Урок 5

Знакомство с технологией WPF

Особенности платформы WPF. Введение в язык XAML. Стили. Обзор элементов управления и их свойств.

- Что такое WPF?
  - WPF vs WinForms
    - Достоинства WPF
    - Достоинства WinForms
- Hello, WPF!
  - Работа с App.xaml
  - Запуск приложения с параметрами
- XAML
  - Основы XAML
  - Пространства имен XAML
  - События в XAML
  - Регистрация обработчика событий в файле отделенного кода
- Ресурсы
  - StaticResource vs DynamicResource
  - Локальные ресурсы и ресурсы уровня приложения
  - Доступ к ресурсам из файла отделенного кода
  - Словари ресурсов
- Стили
  - Стили контролов
  - Стили дочерних контролов
  - Стили, определенные для всего диалогового окна
  - Стили, определенные для всего приложения
  - Явное применение стилей
- Обзор элементов управления и их свойств
  - TextBlock
  - Label
  - TextBox
  - Button
  - CheckBox
  - RadioButton
  - ItemsControl
  - ListBox

ComboBox

Window

Практическое задание

Дополнительные материалы

Используемая литература

Что такое WPF?

Технология **WPF** (Windows Presentation Foundation) является частью платформы .NET и представляет собой подсистему для построения графических интерфейсов.

WPF – это комбинация XAML и одного из языков .NET.

XAML (Extensible Application Markup Language) – расширяемый язык разметки приложений.

WPF vs WinForms

Наиболее заметное отличие **WinForms** и **WPF** состоит в том, что **WinForms** представляет собой всего лишь надстройку над стандартными элементами Windows-интерфейса (например, TextBox-ом). **WPF** же создан с нуля и в большинстве случаев не основывается на стандартных элементах Windows.

Пример отличия технологий **WinForms** и **WPF** – **Button** с изображением и текстом. Так как не существует подобного стандартного элемента Windows, изначально WinForms не предоставляют данный элемент. Вместо этого мы вынуждены разрабатывать собственный **Button**, который может содержать изображение или использовать графические компоненты сторонних производителей.

В случае с **WPF Button** может содержать любые элементы, поскольку представляет собой рамку с содержимым и различные состояния элемента (нажата, отжата и т.п.). **WPF Button** не имеет определенного внешнего вида, как и большинство WPF-элементов. Это означает, что они могут содержать набор произвольных элементов.

Чтобы получить **Button** с текстом и изображением, достаточно добавить **Image** и **TextBlock** внутрь **Button**.

Достоинства WPF

- Новая технология, соответствующая текущим стандартам разработки;
- Широко используется компанией Microsoft, в т.ч. при создании VisualStudio;
- Гибкая технология – нет необходимости создавать свои контролы или покупать дополнительные;
- **XAML** облегчает разработку и изменение интерфейса приложения, позволяя разделить эту работу между дизайнером и программистом;
- Технология связывания данных позволяет провести четкую границу между данными и разметкой;
- Использование аппаратного ускорения при отображении GUI увеличивает производительность приложения;
- Возможно создание пользовательских интерфейсов для windows- и web-приложений (SilverLight).

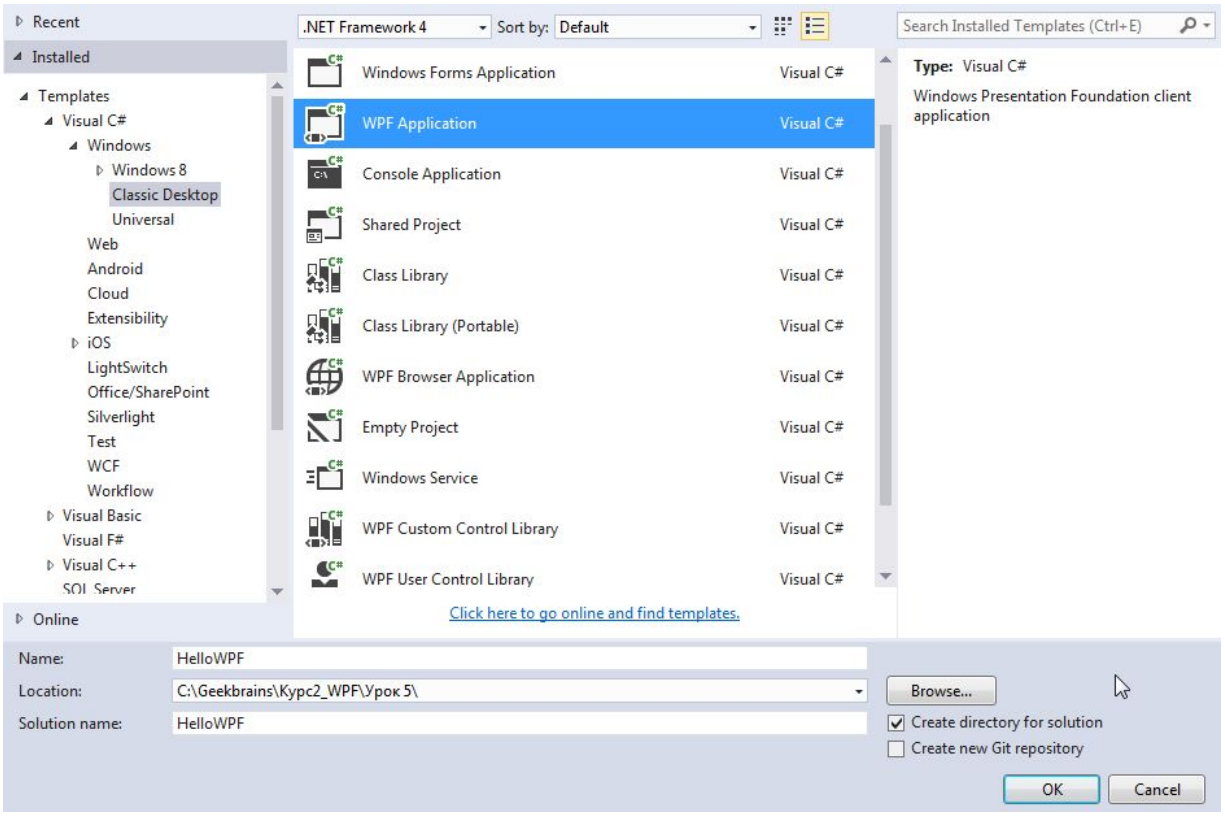
Достоинства WinForms

- Накоплен огромный опыт по разработке приложений с использованием WinForms;
- Большое количество контролов от сторонних разработчиков, платных и бесплатных;
- Редактор для WinForms удобнее, чем в WPF.

