid>
      <Ellipse Height="42"
      Stroke="{TemplateBinding BorderBrush}" Width="180"
      Fill="{TemplateBinding Background}"/>
         <ContentPresenter HorizontalAlignment="Center"
          VerticalAlignment="Center"></ContentPresenter>
    </Grid>
  </ControlTemplate>

  <Style TargetType="{x:Type Button}">
    <Setter Property="Template" Value="{StaticResource ButtonTemplate}"/>
    <Setter Property="Background" Value="LightBlue"/>
    <Setter Property="BorderBrush" Value="CadetBlue"/>
  </Style>
</Window.Resources>

<StackPanel>
  <Button Name="button1" Margin="5"
   Background="Red" BorderBrush="DarkRed">Кнопочка 1</Button>
  <Button Name="button2" Margin="5">Кнопочка 2</Button>
</StackPanel>
```

В результате получим:



Библиотеки ресурсов (Resource Dictionary)

Как уже упоминалось ранее в разделе Ресурсы, если нужно, чтобы ресурсы были доступны для всего приложения, то нужно их объявлять в **App.xaml**.

Если объявлений не много, то это выход из положения, но если ресурсов много, то можно превратить **App.xaml** в абсолютно нечитабельный файл.

Чтобы избежать этого в **WPF** можно при помощи **библиотек ресурсов** (**Resource Dictionary**). Кроме того, библиотеки ресурсов удобны при применении одного дизайна для нескольких приложений – достаточно просто скопировать файлы библиотек.

Библиотека ресурсов представляет собой файл с расширением **xaml**.

Для подключения библиотеки ресурсов в Microsoft Visual Studio необходимо выбрать пункт меню **Project → Add Resource Dictionary** для создания новой библиотеки ресурсов, либо **Project → Add Existing Item** для подключения уже существующей библиотеки к проекту.

При создании новой библиотеки ресурсов появится файл со следующим содержанием:

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

</ResourceDictionary>
```

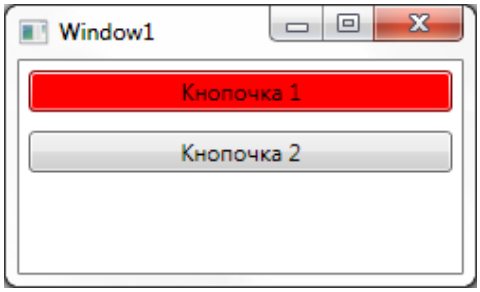
Далее мы можем наполнять файл библиотеки ресурсов ресурсами точно так же, как мы наполняли ресурсами свойство **Resources** в программе.

Перенесем ресурсы кнопок из последнего примера в словарь ресурсов с названием **Buttons.xaml**:

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
```

```
        <Grid>
            <Ellipse Height="42" Stroke="{TemplateBinding BorderBrush}"
             Width="180" Fill="{TemplateBinding Background}"/>
            <ContentPresenter HorizontalAlignment="Center"
             VerticalAlignment="Center"></ContentPresenter>
        </Grid>
    </ControlTemplate>

    <Style TargetType="{x:Type Button}">
        <Setter Property="Template" Value="{StaticResource ButtonTemplate}"/>
        <Setter Property="Background" Value="LightBlue"/>
        <Setter Property="BorderBrush" Value="CadetBlue"/>
    </Style>

</ResourceDictionary>
```

Несмотря на то, что файл библиотеки ресурсов подключен к проекту, приложение будет выглядеть следующим образом:



Это происходит потому, что библиотеки ресурсов автоматически не подгружаются, и их нужно подключать на том уровне, где это необходимо. То есть, если библиотека используется только в одном окне, тогда ее нужно подключать в этом окне. В случае, если библиотека ресурсов используется часто в разных окнах, имеет смысл подключить ее на уровне приложения (в файле **App.xaml**).

Подключаются библиотеки следующим образом:

