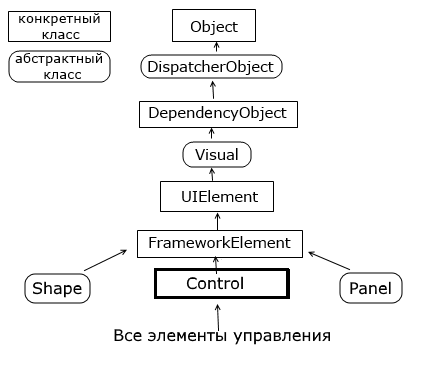
## Обзор элементов управления и их свойств

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".

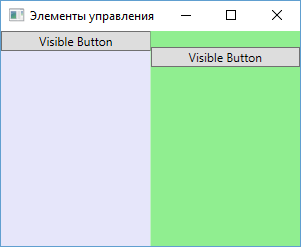
### Visibility

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" /> |

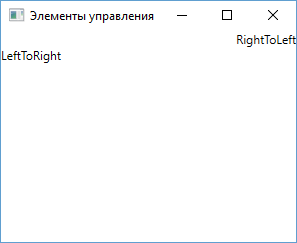
### Cursor

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**.