Hello, WPF!

Классическим первым приложением при изучении программирования является «Hello, World». Цель такого рода приложений – вывести на экран фрагмент текста и продемонстрировать, насколько легко начать использовать новую технологию или язык программирования.

Создадим с помощью Visual Studio проект **HelloWPF**, используя шаблон **WPF Application**.



В результате в папке проекта будет создано несколько файлов. Сейчас нас интересует файл **MainWindow.xaml**. По умолчанию это главная форма нашего приложения. Она отображается на экране после старта приложения.

Для данной формы код **XAML** выглядит следующим образом:

MainWindow.xaml

```
<Window x:Class="HelloWPF.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:HelloWPF"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>
```

Добавим контрол **TextBox** в Grid-панель формы. Установим значение свойства **Text** у элемента **TextBlock** равным «**Hello, WPF!**». Размер шрифта и выравнивание текста установите на свое усмотрение.

MainWindow.xaml

```
<Window x:Class="HelloWPF.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:HelloWPF"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid Name="MainGrid">
        <TextBlock x:Name="textBlock"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center"
                   Text=" Hello, WPF!" FontSize="36"/>
    </Grid>
</Window>
```

Запустим приложение:

Работа с App.xaml

В **WPF** приложение проходит через простой жизненный цикл. Вскоре после запуска приложения создается объект **Application**. Во время его выполнения возникают различные события, которые можно отслеживать. Когда объект приложения освобождается, оно завершается.

Файл **App.xaml** является тем объектом, в котором описывается запуск приложения. **VisualStudio** создает данный файл каждый раз при создании WPF-приложения, вместе с файлом отделенного кода **App.xaml.cs**.

Файл **App.xaml.cs** позволяет расширить класс **Application**, который является основным классом WPF-приложения. Платформа .Net использует **Application** для определения параметров старта приложения и запуска требуемого **Window** или **Page**. Кроме того, класс **Application** применяется для регистрации обработчиков событий, возникающих при старте приложения.

Наиболее часто файл **App.xaml** используется для описания глобальных ресурсов, задействованных в приложении, например, глобальных стилей.

Пример файла **App.xaml**:

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

Наибольший интерес в данном файле вызывает свойство **StartupUri**. С его помощью определяется, какие **Window** или **Page** будут отображены при старте приложения.

Если необходимо контролировать, когда и как отображается первое окно, удобнее удалить свойство **StartupUri** из файла **App**.xaml и определить процедуру отображения первого окна в файле отделенного кода.

Например, можно зарегистрировать обработчик события **Startup**, в котором создать первое окно вручную.

App.xaml

```
<Application x:Class="HelloWPF.App"

             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

             xmlns:local="clr-namespace:HelloWPF"

             Startup="Application_Startup">

            <!--StartupUri="MainWindow.xaml"-->

    <Application.Resources>

    </Application.Resources>

</Application>
```

App.xaml.cs

```
using System.Windows;

namespace HelloWPF

{

    /// <summary>

    /// Interaction logic for App.xaml

    /// </summary>

    public partial class App : Application

    {

        private void Application_Startup(object sender, StartupEventArgs e)

        {

            // Создаем первое окно

            MainWindow wnd = new MainWindow();

            // Определяем необходимые свойства окна

            wnd.Title = "Hello, WPF!";

            // Отображаем окно

            wnd.Show();

        }

    }

}
```

Запуск приложения с параметрами

Для использования параметров при запуске WPF-приложения достаточно в коде обратиться к параметру **StartupEventArgs**. Он содержит свойство **Args**, представляющее собой массив строк со значениями параметров приложения.

App.xaml

```
using System.Windows;

namespace HelloWPF

{

    /// <summary>

    /// Interaction logic for App.xaml

    /// </summary>

    public partial class App : Application

    {

        private void Application_Startup(object sender, StartupEventArgs e)

        {

            // Создаем первое окно

            MainWindow wnd = new MainWindow();

            if (e.Args.Length == 1)

                MessageBox.Show("Параметр: \n\n" + e.Args[0]);

            wnd.Show();

        }

    }

}
```

XAML

XAML – это язык разметки от Microsoft, основанный на XML и предназначенный для декларативного программирования приложений. В WinForms графический интерфейс создавался на том же языке программирования, что использовался для взаимодействия с графическим интерфейсом. Синтаксис C# или VB.Net поддерживался редактором форм (Visual Studio), но в случае с XAML Microsoft пошел другим путем.

Когда вы создаете **Window** или **Page**, они содержат файл **XAML** и файл отделенного кода (**codebehind file**). Файл **XAML** описывает интерфейс со всеми элементами, а файл отделенного кода обрабатывает все события и управляет XAML-контролами. При компиляции приложения в Visual Studio код в xaml-файлах также компилируется в бинарное представление кода xaml, которое называется **BAML**

(Binary Application Markup Language). И затем код baml встраивается в финальную сборку приложения – **exe** или **dll**-файл.

Теперь, как и в случае с HTML, вы можете легко создавать и изменять графический интерфейс своего приложения.

Функциональность **XAML** не ограничивается только графическими интерфейсами: данный язык также используется в технологиях **WCF** и **WorkflowFoundation**, где он никак не связан с графическим интерфейсом.

Основы XAML

Для создания контрола XAML достаточно написать название, заключив его в угловые скобки. Теги XAML должны быть «закрытыми», т.е. содержать закрывающий тэг, либо слэш в конце открывающего тэга. Например, **<Button></Button>** или **<Button />**.

Большинство контролов позволяют помещать содержимое между открывающим и закрывающим тэгом. Например, **Button** позволяет указывать отображаемый на кнопке текст следующим образом: **<Button> Кнопка </Button>**.