```
<Window.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Buttons.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
 </Window.Resources>
```

В результате получим ожидаемый результат:

Скины

При разработке Windows приложения программист определяет его внешний вид, так называемый UI (User Interface). При этом можно предусмотреть возможность изменения внешнего вида программы в зависимости от некоторых внешних факторов (системных цветов, темы операционной системы, действий пользователя и так далее).

Одним из типичных действий пользователя является смена так называемого скина (skin). **Skin** в переводе с английского **кожа**. Что такое скин с точки зрения разработчика? Скин – это оформление программы, с помощью некоторого набора стилей. Обычно физически скины представляются в виде файла находящегося на диске. Примером использования скинов может служить широко известное приложение **Winamp**.

Пример скина **Winamp Classic:**

Поставим другой скин **Winamp Modern**:

В результате смены скина **Winamp** не меняет свою функциональность, а только лишь демонстрирует новый внешний вид. Важно отметить, что не каждое приложение поддерживает систему скинов. Для того чтобы данный механизм работал необходимо написать соответствующий программный код.

Реализация скинов в WPF происходит при помощи словарей ресурсов, рассмотренных выше. Замена скинов – это всего лишь программная замена словаря ресурсов. Рассмотрим, как это делается на примере.

Для начала создадим само приложение. Причем сразу создадим библиотеку ресурсов **DefaultStyle.xaml**, которую будем подключать в начале работы.

Файл окна Window1.xaml

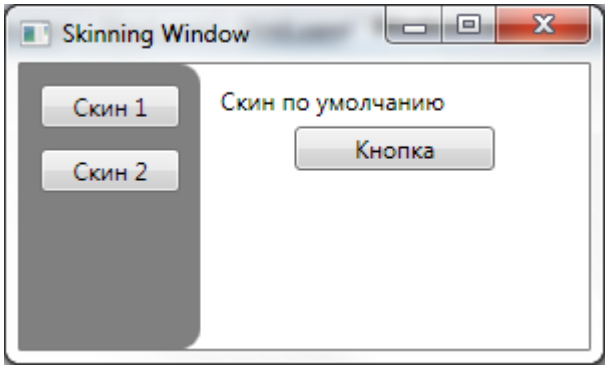
```
<Window x:Class="SkinningDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Skinning Window" Height="180" Width="300">

    <Window.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="DefaultSkin.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Window.Resources>

    <DockPanel>
        <Border Background="Gray" CornerRadius="0,10,10,0">
            <StackPanel Margin="5">
                <Button Margin="5" Width="70" Click="Button_Click_1">
                 Скин 1</Button>
                <Button Margin="5" Width="70" Click="Button_Click_2">
                 Скин 2</Button>
            </StackPanel>
```

```
        </Border>
        <Border>
            <StackPanel Margin="5">
                <Label Name="LabelSkin">Скин по умолчанию</Label>
                <Button Width="100" Style="{DynamicResource
                 ButtonStyle}">Кнопка</Button>
            </StackPanel>
        </Border>
    </DockPanel>
</Window>
```

Так как в первоначальном виде ничего не изменялось, то файл DefaultStyle.xaml выглядит следующим образом:

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Style TargetType="{x:Type Button}"  x:Key="ButtonStyle">

    </Style>
 </ResourceDictionary>
```

Программа будет выглядеть следующим образом:



Далее добавляем файлы скинов в проект (Skin1.xaml, Skin2.xaml).

