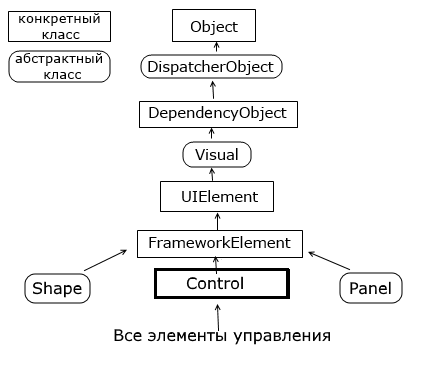
## Обзор элементов управления и их свойств

Чтобы как-то взаимодействовать с пользователем, получать от пользователя ввод с клавиатуры или мыши и использовать введенные данные в программе, нам нужны элементы управления. WPF предлагает нам богатый стандартный набор элементов управления

Все элементы управления могут быть условно разделены на несколько подгрупп:

* **Элементы управления содержимым**, например кнопки (Button), метки (Label)
* **Специальные контейнеры**, которые содержат другие элементы, но в отличие от элементов Grid или Canvas не являются контейнерами компоновки - ScrollViewer,GroupBox
* **Декораторы**, чье предназначение создание определенного фона вокруг вложенных элементов, например, Border или Viewbox.
* **Элементы управления списками**, например, ListBox, ComboBox.
* **Текстовые элементы управления**, например, TextBox, RichTextBox.
* **Элементы, основанные на диапазонах значений**, например, ProgressBar, Slider.
* **Элементы для работ с датами**, например, DatePicker и Calendar.
* **Остальные элементы управления**, которые не вошли в предыдущие подгруппы, например, Image.

Все элементы управления наследуются от общего класса **System.Window.Controls.Control** и имеют ряд общих свойств. А общую иерархию элементов управления можно представить следующим образом:



Вкратце рассмотрим, что представляют все эти типы в иерархии.

### System.Threading.DispatcherObject

В основе WPF лежит модель STA (Single-Thread Affinity), согласно которой за пользовательский интерфейс отвечает один поток. И чтобы пользовательский интерфейс мог взаимодействовать с другими потоками, WPF использует концепцию диспетчера - специального объекта, управляющего обменом сообщениями, через которые взаимодействуют потоки. Наследование типов от класса DispatcherObject позволяет получить доступ к подобному объекту-диспетчеру и и другим функциям по управлению параллелизмом.

### System.Windows.DependencyObject

Наследование от этого класса позволяет взаимодействовать с элементами в приложении через их специальную модель свойств, которые называются свойствами зависимостей (dependency properties). Эта модель упрощает применение ряда особенностей WPF, например, привязки данных. Так, система свойств зависимостей отслеживает зависимости между значениями свойств, автоматически проверяет их и изменяет при изменении зависимости.

### System.Windows.Media.Visual

Класс Visual содержит инструкции, которые отвечают за отрисовку, визуализацию объекта.

### System.Windows.UIElement

Класс UIElement добавляет возможности по компоновке элемента, обработку событий и получение ввода.

### System.Windows.FrameworkElement

Класс FrameworkElement добавляет поддержку привязки данных, анимации, стилй. Также добавляет ряд свойств, связанных с компоновкой (выравнивание, отступы) и ряд других.

### System.Windows.Controls.Control

Класс Control представляет элемент управления, с которым взаимодействует пользователь. Этот класс добавляет ряд дополнительных свойств для поддержки элементами шрифтов, цветов фона, шрифта, а также добавляет поддержку шаблонов - специального механизма в WPF, который позволяет изменять стандартное представление элемента, кастомизировать его.

И далее от класса Control наследуются непосредственно конкретные элементы управления или их базовые классы, которые получают весь функционал, добавляемый к типам в этой иерархии классов.

Рассмотрим некоторые из основных свойств, которые наследуются элементами управления.