В отличие от HTML, XAML чувствителен к регистру, поскольку название контролов должны соответствовать типам в .Net-фреймворке. Аналогично, наименования атрибутов контролов чувствительны к регистру. Например, **<Button FontWeight="Bold" Content=" Кнопка" />**.

Кроме того, существует множество контролов, которые в качестве содержимого принимают не только текст, но и другие контролы.

Пример:

MainWindow.xaml

```
<Window x:Class="HelloWPF.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:HelloWPF"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button FontWeight="Bold" Width="140" Height="30">
            <WrapPanel Name="pnlMain">
                <TextBlock Foreground="Blue">Разно</TextBlock>
                <TextBlock Foreground="Red">цветная</TextBlock>
                <TextBlock>кнопка</TextBlock>
            </WrapPanel>
        </Button>
    </Grid>
</Window>
```

Свойство **Content** может содержать единственный дочерний элемент, поэтому используется контрол **WrapPanel** для объединения отдельных контролов **TextBlock**.

Аналогичная разметка может быть создана с использованием только кода C#, однако он будет проигрывать в наглядности.

```
Button btn = new Button();
 btn.FontWeight = FontWeights.Bold;
 WrapPanel pnl = new WrapPanel();
 TextBlock txt = new TextBlock();
 txt.Text = "Разно";
 txt.Foreground = Brushes.Blue;
 pnl.Children.Add(txt);
 txt = new TextBlock();
 txt.Text = "цветная";
 txt.Foreground = Brushes.Red;
 pnl.Children.Add(txt);
 txt = new TextBlock();
 txt.Text = "кнопка";
 pnl.Children.Add(txt);
 btn.Content = pnl;
 pnlMain.Children.Add(btn);
```

Пространства имен XAML

Для написания кода на языке C#, чтобы нам были доступны определенные классы, подключаем пространства имен с помощью директивы **using**.

Чтобы задействовать элементы в XAML, также подключаются пространства имен. **xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"** и **xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"** – это пространства имен, подключаемые в проект по умолчанию. А **xmlns** – специальный атрибут для определения пространства имен в XML.

Пространство имен **http://schemas.microsoft.com/winfx/2006/xaml/presentation** содержит описание и определение большинства элементов управления. Так как оно является пространством имен по умолчанию, то объявляется без всяких префиксов.

http://schemas.microsoft.com/winfx/2006/xaml – это пространство имен, которое определяет некоторые свойства XAML (например, **Name** или **Key**). Префикс **x** в определении **xmlns:x** означает, что те свойства элементов, которые заключены в этом пространстве имен, будут использоваться с префиксом **x** – например, **x:Name** или **x:Key**. Это же пространство имен используется уже в первой строчке **x:Class="HelloWPF.MainWindow"** – здесь создается новый класс **MainWindow** и соответствующий ему файл кода, куда будет прописываться логика для данного окна приложения.

Другие пространства имен:

- **xmlns:d="http://schemas.microsoft.com/expression/blend/2008"** – предоставляет поддержку атрибутов в режиме дизайнера. Это пространство имен преимущественно предназначено для другого инструмента по созданию дизайна на XAML – Microsoft Expression Blend.

- **xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"** – обеспечивает режим совместимости разметок XAML. В определении объекта Window (ниже) можно найти его применение: **mc:Ignorable="d"**. Это выражение позволяет игнорировать парсерам XAML во время выполнения приложения дизайнерские атрибуты из пространства имен с префиксом **d**, то есть из "**http://schemas.microsoft.com/expression/blend/2008«**.

- **xmlns:local="clr-namespace:HelloWPF"** – пространство имен текущего проекта. Через префикс **local** можно получить в XAML различные объекты, которые определены в проекте.

Пространства имен не эквивалентны тем, которые подключаются при помощи директивы **using** в C#. Так, например, **http://schemas.microsoft.com/winfx/2006/xaml/presentation** подключает в проект следующие пространства имен:

- System.Windows
- System.Windows.Automation
- System.Windows.Controls
- System.Windows.Controls.Primitives
- System.Windows.Data
- System.Windows.Documents
- System.Windows.Forms.Integration
- System.Windows.Ink
- System.Windows.Input
- System.Windows.Media
- System.Windows.Media.Animation
- System.Windows.Media.Effects
- System.Windows.Media.Imaging
- System.Windows.Media.Media3D
- System.Windows.Media.TextFormatting
- System.Windows.Navigation
- System.Windows.Shapes
- System.Windows.Shell

События в XAML

WPF, как и большинство современных фреймворков, предназначенных для реализации пользовательского интерфейса, управляется событиями (**event driven**).

Все контролы, включая **Window**, который является потомком **Control**, предоставляют множество событий, на которые может быть подписано приложение.

Существует огромное количество типов событий. Большинство контролов содержит события **KeyDown**, **KeyUp**, **MouseDown**, **MouseEnter**, **MouseLeave**, **MouseUp** и т.п.

Язык **XAML** позволяет конструировать пользовательский интерфейс, но для создания функционирующего приложения необходим способ подключения обработчиков событий. В **XAML** это легко сделать с помощью атрибута **Class**:

```
<Window x:Class="HelloWPF.MainWindow"
```

Атрибут **Class** сообщает анализатору **XAML**, чтобы он сгенерировал новый класс с указанным именем. Он наследуется от класса, именованного элементом XML. Другими словами, этот пример создает новый класс по имени **MainWindow**, который наследуется от базового класса **Window**.

Класс **MainWindow** генерируется автоматически во время компиляции. Разработчик приложения может предоставить часть класса **MainWindow**, которая будет объединена с автоматически сгенерированной частью этого класса. Указанная часть – блестящий контейнер для кода обработки событий.

Этот «фокус» возможен благодаря средству C#, известному как частичные классы (**partial class**). Они позволяют разделить класс на две или более отдельных части во время разработки, которые соединяются в скомпилированной сборке. Частичные классы могут применяться во многих сценариях управления кодом, но более всего удобны, когда код должен объединяться с файлом, сгенерированным визуальным конструктором.

Рассмотрим пример, в котором реализован обработчик **MouseUp** в контроле **Grid**.