Код Skin1.xaml:

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Style TargetType="{x:Type DockPanel}">
        <Setter Property="Background" Value="Cornsilk"/>
    </Style>

    <LinearGradientBrush x:Key="BrushForButton" StartPoint="0,0"
     EndPoint="0,1">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="DarkOrange" Offset="0.9"/>
    </LinearGradientBrush>

    <Style TargetType="{x:Type Button}"  x:Key="ButtonStyle">
```

```
        <Setter Property="Background"  Value="{DynamicResource
        BrushForButton}"/>
    </Style>

    <Style TargetType="{x:Type Label}">
        <Setter Property="FontSize" Value="18"/>
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
    </Style>
 </ResourceDictionary>
```

Код Skin2.xaml:

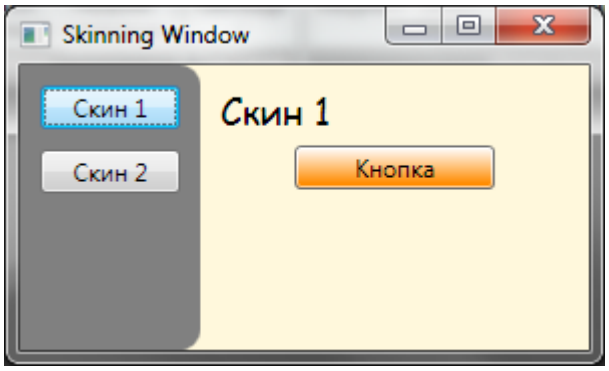
```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Style TargetType="{x:Type DockPanel}">
        <Setter Property="Background" Value="Black"/>
    </Style>

    <LinearGradientBrush x:Key="BrushForButton" StartPoint="0,0"
     EndPoint="0,1">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="Gray" Offset="0.9"/>
    </LinearGradientBrush>

    <Style TargetType="{x:Type Button}"  x:Key="ButtonStyle">
        <Setter Property="Background"  Value="{DynamicResource
        BrushForButton}"/>
    </Style>

    <Style TargetType="{x:Type Label}">
        <Setter Property="FontSize" Value="20"/>
        <Setter Property="FontFamily" Value="Times New Roman"/>
        <Setter Property="Foreground" Value="White"/>
    </Style>
</ResourceDictionary>
```

Переходим к основной цели нашей программы – динамической смене скинов. Для этого в обработчиках кнопок «Скин 1» и «Скин 2» пишем следующий код:

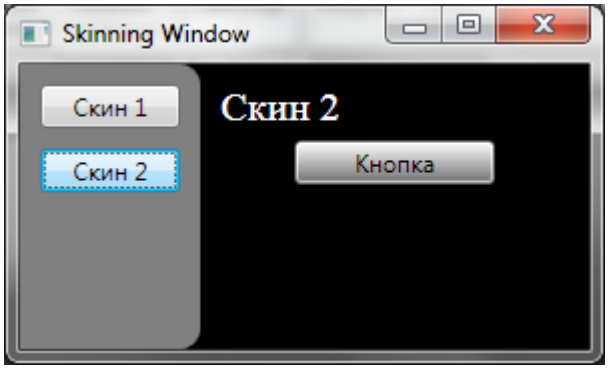
```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    ResourceDictionary newDictionary = new ResourceDictionary();
    newDictionary.Source = new Uri("Skin1.xaml", UriKind.Relative);
    this.Resources.MergedDictionaries[0] = newDictionary;
    LabelSkin.Content = "Скин 1";
}

private void Button_Click_2(object sender, RoutedEventArgs e)
{
    ResourceDictionary newDictionary = new ResourceDictionary();
    newDictionary.Source = new Uri("Skin2.xaml", UriKind.Relative);
    this.Resources.MergedDictionaries[0] = newDictionary;
    LabelSkin.Content = "Скин 2";
```