### Name

Наверное важнейшее свойство. По установленному имени впоследствии можно будет обращаться к элементу, как в коде, так и в xaml разметке. Например, в xaml-коде у нас определена следующая кнопка:

|  |  |
| --- | --- |
| 1 | <Button x:Name="button1" Width="60" Height="30" Content="Текст" Click="button1\_Click" /> |

Здесь у нас задан атрибут Click с названием метода обработчика button1\_Click, который будет определен в файле кода C# и будет вызываться по нажатию кнопки. Тогда в связанном файле кода C# мы можем обратиться к этой кнопке:

|  |  |
| --- | --- |
| 1  2  3  4 | private void button1\_Click(object sender, RoutedEventArgs e)  {      button1.Content = "Привет!";  } |

Поскольку свойство Name имеет значение button1, то через это значение мы можем обратиться к кнопке в коде.

### FieldModifier

Свойство FieldModifier задает модификатор доступа к объекту:

|  |  |
| --- | --- |
| 1  2  3  4 | <StackPanel>      <Button x:FieldModifier="private" x:Name="button1" Content="Hello World" />      <Button x:FieldModifier="internal" x:Name="button2" Content="Hello WPF" />  </StackPanel> |

В качестве значения используются стандартные модификатора доступа языка C#: private, protected, internal, protected internal и public. В данном случае объявление кнопок с модификаторами будет равноценно следующему их определению в коде:

|  |  |
| --- | --- |
| 1  2 | private Button button1;  internal Button button2; |

Если для элемента не определен атрибут x:FieldModifier, то по умолчанию он равен "protected internal".

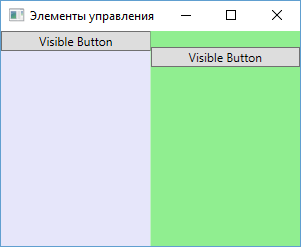
### Visibility

Это свойство устанавливает параметры видимости элемента и может принимать одно из трех значений:

* **Visible** - элемент виден и участвует в компоновке.
* **Collapsed** - элемент не виден и не участвует в компоновке.
* **Hidden** - элемент не виден, но при этом участвует в компоновке.

Различия между Collapsed и Hidden можно продемонстрировать на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <Grid>      <Grid.ColumnDefinitions>          <ColumnDefinition Width="\*" />          <ColumnDefinition Width="\*" />      </Grid.ColumnDefinitions>      <StackPanel Grid.Column="0" Background="Lavender">          <Button Visibility="Collapsed" Content="Панель Collapsed" />          <Button Height="20" Content="Visible Button" />      </StackPanel>      <StackPanel Grid.Column="1" Background="LightGreen">          <Button Visibility="Hidden" Content="Панель Hidden" />          <Button Height="20" Content="Visible Button" />      </StackPanel>  </Grid> |



### Свойства настройки шрифтов

* **FontFamily** - определяет семейство шрифта (например, Arial, Verdana и т.д.)
* **FontSize** - определяет высоту шрифта
* **FontStyle** - определяет наклон шрифта, принимает одно из трех значений - **Normal**, **Italic**,**Oblique**.
* **FontWeight** - определяет толщину шрифта и принимает ряд значений, как **Black**,**Bold** и др.
* **FontStretch** - определяет, как будет растягивать или сжимать текст, например, значение **Condensed** сжимает текст, а **Expanded** - расстягивает.

Например:

|  |  |
| --- | --- |
| 1 | <Button Content="Hello World!" FontFamily="Verdana" FontSize="13" FontStretch="Expanded" /> |

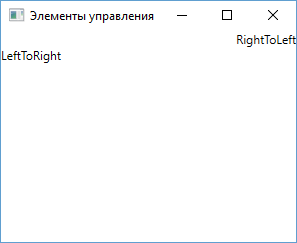
### Cursor

Это свойство позволяет нам получить или установить курсор для элемента управления в одно из значений, например, **Hand**, **Arrow**, **Wait** и др. Например, установка курсора в коде c#: button1.Cursor=Cursors.Hand;

### FlowDirection

Данное свойство задает направление текста. Если оно равно RightToLeft, то текст начинается с правого края, если - LeftToRight, то с левого.

|  |  |
| --- | --- |
| 1  2  3  4 | <StackPanel>      <TextBlock FlowDirection="RightToLeft">RightToLeft</TextBlock>      <TextBlock FlowDirection="LeftToRight">LeftToRight</TextBlock>  </StackPanel> |



### Цвета фона и шрифта

Свойства **Background** и **Foreground** задают соответственно цвет фона и текста элемента управления.