В коде **XAML** необходимо указать имя метода обработчика события **MouseUp**. Таким образом, мы подписываемся на указанное событие.

MainWindow.xaml

```
<Window x:Class="HelloWPF.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:HelloWPF"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid Name="MainGrid" MouseUp="Grid_MouseUp" Background="LightBlue">
    </Grid>
</Window>
```

Тело метода обработчика события указывается в файле отделенного кода.

MainWindow.xaml.cs

```
using System.Windows;

namespace HelloWPF

{

    /// <summary>

    /// Interaction logic for MainWindow.xaml

    /// </summary>

    public partial class MainWindow : Window

    {

        public MainWindow()

        {

            InitializeComponent();

        }

                    private          void          Grid_MouseUp(object          sender,
System.Windows.Input.MouseButtonEventArgs e)

        {

            MessageBox.Show("Координаты " + e.GetPosition(this).ToString());

        }

    }

}
```

Событие **MouseUp** использует делегат **MouseButtonEventHandler**, который имеет два параметра: **Sender** – контрол, вызвавший событие; **MouseButtonEventArgs** – объект, содержащий информацию о координатах курсора мыши.

Регистрация обработчика событий в файле отделенного кода

В большинстве случаев удобно использовать описанный выше способ регистрации обработчиков событий. Но иногда может потребоваться зарегистрировать обработчик событий в файле отделенного кода.

MainWindow.xaml.cs

```
using System.Windows;
namespace HelloWPF
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            MainGrid.MouseUp += MainGrid_MouseUp;
        }
       private void MainGrid_MouseUp(object sender,
       System.Windows.Input.MouseButtonEventArgs e)
        {
            MessageBox.Show("Координаты " + e.GetPosition(this).ToString());
        }
    }
}
```

Ресурсы

WPF предоставляет разработчикам возможность сохранять данные как ресурсы для отдельного контрола, всего окна или приложения в целом. Здесь речь идет о логических ресурсах, которые могут представлять различные объекты – элементы управления, кисти, коллекции объектов и т.д. Логические ресурсы можно установить в коде XAML или в коде C# с помощью свойства **Resources**. Оно определено в базовом классе **FrameworkElement**, поэтому его имеет большинство классов WPF.

В чем смысл использования ресурсов? Они повышают эффективность: можно определить один раз какой-либо ресурс и затем многократно использовать его на различных участках приложения. Улучшается поддержка: если возникнет необходимость изменить ресурс, достаточно это сделать в одном месте – и изменения произойдут глобально в приложении.

Свойство **Resources** представляет объект **ResourceDictionary** или словарь ресурсов, где каждый хранящийся ресурс имеет определенный ключ.

Пример использования ресурсов

Поскольку в качестве ресурса используется строковая переменная, необходимо добавить соответствующий **namespace** в **MainWindow.xaml**: **xmlns:sys="clr-namespace:System;assembly=mscorlib"**.

Затем необходимо добавить непосредственно сам ресурс, указав значение идентификатора ресурса, **x:Key**.

```
    <Window.Resources>
        <sys:String x:Key="strHelloWPF">Hello, WPF!</sys:String>
    </Window.Resources>
```

После этого становится возможным в различных элементах окна **MainWindow** ссылаться на данный ресурс, используя значение идентификатора ресурса, **strHelloWPF**.

```
<Window x:Class="HelloWPF.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:sys="clr-namespace:System;assembly=mscorlib"
        xmlns:local="clr-namespace:HelloWPF"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <sys:String x:Key="strHelloWPF">Hello, WPF!</sys:String>
    </Window.Resources>
    <StackPanel Margin="10">
        <TextBlock Text="{StaticResource strHelloWPF}" FontSize="56" />
         <TextBlock>Еще раз "<TextBlock Text="{StaticResource strHelloWPF}" />", но
из ресурсов!</TextBlock>
    </StackPanel>
</Window>
```

StaticResource vs DynamicResource

В предыдущем примере использовались статические ресурсы. Но помимо них можно применять и динамические ресурсы.

Статические ресурсы устанавливается только один раз при загрузке XAML-файла. А динамические ресурсы могут меняться в течение работы программы. Они позволяют использовать данные, которые отсутствуют во время разработки приложения и формируются только при его выполнении.

Пример одновременного использования статических и динамических ресурсов:

MainWindow.xaml

```
<Window x:Class="WpfResources.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:WpfResources"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        mc:Ignorable="d"
        Height="160"
        Width="300"
        Title="Ресурсы"
        Background="{DynamicResource WindowBackgroundBrush}">
    <Window.Resources>
        <sys:String x:Key="ComboBoxTitle">Список:</sys:String>
        <x:Array x:Key="ComboBoxItems" Type="sys:String">
            <sys:String>Элемент #1</sys:String>
            <sys:String>Элемент #2</sys:String>
            <sys:String>Элемент #3</sys:String>
        </x:Array>
        <LinearGradientBrush x:Key="WindowBackgroundBrush">
            <GradientStop Offset="0" Color="Silver"/>
            <GradientStop Offset="1" Color="AliceBlue"/>
        </LinearGradientBrush>
    </Window.Resources>
    <StackPanel Margin="10">
        <Label Content="{StaticResource ComboBoxTitle}" />
        <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
    </StackPanel>
</Window>
```

Здесь статические ресурсы используются для определения заголовка и содержимого выпадающего списка, а также применяется динамический ресурс для задания фона диалогового окна.