```
}
```

Как видно из кода, в каждом обработчике мы просто подменяем текущую библиотеку ресурсов на другую. В результате при нажатии на кнопку «Скин 1» мы получим:



А при нажатии на «Скин 2» увидим на экране:



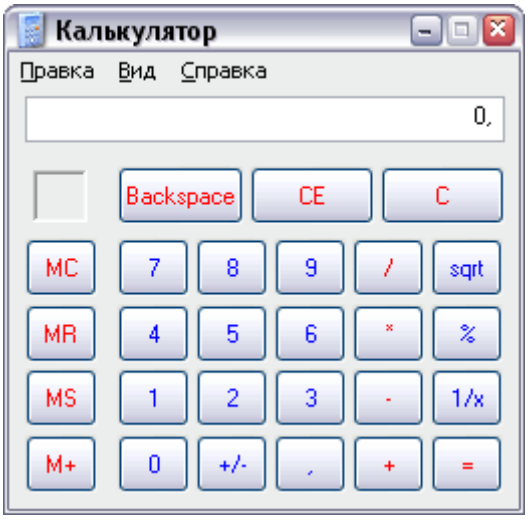
Темы

Хотя многие путают понятия «скины» и «темы», под темами в WPF понимаются темы операционной системы Windows:

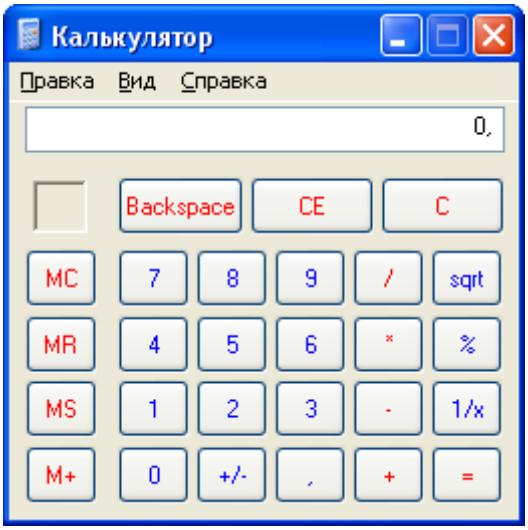
Тема	Описание
Classic	Классическая тема в Windows XP и Vista, Windows 7
Aero	Стиль оформления Windows Vista, Windows 7 (NormalColor)
Luna	Стиль оформления Windows XP (синяя (NormalColor), оливковая (Homestead) и серебристая (Metallic))
Royale	Стиль оформления Windows Media Center Edition (NormalColor).

Для каждой темы у WPF есть библиотека ресурсов, где описано как должны выглядеть элементы управления. Каждый раз при изменении темы Windows меняются шаблоны элементов управления в соответствии с текущей темой.

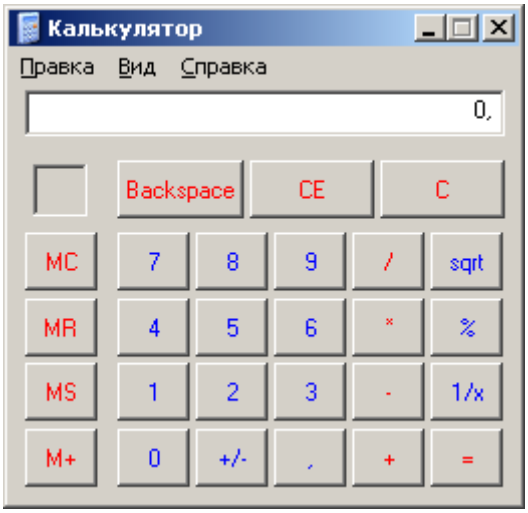
Например, запустим в операционной системе Windows XP одно из типичных приложений – калькулятор.



Так он выглядит при текущей установленной теме (Luna серебристая (metallic)), изменим тему на другую (Luna синяя):



Поставим классическую тему и видим:



То есть мы увидели, что внешний вид калькулятора меняется в зависимости от установленной темы.

Если мы хотим, чтобы вид приложения менялся в зависимости от темы Windows, то WPF дает нам и такую возможность. Для этого в проекте надо создать папку **Themes**.

Далее эту папку нужно наполнить набором библиотек ресурсов. Главное при этом xaml-файлы называть определенным образом – **Название_темы.Цвет.xaml** за исключением классической темы:

Тема	Название xaml-файла	
Classic	Classic.xaml	
Aero	Aero.NormalColor.xaml	
Luna	Синяя	Luna.NormalColor.xaml
	Оливковая	Luna.Homestead.xaml
	серебристая	Luna.Metallic.xaml
Royale	Royale.NormalColor.xaml	

Но для того, чтобы программа адекватно реагировала на темы, недостаточно только создать библиотеки. Необходимо в файле **AssemblyInfo.cs** найти следующую строчку:

```
[assembly: ThemeInfo(
    ResourceDictionaryLocation.None, //where theme specific resource
    //dictionaries are located
    //(used if a resource is not found in the page,
```

```
    // or application resource dictionaries)
    ResourceDictionaryLocation.SourceAssembly //where the generic resource
    //dictionary is located
    //(used if a resource is not found in the page,
    // app, or any theme specific resource dictionaries)
)]
```

И заменить ее на следующую:

```
[assembly: ThemeInfo(
    ResourceDictionaryLocation.SourceAssembly, //where theme
    //specific resource dictionaries are located
    //(used if a resource is not found in the page,
    // or application resource dictionaries)
    ResourceDictionaryLocation.SourceAssembly //where the generic resource
    //dictionary is located
    //(used if a resource is not found in the page,
    // app, or any theme specific resource dictionaries)
)]
```

А кроме этого необходимо подключить библиотеки тем следующим образом:

```
<ResourceDictionary>
       <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="{ThemeDictionary ThemeDemo}" />
       </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

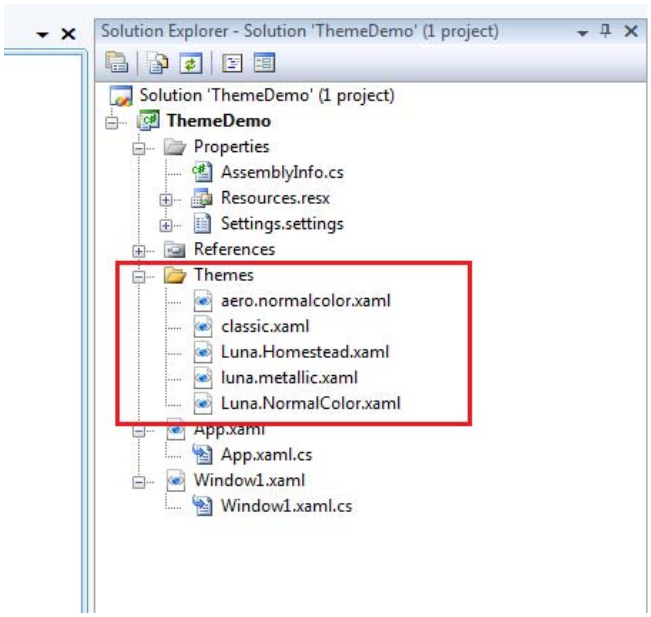
Попробуем создать пример, чтобы при классической теме Windows он использовал одни стили, а если используется тема Aero, то другие стили.

Окно выглядит следующим образом (код XAML):

```
<Window x:Class="ThemeDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="150" Width="300">

    <StackPanel>
        <TextBox Name="textBox1" Width="200" Margin="5">какой-то
текст</TextBox>
        <Button Height="23" Name="button1" Width="75"
Margin="5">Button</Button>
    </StackPanel>

  </Window>
```

Для примера мы на форме разместили только текстовое поле и кнопку.

Далее необходимо создать папку **Themes** в проекте, и в ней создать файлы для всех тем. Даже если не планируете прописывать ресурсы для какой-либо темы, создайте пустой словарь ресурсов.



Потом пропишем ресурсы для классического вида Windows (classic.xaml).