Простейший способ задания цвета в коде xaml: Background="#ffffff". В качестве значения свойство Background (Foreground) может принимать запись в виде шестнадцатеричного значения в формате #rrggbb, где rr - красная составляющая, gg - зеленая составляющая, а bb - синяя. Также можно задать цвет в формате #aarrggbb.

Либо можно использовать названия цветов напрямую:

|  |  |
| --- | --- |
| 1 | <Button Width="60" Height="30" Background="LightGray" Foreground="DarkRed" Content="Цвет" /> |

Однако при компиляции будет создаваться объект SolidColorBrush, который и будет задавать цвет элемента. То есть определение кнопки выше фактически будет равноценно следующему:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <Button Width="60" Height="30" Content="Цвет">      <Button.Background>          <SolidColorBrush Color="LightGray" />      </Button.Background>      <Button.Foreground>          <SolidColorBrush Color="DarkRed" />      </Button.Foreground>  </Button> |

SolidColorBrush представляет собой кисть, покрывающую элемент одним цветом. Позже мы подробнее поговорим о цветах. А пока надо знать, что эти записи эквивалентны, кроме того, вторая форма определения цвета позволяет задать другие кисти - например, градиент.

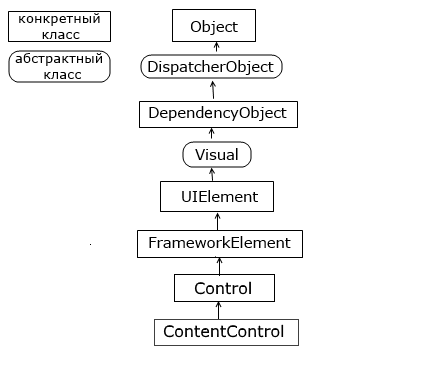
Это надо также учитывать при установке или получении цвета элемента в коде c#:

|  |  |
| --- | --- |
| 1  2 | button1.Background = new SolidColorBrush(Colors.Red);  button1.Foreground = new SolidColorBrush(Color.FromRgb(0,255, 0)); |

Класс Colors предлагает ряд встроенный цветовых констант, которыми мы можем воспользоваться. А если мы захотим конкретизировать настройки цвета с помощью значений RGB, то можно использовать метод **Color.FromRgb**.

## Элементы управления содержимым

Элементы управления содержимым (content controls) представляют такие элементы управления, которые содержат в себе другой элемент. Все элементы управления содержимым наследуются от класса **ContentControl**, который в свою очередь наследуется от класса System.Window.Controls.Control.



К элементам управления содержимым относятся такие элементы как Button, Label, ToggleButton, ToolTip, RadioButton, CheckBox, GroupBox, TabItem, Expander, ScrollViewer. Также элементом управления содержимым является и главный элемент окна - Window.

Отличительной чертой всех этих элементов является наличие свойства **Content**, которое и устанавливает вложенный элемент. В этом элементы управления содержимым схожи с контейнерами компоновки. Только контейнеры могут иметь множество вложенных элементов, а элементы управления содержимым только один.

Свойство Content может представлять любой объект, который может относиться к одному из двух типов:

* Объект класса, не наследующего от UIElement. Для такого объекта вызывается метод ToString(), который возвращает строковое преставление объекта. Затем эта строка устанавливается в качестве содержимого.
* Объект класса, наследующего от UIElement. Для такого объекта вызывается метод UIElement.OnRender(), который выполняет отрисовку внутри элемента управления содержимым.