Локальные ресурсы и ресурсы уровня приложения

В предыдущих примерах ресурсы определялись на уровне диалогового окна, т.е. были доступны для каждого элемента данного окна. Если некоторые ресурсы требуются только для конкретного контрола, возможно добавление ресурсов только для ограниченного множества контролов. Изменим предыдущий пример, чтобы заголовок выпадающего списка был доступен только для дочерних элементов **StackPanel**:

MainWindow.xaml

```
<Window x:Class="WpfResources.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:WpfResources"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        mc:Ignorable="d"
        Height="160"
        Width="300"
        Title="Ресурсы"
        Background="{DynamicResource WindowBackgroundBrush}">
    <Window.Resources>
        <!--<sys:String x:Key="ComboBoxTitle">Список:</sys:String>-->
        <x:Array x:Key="ComboBoxItems" Type="sys:String">
            <sys:String>Элемент #1</sys:String>
            <sys:String>Элемент #2</sys:String>
            <sys:String>Элемент #3</sys:String>
        </x:Array>
        <LinearGradientBrush x:Key="WindowBackgroundBrush">
            <GradientStop Offset="0" Color="Silver"/>
            <GradientStop Offset="1" Color="AliceBlue"/>
        </LinearGradientBrush>
    </Window.Resources>
    <StackPanel Margin="10">
        <StackPanel.Resources>
            <sys:String x:Key="ComboBoxTitle">Список:</sys:String>
        </StackPanel.Resources>
        <Label Content="{StaticResource ComboBoxTitle}" />
        <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
    </StackPanel>
</Window>
```

Если есть необходимость доступа к определенным ресурсам из разных окон, необходимо сохранять ресурсы на уровне приложения, в файле **App.xaml**.

```
<Application x:Class="WpfResources.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfResources"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <sys:String x:Key="ComboBoxTitle">Список:</sys:String>
    </Application.Resources>
</Application>
```

Доступ к ресурсам из файла отделенного кода

Помимо доступа к ресурсам из файла XAML возможна работа с ресурсами из файла отделенного кода.

App.xaml

```
<Application x:Class="WpfResources.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfResources"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <sys:String x:Key="strApp">Hello, from Application!</sys:String>
    </Application.Resources>
</Application>
```

MainWindow.xaml

```
<Window x:Class="WpfResources.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:WpfResources"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        mc:Ignorable="d"
        Height="160"
        Width="300"
        Title="Ресурсы">
    <Window.Resources>
        <sys:String x:Key="strWindow">Hello, from Window!</sys:String>
    </Window.Resources>
    <DockPanel Margin="10" Name="pnlMain">
        <DockPanel.Resources>
            <sys:String x:Key="strPanel">Hello, from Panel!</sys:String>
        </DockPanel.Resources>
        <WrapPanel DockPanel.Dock="Top" HorizontalAlignment="Center" Margin="10">
            <Button Name="btnClickMe" Click="btnClickMe_Click">Click me!</Button>
        </WrapPanel>
        <ListBox Name="lbResult" />
    </DockPanel>
</Window>
```

MainWindow.xaml.cs

```
using System.Windows;
namespace WpfResources
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private void btnClickMe_Click(object sender, RoutedEventArgs e)
        {
            lbResult.Items.Add(pnlMain.FindResource("strPanel").ToString());
            lbResult.Items.Add(this.FindResource("strWindow").ToString());

lbResult.Items.Add(Application.Current.FindResource("strApp").ToString());
        }
    }
}
```

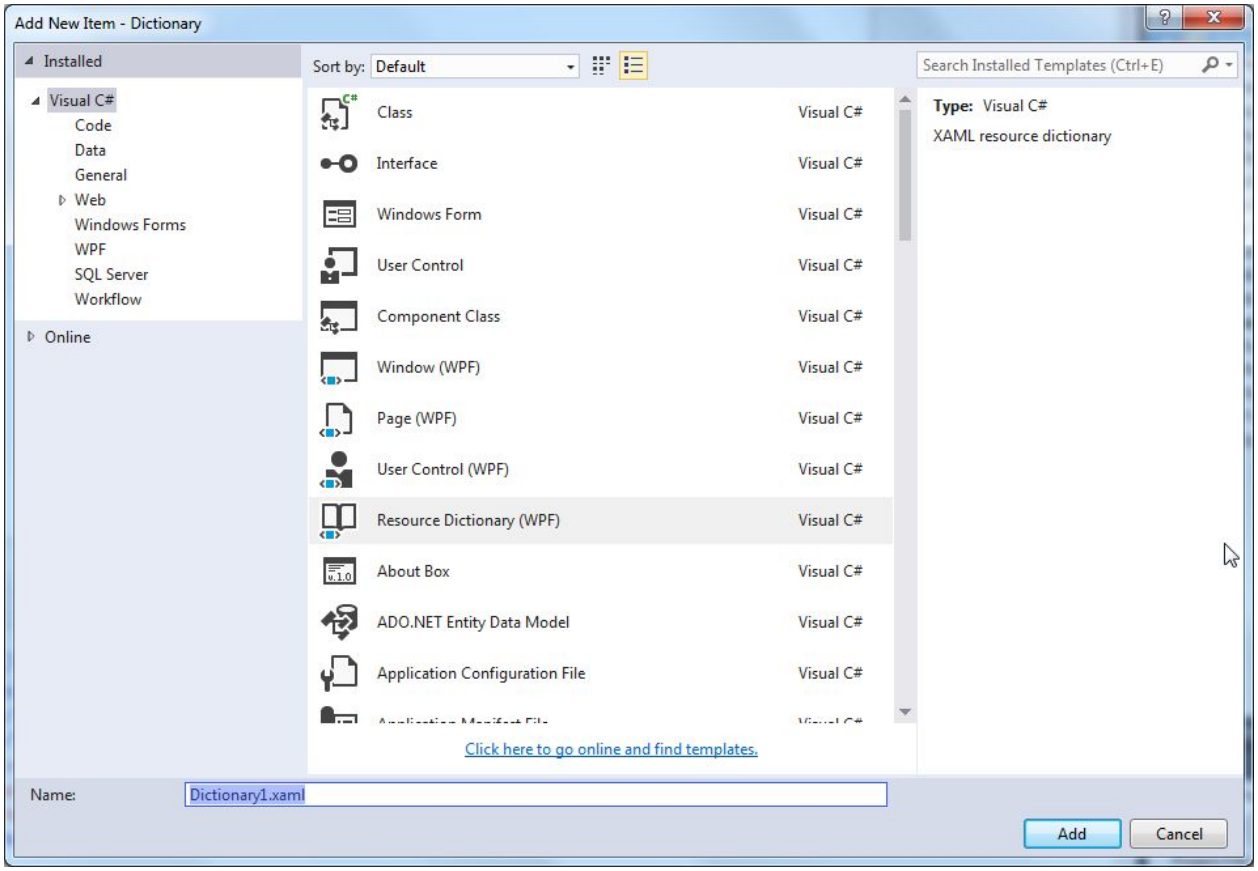
Для поиска ресурсов используется метод **FindResource()**. В примере выше поиск ресурсов производится сначала в панели, затем на уровне окна, а в конце – на уровне приложения. Если поиск не обнаруживает запрошенные ресурсы на одном из уровней, он продолжается на уровне выше, пока не обнаружит искомое значение. В принципе, поиск ресурсов в предыдущем примере можно выполнять всегда на уровне панели.

```
private void btnClickMe_Click(object sender, RoutedEventArgs e)
        {
            lbResult.Items.Add(pnlMain.FindResource("strPanel").ToString());
            lbResult.Items.Add(pnlMain.FindResource("strWindow").ToString());
            lbResult.Items.Add(pnlMain.FindResource("strApp").ToString());
        }
```

Словари ресурсов

Существует еще один способ определения ресурсов, который предполагает использование словаря ресурсов.