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Style TargetType="{x:Type StackPanel}">
        <Setter Property="Background" Value="DarkBlue"/>
    </Style>
    <LinearGradientBrush x:Key="BrushForButton" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="Blue" Offset="0"/>
        <GradientStop Color="White" Offset="0.9"/>
    </LinearGradientBrush>
    <Style TargetType="{x:Type Button}">
        <Setter Property="Background"  Value="{DynamicResource
BrushForButton}"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="FontSize" Value="14"/>
        <Setter Property="FontFamily" Value="Times New Roman"/>
    </Style>
 </ResourceDictionary>
```

Ресурсы для Aero (aero.normalcolor.xaml).

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

```
    <LinearGradientBrush x:Key="BrushForButton" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="DarkOrange" Offset="0.9"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="BrushForPanel" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="Yellow" Offset="0"/>
        <GradientStop Color="White" Offset="0.9"/>
    </LinearGradientBrush>
    <Style TargetType="{x:Type Button}">
        <Setter Property="Background"  Value="{DynamicResource
BrushForButton}"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="FontSize" Value="20"/>
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
    </Style>
    <Style TargetType="{x:Type StackPanel}">
        <Setter Property="Background" Value="{DynamicResource
BrushForPanel}"/>
    </Style>
</ResourceDictionary>
```

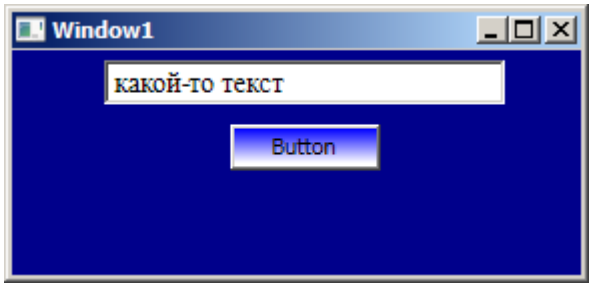
В App.xaml подключаем библиотеки ресурсов для тем Windows.

```
<Application x:Class="ThemeDemo.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="{ThemeDictionary ThemeDemo}" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

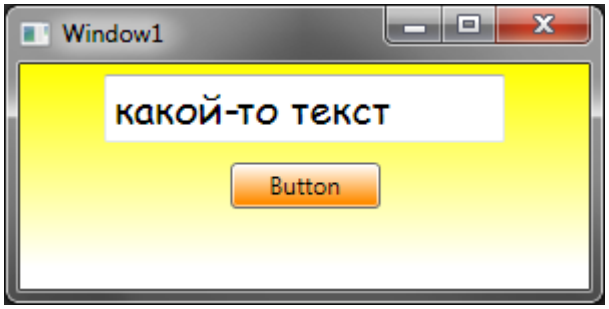
И в конце пропишем ThemeInfo в AssemblyInfo.cs

```
[assembly: ThemeInfo(
    ResourceDictionaryLocation.SourceAssembly, //where theme specific
resource dictionaries are located
    //(used if a resource is not found in the page,
//[assembly: NeutralResourcesLanguage("en-US",
UltimateResourceFallbackLocation.Satellite)]


    // or application resource dictionaries)
    ResourceDictionaryLocation.SourceAssembly //where the generic resource
dictionary is located
    //(used if a resource is not found in the page,
    // app, or any theme specific resource dictionaries)
  )]
```

В результате при классической теме Windows пример будет выглядеть следующим образом:



А при включенной теме Aero программа автоматически изменит свой вид на следующий:



Для того, чтобы у программы был различный внешний вид при других темах, достаточно прописать стили в остальных библиотеках ресурсов в папке Themes.

Домашнее задание:

1. Реализуйте приложение для просмотра текстовых файлов. Приложение должно уметь загружать текстовые файлы и производить предопределенное форматирование. Например, выделять жирным ключевые слова, ссылки выделять курсивом (используйте элемент Hyperlink), и так далее.
2. Используя связывание данных, реализуйте приложение для оперирования персоналом компании. Приложение должно позволять добавлять, удалять, изменять данные, искать персонал по различным критериям поиска (имя,фамилия,отчество, и т.д.)
3. Добавьте в задания этого урока стили, шаблоны, скины, поддержку тем.