Рассмотрим на примере кнопки, которая является элементом управления содержимым:

|  |  |
| --- | --- |
| 1 | <Button Content="Hello World!" /> |

В качестве содержимого устанавливается обычная строка. Этот же пример мы можем в XAML прописать иначе:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <Button>      <Button.Content>          Hello World!      </Button.Content>  </Button> |

Либо мы можем использовать сокращенное неявное определения свойства Content:

|  |  |
| --- | --- |
| 1  2  3 | <Button>      Hello World!  </Button> |

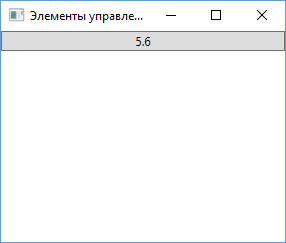
Возьмем другой пример. Определим кнопку с именем button1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <Window x:Class="ControlsApp.MainWindow"          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"          xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"          xmlns:local="clr-namespace:ControlsApp"          mc:Ignorable="d"          Title="Элементы управления" Height="250" Width="300">      <StackPanel>          <Button x:Name="button1" />      </StackPanel>  </Window> |

А в файле коде MainWindow.xaml.cs присвоим ее свойству Content какой-либо объект:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using System;  using System.Windows;    namespace ControlsApp  {      public partial class MainWindow : Window      {          public MainWindow()          {              InitializeComponent();              double d = 5.6;              button1.Content = d;          }      }  } |

В итоге мы получим следующую кнопку:

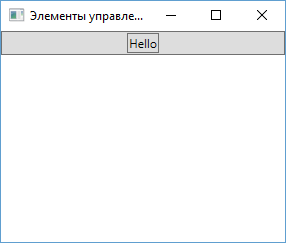


В итоге число конвертируется в строку и устанавливается в качесте содержимого.

Иначе все будет работать, если мы в качестве содержимого используем объект, унаследованный от UIElement:

|  |  |
| --- | --- |
| 1  2  3 | <Button x:Name="button1">      <Button Content="Hello" />  </Button> |

Теперь в качестве содержимого будет использоваться другая кнопка, для которой при визуализации будет вызываться меод OnRender():

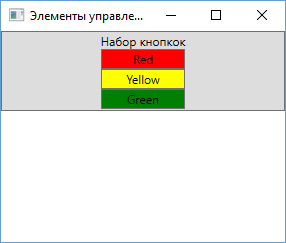


Для создания той же кнопки через код C# мы бы могли прописать следующее выражение:

|  |  |
| --- | --- |
| 1 | button1.Content = new Button { Content = "Hello" }; |

В отличие от контейнеров компоновки для элементов управления содержимым мы можем задать только один вложенный элемент. Если же нам надо вложить в элемент управления содержимым несколько элементов, то мы можем использовать те же контейнеры компоновки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <Button x:Name="button1">      <StackPanel>          <TextBlock Text="Набор кнопкок" />          <Button Background="Red" Height="20" Content="Red" />          <Button Background="Yellow" Height="20" Content="Yellow" />          <Button Background="Green" Height="20" Content="Green" />      </StackPanel>  </Button> |



То же самое мы могли бы прописать через код C#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | StackPanel stackPanel = new StackPanel();  stackPanel.Children.Add(new TextBlock { Text = "Набор кнопок" });  stackPanel.Children.Add(new Button { Content = "Red", Height = 20, Background = new SolidColorBrush(Colors.Red) });  stackPanel.Children.Add(new Button { Content = "Yellow", Height = 20, Background = new SolidColorBrush(Colors.Yellow) });  stackPanel.Children.Add(new Button { Content = "Green", Height = 20, Background = new SolidColorBrush(Colors.Green) });  button1.Content = stackPanel; |

### Позиционирование контента

#### Content Alignment

Выравнивание содержимого внутри элемента задается свойствами **HorizontalContentAlignment** (выравнивание по горизонтали) и **VerticalContentAlignment** (выравнивание по вертикали), аналогичны свойствам VerticalAlignment/HorizontalAlignment. Свойство HorizontalContentAlignment принимает значения Left, Right, Center (положение по центру), Stretch (растяжение по всей ширине). Например:

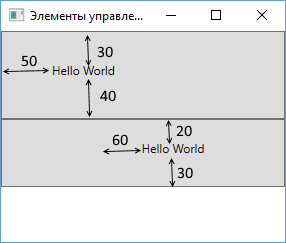
|  |  |
| --- | --- |
| 1  2  3  4  5 | <StackPanel>      <Button Margin="5" HorizontalContentAlignment="Left" Content="Left" Height="90" Width="500" />      <Button Margin="5" HorizontalContentAlignment="Right" Content="Right" Height="90" Width="500" />      <Button Margin="5" HorizontalContentAlignment="Center" Content="Center" Height="90" Width="500" />  </StackPanel> |

VerticalContentAlignment принимает значения Top (положение в верху), Bottom (положение внизу), Center (положение по центру), Stretch (растяжение по всей высоте)

#### Padding

С помощью свойства Padding мы можем установить отступ содержимого элемента:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <StackPanel>      <Button x:Name="button1" Padding="50 30 0 40" HorizontalContentAlignment="Left">          Hello World      </Button>      <Button x:Name="button2" Padding="60 20 0 30" HorizontalContentAlignment="Center">          Hello World      </Button>  </StackPanel> |



Свойство Padding задается в формате Padding="отступ\_слева отступ\_сверху отступ\_справа отступ\_снизу".

Если со всех четырех сторон предполагается один и тот же отступ, то, как и в случае с Margin, мы можем задать одно число:

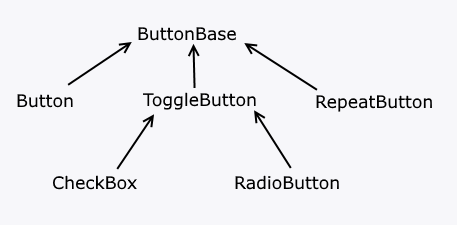
|  |  |
| --- | --- |
| 1 | <Button x:Name="button2" Padding="20"  Content="Hello World" /> |

Важно понимать, от какой точки задается отступ. В случае с первой кнопкой в ней контект выравнивается по левому краю, поэтому отступ слева будет предполагать отступ от левого края элемента Button. А вторая кнопка располагается по центру. Поэтому для нее отступ слева предполагает отступ от той точки, в которой содержимое бы находилось при центрировании без применения Padding.

Комбинация значений свойств HorizontalContentAlignment/VerticalContentAlignment и Padding позволяет оптимальным образом задать расположение содержимого.

## Кнопки

 WPF кнопки представлены целым рядом классов, которые наследуются от базового класса ButtonBase:



### Button

Элемент Button представляет обычную кнопку:

|  |  |
| --- | --- |
| 1 | <Button x:Name="button1" Width="60" Height="30" Background="LightGray" /> |

От класса ButtonBase кнопка наследует ряд событий, например, Click, которые позволяют обрабатывать пользовательский ввод.

Чтобы связать кнопку с обработчиком события нажатия, нам надо определить в самой кнопке атрибут Click. А значением этого атрибута будет название обработчика в коде C#. А затем в самом коде C# определить этот обработчик.

Например, код xaml:

|  |  |
| --- | --- |
| 1 | <Button x:Name="button1" Width="60" Height="30" Content="Нажать" Click="Button\_Click" /> |

И обработчик в коде C#:

|  |  |
| --- | --- |
| 1  2  3  4 | private void Button\_Click(object sender, RoutedEventArgs e)  {      MessageBox.Show("Кнопка нажата");  } |

Либо можно не задавать обработчик через атрибут, а стандартным образом для C# прописать в коде: button1.Click+=Button\_Click;

К унаследованным свойствам кнопка имеет такие свойства как **IsDefault** и **IsCancel**, которые принимают значения true и false.