Добавим в проект словарь с названием **Dictionary1**:



В файл **Dictionary1.xaml** добавим градиентную заливку с **x:Key="Brush"**:

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:local="clr-namespace:Dictionary">
    <LinearGradientBrush x:Key="Brush">
        <GradientStopCollection>
            <GradientStop Color="White" Offset="0" />
            <GradientStop Color="Blue" Offset="1" />
        </GradientStopCollection>
    </LinearGradientBrush>
</ResourceDictionary>
```

В файл **App.xaml** добавим информацию о созданном словаре **Dictionary1**.

Элемент **ResourceDictionary.MergedDictionaries** представляет собой коллекцию словарей ресурсов, которые добавляются к ресурсам приложения.

```
<Application x:Class="Dictionary.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:Dictionary"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Dictionary1.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

Загрузку словаря ресурсов не обязательно производить через ресурсы приложения. У объекта **ResourceDictionary** имеется свойство **Source**, через которое можно связать ресурсы (например, **Window**) со словарем.

MainWindow.xaml

```
<Window x:Class="Dictionary.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:Dictionary"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <ResourceDictionary Source="Dictionary1.xaml" />
    </Window.Resources>
    <Grid>
                        <Button   Content="Кнопка"   MaxHeight="40"   MaxWidth="80"
Background="{StaticResource Brush}" />
    </Grid>
</Window>
```

Стили

Специалисты, знакомые с HTML и CSS, обнаружат, что XAML очень похож на HTML. С помощью тэгов описывается разметка интерфейса приложения. С помощью таких свойств, как Foreground, FontSize и т.п., возможно изменение внешнего вида отдельных контролов.

Аналогично CSS, в HTML в WPF существуют стили, которые позволяют устанавливать внешний вид для группы контролов или для определенного их типа. И как в CSS, предусмотрена возможность наследования стилей.

Пример использования стилей:

MainWindow.xaml

```
<Window x:Class="WpfStyle.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:WpfStyle"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel Margin="10">
        <StackPanel.Resources>
            <Style TargetType="TextBlock">
                <Setter Property="Foreground" Value="Black" />
                <Setter Property="FontSize" Value="20" />
            </Style>
        </StackPanel.Resources>
        <TextBlock>Заголовок 1</TextBlock>
        <TextBlock Foreground="Blue">Заголовок 2</TextBlock>
    </StackPanel>
</Window>
```

В ресурсах контрола **StackPanel** определен стиль, у которого значение свойства **TargetType** равно **TextBlock**. Это означает, что данный стиль применяется ко всем контролам **TextBlock** внутри **StackPanel**. В стиль добавлены два **Setter**-а, с помощью которых установлены свойства стилей – **ForeGround** и **FontSize**.

Для второго **TextBlock** свойство **ForeGround** установлено в **Blue**. Это демонстрирует возможность изменять значение свойств определенных контролов вне зависимости от используемых стилей.

Стили контролов

Можно применить стиль исключительно к одному контролу. При этом стиль определяется непосредственно внутри этого контрола.

MainWindow.xaml

```
<Window x:Class="WpfUsingStyle.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:WpfUsingStyle"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid Margin="10">
        <TextBlock Text="Style test">
            <TextBlock.Style>
                <Style>
                    <Setter Property="TextBlock.FontSize" Value="20" />
                </Style>
            </TextBlock.Style>
        </TextBlock>
    </Grid>
```

Стили дочерних контролов

Определив стили внутри контрола, можно применить их и для дочерних элементов.

MainWindow.xaml

```
<Window x:Class="WpfStyle.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:WpfStyle"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel Margin="10">
        <StackPanel.Resources>
            <Style TargetType="TextBlock">
                <Setter Property="Foreground" Value="Black" />
                <Setter Property="FontSize" Value="20" />
            </Style>
        </StackPanel.Resources>
        <TextBlock>Заголовок 1</TextBlock>
        <TextBlock Foreground="Blue">Заголовок 2</TextBlock>
    </StackPanel>
</Window>
```

Стили, определенные для всего диалогового окна

MainWindow.xaml

```
<Window x:Class="WpfStyle.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:WpfStyle"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Foreground" Value="Black" />
            <Setter Property="FontSize" Value="20" />
        </Style>
    </Window.Resources>
    <StackPanel Margin="10">
        <TextBlock>Заголовок 1</TextBlock>
        <TextBlock Foreground="Blue">Заголовок 2</TextBlock>
    </StackPanel>
</Window>
```

Стили, определенные для всего приложения

App.xaml

```
<Application x:Class="WpfStyle.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfStyle"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Foreground" Value="Black" />
            <Setter Property="FontSize" Value="20" />
        </Style>
    </Application.Resources>
</Application>
```

MainWindow.xaml

```
<Window x:Class="WpfStyle.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:WpfStyle"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel Margin="10">
        <TextBlock>Заголовок 1</TextBlock>
        <TextBlock Foreground="Blue">Заголовок 2</TextBlock>
    </StackPanel>
</Window>
```

Явное применение стилей

Во всех приведенных выше примерах тот или иной стиль применялся ко всем контролам, тип которых совпадает со значением **TargetType** в объявлении стиля. Но если указать для стиля определенное значение свойства **x:Key**, можно задать стиль только для конкретного контрола. И это возможно, несмотря на указание **TargetType**. Пример:

MainWindow.xaml

```
<Window x:Class="WpfStyle.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:WpfStyle"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="TextBlock" x:Key="HeaderStyle">
            <Setter Property="Foreground" Value="Black" />
            <Setter Property="FontSize" Value="20" />
        </Style>
    </Window.Resources>
    <StackPanel Margin="10">
        <TextBlock Style="{StaticResource HeaderStyle}">Заголовок 1</TextBlock>
        <TextBlock>Заголовок 2</TextBlock>
    </StackPanel>
</Window>
```

Обзор элементов управления и их свойств

TextBlock

TextBlock позволяет отображать фрагмент текста на экране почти так же, как и **Label**, но проделывает это проще и с меньшей затратой ресурсов. Основные отличия **TextBlock** и **Label** состоят в следующем:

- **Label** может отображать одну строку текста, которая может включать другие контролы;
- **TextBlock** может отображать несколько строк только текста.

TextBlock подходит для работы с крупными фрагментами многострочного текста. Но нужно иметь в виду, что если текст превышает вместимость **TextBlock**, **WPF** отображает только ту часть, что поместилась в **TextBox**.

Чтобы обойти это ограничение, существуют следующие приемы:

- LineBreak;
- TextTrimming;
- TextWrapping.

MainWindow.xaml

```
<Window x:Class="TextBlock.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:TextBlock"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <TextBlock Margin="10" Foreground="Red">
                                    Меня зовут Бакаре Тунде, я брат первого нигерийского
космонавта,<LineBreak />
                        майора ВВС Нигерии Абака Тунде.
        </TextBlock>
        <TextBlock Margin="10" TextTrimming="CharacterEllipsis" Foreground="Green">
                         Мой брат стал первым африканским космонавтом, который отправился
с секретной миссией на советскую станцию «Салют-6» в далеком 1979 году.
        </TextBlock>
        <TextBlock Margin="10" TextWrapping="Wrap" Foreground="Blue">
                        Все русские члены команды сумели вернуться на землю, однако моему
брату не хватило в корабле места.
        </TextBlock>
    </StackPanel>
</Window>
```

Label

Для **Label**, в отличие от **TextBlock**, можно указать набор горячих клавиш, с помощью которых устанавливается фокус на определенный контрол. Набор горячих клавиш формируется из «**Alt**» и буквы после символа «**_**».

MainWindow.xaml

```
<Window x:Class="Label.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:Label"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel Margin="10">
        <Label Content="_Name:" Target="{Binding ElementName=txtName}" />
        <TextBox Name="txtName" />
        <Label Content="_Mail:" Target="{Binding ElementName=txtMail}" />
        <TextBox Name="txtMail" />
    </StackPanel>
</Window>
```

TextBox

Элемент **TextBox** представляет поле для ввода текстовой информации (допустимое количество символов можно ограничить с помощью свойства **MaxLength**). Однако часто бывает нужно многострочное текстовое окно для работы с большим объемом содержимого. Для этого нужно присвоить свойству **TextWrapping** значение **Wrap** или **WrapWithOverflow**. При значении **Wrap** текст всегда разрывается на краю элемента управления, даже если придется разбить слишком длинные слова.

MainWindow.xaml

```
<Window x:Class="TextBox.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:TextBox"
        mc:Ignorable="d"
        Title="MainWindow" Height="200" Width="200">
    <Grid Margin="10">
        <TextBox AcceptsReturn="True" TextWrapping="Wrap" />
    </Grid>
</Window>
```

AcceptReturn разрешает использование перевода строки (**Enter**) при вводе данных.

Button

В **WPF** существует три типа кнопок: **Button**, **CheckBox** и **RadioButton**. Все эти кнопки являются наследниками класса **ButtonBase**.

Класс **ButtonBase** определяет событие **Click** и добавляет поддержку команд, которые позволяют подключать кнопки к высокоуровневым задачам приложений. Класс **ButtonBase** добавляет свойство **ClickMode**, которое определяет, когда кнопка генерирует событие **Click** в ответ на действия мыши. По умолчанию используется значение **ClickMode.Release**, которое означает, что событие Click будет сгенерировано после нажатия и последующего отпускания кнопки мыши. Но можно сделать и так, чтобы событие **Click** возникало сразу при нажатии кнопки мыши (**ClickMode.Press**) или даже когда указатель мыши будет наведен на кнопку и задержится над ней (**ClickMode.Hover**).

Все кнопки поддерживают горячие клавиши. Для обозначения горячей клавиши служит символ подчеркивания. Когда пользователь нажмет клавишу **<Alt>** и клавишу доступа, возникнет событие **Click** данной кнопки.

Класс **Button** представляет собой кнопку **Windows**. Он добавляет всего два доступных для записи свойства: **IsCancel** и **IsDefault**.

Если свойство **IsCancel** имеет значение **true**, эта кнопка будет работать в окне как кнопка отмены. Если нажать клавишу **<Esc>**, когда фокус находится в текущем окне, эта кнопка сработает.

Если свойство **IsDefault** имеет значение **true**, эта кнопка считается кнопкой по умолчанию (кнопкой принятия).

CheckBox

Элемент **CheckBox** представляет собой обычный флажок. Он может принимать три состояния: **Checked**, **Unchecked** и **Intermediate**.

Ключевые события флажка – **Checked** (генерируется при установке флажка в отмеченное состояние), **Unchecked** (генерируется при снятии отметки с флажка) и **Indeterminate** (флажок переведен в неопределенное состояние).

Атрибут **IsThreeState="True"** указывает, что флажок может находиться в трех состояниях.

MainWindow.xaml

```
<Window x:Class="CheckBox.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:CheckBox"
        mc:Ignorable="d"
        Title="MainWindow" Height="150" Width="150">
    <StackPanel x:Name="stackPanel">
              <CheckBox  x:Name="checkBox1"  IsThreeState="True"  IsChecked="False"
Height="20" Content="Не отмечено" />
               <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"/>
        <CheckBox x:Name="checkBox" IsChecked="False" Height="20" Content="Флажок"
                  IsThreeState="True"
                  Unchecked="checkBox_Unchecked"
                  Indeterminate="checkBox_Indeterminate"
                  Checked="checkBox_Checked" />
    </StackPanel>
</Window>
```

MainWindow.xaml.cs

```
using System.Windows;
namespace CheckBox
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        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");
        }
    }
}
```

RadioButton

Элемент управления **RadioButton** представляет собой 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.