Если свойство IsDefault установлено в true, то при нажатии клавиши Enter будет вызываться обработчик нажатия этой кнопки.

Аналогично если свойство IsCancel будет установлено в true, то при нажатии на клавишу Esc будет вызываться обработчик нажатия этой кнопки.

Например, определим код xaml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <Window x:Class="ControlsApp.MainWindow"          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"          xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"          xmlns:local="clr-namespace:ControlsApp"          mc:Ignorable="d"          Title="Элементы управления" Height="250" Width="300">      <StackPanel>          <Button x:Name="acceptButton" Content="ОК" IsDefault="True" Click="acceptButton\_Click" />          <Button x:Name="escButton" Content="Выход" IsCancel="True" Click="escButton\_Click" />      </StackPanel>  </Window> |

А в коде MainWindow.xaml.cs определим следующий код C#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System.Windows;    namespace ControlsApp  {      public partial class MainWindow : Window      {          public MainWindow()          {              InitializeComponent();          }            private void acceptButton\_Click(object sender, RoutedEventArgs e)          {              MessageBox.Show("Действие выполнено");          }            private void escButton\_Click(object sender, RoutedEventArgs e)          {              this.Close(); // закрытие окна          }      }  } |

Теперь при нажатии на клавишу Enter будет отображаться сообщение, а при нажатии на Esc будет происходить выход из приложения и закрытие окна.

### RepeatButton

Отличительная особенность элемента RepeatButton - непрерывная генерация события Click, пока нажата кнопка. Интервал генерации события корректируется свойствами **Delay** и **Interval**.

Сам по себе элемент RepeatButton редко используется, однако он может служить основой для создания ползунка в элементах ScrollBar и ScrollViewer, в которых нажатие на ползунок инициирует постоянную прокрутку.

### ToggleButton

Представляет элементарный переключатель. Может находиться в трех состояниях - true, false и "нулевом" (неотмеченном) состоянии, а его значение представляет значение типа bool? в языке C#. Состояние можно установить или получить с помощью свойства **IsChecked**. Также добавляет три события - **Checked** (переход в отмеченное состояние), **Unchecked** (снятие отметки) и **Intermediate** (если значение равно null). Чтобы отрабатывать все три события, надо установить свойство **IsThreeState="True"**

ToggleButton, как правило, сам по себе тоже редко используется, однако при этом он служит основой для создания других более функциональных элементов, таких как checkbox и radiobutton.

## v CheckBox и RadioButton

### CheckBox

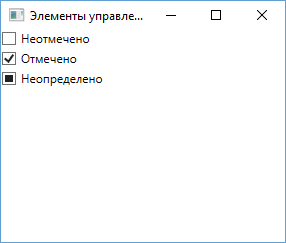
Элемент CheckBox представляет собой обычный флажок. Данный элемент является производным от класса ToggleButton и поэтому может принимать также три состояния: **Checked**, **Unchecked** и **Intermediate**.

Чтобы получить или установить определенное состояние, надо использовать свойство **IsChecked**, которое также унаследовано от ToggleButton:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <StackPanel x:Name="stackPanel">      <CheckBox x:Name="checkBox1" IsThreeState="True" IsChecked="False" Height="20" Content="Неотмечено" />      <CheckBox x:Name="checkBox2" IsThreeState="True" IsChecked="True" Height="20" Content="Отмечено" />      <CheckBox x:Name="checkBox3" IsThreeState="True" IsChecked="{x:Null}" Height="20" Content="Неопределено"/>  </StackPanel> |

Установка свойства IsChecked="{x:Null}" задает неопределенное состояние для элемента checkbox. Остальные два состояния задаются с помощью True и False. В данном примере также привязан к двум флажкам обработчик события Checked. Это событие возникает при установке checkbox в отмеченное состояние.

А атрибут IsThreeState="True" указывает, что флажок может находиться в трех состояниях.



Ключевыми событиями флажка являются события **Checked** (генерируется при установке флажка в отмеченное состояние), **Unchecked** (генерируется при снятии отметки с флажка) и **Indeterminate** (флажок переведен в неопределенное состояние). Например, определим флажок:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <CheckBox x:Name="checkBox" IsChecked="False" Height="20" Content="Флажок"      IsThreeState="True"      Unchecked="checkBox\_Unchecked"      Indeterminate="checkBox\_Indeterminate"      Checked="checkBox\_Checked" /> |

А в файле кода C# пропишем для него обработчики:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | private void checkBox\_Checked(object sender, RoutedEventArgs e)  {      MessageBox.Show(checkBox.Content.ToString() + " отмечен");  }    private void checkBox\_Unchecked(object sender, RoutedEventArgs e)  {      MessageBox.Show(checkBox.Content.ToString() + " не отмечен");  }    private void checkBox\_Indeterminate(object sender, RoutedEventArgs e)  {      MessageBox.Show(checkBox.Content.ToString() + " в неопределенном состоянии");  } |

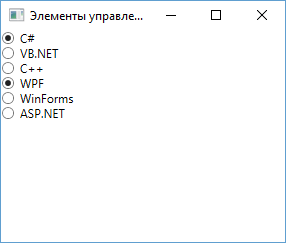
Программное добавление флажка:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | using System.Windows;  using System.Windows.Controls;    namespace ControlsApp  {      public partial class MainWindow : Window      {          public MainWindow()          {              InitializeComponent();              // создаем флажок              CheckBox checkBox2 = new CheckBox { Content = "Новый флажок", MinHeight = 20, IsChecked=true };              // установка обработчика              checkBox2.Checked += checkBox\_Checked;              // добавление в StackPanel              stackPanel.Children.Add(checkBox2);          }            private void checkBox\_Checked(object sender, RoutedEventArgs e)          {              CheckBox chBox = (CheckBox)sender;              MessageBox.Show(chBox.Content.ToString() + " отмечен");          }      }  } |

### RadioButton

Элемент управления, также производный от ToggleButton, представляющий переключатель. Главная его особенность - поддержка групп. Несколько элементов RadioButton можно объединить в группы, и в один момент времени мы можем выбрать из этой группы только один переключатель. Например,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <StackPanel x:Name="stackPanel">      <RadioButton GroupName="Languages" Content="C#" IsChecked="True" />      <RadioButton GroupName="Languages" Content="VB.NET" />      <RadioButton GroupName="Languages" Content="C++" />      <RadioButton GroupName="Technologies" Content="WPF" IsChecked="True" />      <RadioButton GroupName="Technologies" Content="WinForms" />      <RadioButton GroupName="Technologies" Content="ASP.NET" />  </StackPanel> |



Чтобы включить элемент в определенную группу, используется свойство GroupName. В данном случае у нас две группы - Languages и Technologies. Мы можем отметить не более одного элемента RadioButton в пределах одной группы, зафиксировав тем самым выбор из нескольких возможностей.

Чтобы проследить за выбором того или иного элемента, мы также можем определить у элементов событие Checked и его обрабатывать в коде:

|  |  |
| --- | --- |
| 1 | <RadioButton GroupName="Languages" Content="VB.NET" Checked="RadioButton\_Checked" /> |

Обработчик в файле кода:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private void RadioButton\_Checked(object sender, RoutedEventArgs e)  {      RadioButton pressed = (RadioButton)sender;      MessageBox.Show(pressed.Content.ToString());  } |

Программное добавление элемента RadioButton:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using System.Windows;  using System.Windows.Controls;    namespace ControlsApp  {      public partial class MainWindow : Window      {          public MainWindow()          {              InitializeComponent();                RadioButton rb = new RadioButton { IsChecked = true, GroupName = "Languages", Content = "JavaScript" };              rb.Checked += RadioButton\_Checked;              stackPanel.Children.Add(rb);          }            private void RadioButton\_Checked(object sender, RoutedEventArgs e)          {              RadioButton pressed = (RadioButton)sender;              MessageBox.Show(pressed.Content.ToString());          }      }  } |