Чтобы включить элемент в определенную группу, используется свойство **GroupName**.

Свойство **IsChecked** применяется, чтобы установить значение для **RadioButton**, а также чтобы определить, установлен ли переключатель в файле отделенного кода.

MainWindow.xaml

```
<Window x:Class="RadioButton.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:RadioButton"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel Margin="10">
        <Label FontWeight="Bold">Вы знаете C#?</Label>
        <RadioButton GroupName="ready">Да</RadioButton>
        <RadioButton GroupName="ready">Нет</RadioButton>
        <RadioButton GroupName="ready" IsChecked="True">Не уверен</RadioButton>
    </StackPanel>
</Window>
```

ItemsControl

Элементы управления списками представлены в **WPF** довольно широко. Все они являются производными от класса **ItemsControl** (наследника класса **Control**). Все они содержат коллекцию элементов. Элементы могут быть напрямую добавлены в коллекцию, и возможна также привязка некоторого массива данных к коллекции.

MainWindow.xaml

```
<Window x:Class="ItemsControl.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:system="clr-namespace:System;assembly=mscorlib"
        xmlns:local="clr-namespace:ItemsControl"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid Margin="10">
        <ItemsControl>
            <system:String>Строка #1</system:String>
            <system:String>Строка #2</system:String>
            <system:String>Строка #3</system:String>
            <system:String>Строка #4</system:String>
            <system:String>Строка #5</system:String>
        </ItemsControl>
    </Grid>
</Window>
```

По умолчанию **ItemsControl** не имеет вертикальных и горизонтальных скроллбаров, но можно дополнить ими этот класс. Достаточно поместить элемент **ItemsControl** внутрь элемента **ScrollViewer**.

MainWindow.xaml

```
<Window x:Class="ItemsControl.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:system="clr-namespace:System;assembly=mscorlib"

        xmlns:local="clr-namespace:ItemsControl"

        mc:Ignorable="d"

        Title="MainWindow" Height="350" Width="525">

    <Grid Margin="10">

                                <ScrollViewer    VerticalScrollBarVisibility="Auto"
HorizontalScrollBarVisibility="Auto">

            <ItemsControl>

                <system:String>ItemsControl Item #1</system:String>

                <system:String>ItemsControl Item #2</system:String>

                <system:String>ItemsControl Item #3</system:String>

                <system:String>ItemsControl Item #4</system:String>

                <system:String>ItemsControl Item #5</system:String>

            </ItemsControl>

        </ScrollViewer>

    </Grid>

</Window>
```

ListBox

Класс **ListBox** представляет собой список переменной длины, который позволяет пользователю выбрать один из элементов. Класс содержит коллекцию элементов **ListBoxItem**, которые являются элементами управления содержимым.

ListBox поддерживает множественный выбор. Для этого нужно установить свойство **SelectionMode="Multiple"** или **SelectionMode="Extended"**. В последнем случае, чтобы выделить несколько элементов, необходимо держать нажатой клавишу **Ctrl** или **Shift**. По умолчанию

SelectionMode="Single", то есть допускается только единственное выделение. При множественном выборе для получения всех выделенных элементов вместо свойства **SelectedItem** используется коллекция **SelectedItems**.

Объект **ListBox** хранит все вложенные объекты в своей коллекции **Items**.

Объект **ListBox** может хранить не только объекты **ListBoxItem**, но и любые произвольные элементы, так как класс **ListBoxItem** является наследником класса **ContentControl**, который позволяет хранить фрагменты вложенного содержимого. Если такой фрагмент является классом, порожденным от **UIElement**, он будет отображен в элементе **ListBox**. Если же это другой тип объекта, **ListBox** вызовет метод **ToString()** и выведет полученный текст.

MainWindow.xaml

```
<Window x:Class="ListBox.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:ListBox"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <ListBox Margin="10" Name="lbEmployee" SelectionMode="Single"
                                    SelectionChanged="lbEmployee_SelectionChanged"
VerticalAlignment="Top" Height="250"></ListBox>
            <Button Margin="10" Height="25" Width="70" VerticalAlignment="Bottom"
Click="Button_Click" >Добавить</Button>
    </Grid>
```

MainWindow.xaml.cs

```
using System.Collections.ObjectModel;
using System.Windows;
namespace ListBox
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
                                ObservableCollection<Employee>    items    =    new
ObservableCollection<Employee>();
        public MainWindow()
        {
            InitializeComponent();
            FillList();
        }
        void FillList()
```

```
        {
            items.Add(new Employee() { Id = 1, Name = "Vasya", Age = 22, Salary = 3000 });
            items.Add(new Employee() { Id = 2, Name = "Petya", Age = 25, Salary = 6000 });
            items.Add(new Employee() { Id = 3, Name = "Kolya", Age = 23, Salary = 8000 });
            lbEmployee.ItemsSource = items;
        }

        private void lbEmployee_Selected(object sender, RoutedEventArgs e)
        {
            MessageBox.Show(e.Source.ToString());
        }

        private void lbEmployee_SelectionChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e)
        {
            MessageBox.Show(e.AddedItems[0].ToString());
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            items.Add(new Employee() { Id = 1, Name = "Sergey", Age = 26, Salary = 7000 });
        }
    }

    public class Employee
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
        public double Salary { get; set; }
        public override string ToString()
        {
            return $"{Id}\t{Name}\t{Age}\t{Salary}";
        }
    }
}
```

ComboBox

Элемент **ComboBox** хранит коллекцию объектов **ComboBoxItem**. Как и **ListBoxItem**, **ComboBoxItem** является элементом управления содержимым, который может хранить любой вложенный элемент.

Отличие классов **ComboBox** и **ListBox** состоит в способе их отображения в окне. Элемент **ComboBox** использует раскрывающийся список – значит, за один раз можно выбрать только один элемент.

Если нужно сделать так, чтобы пользователь мог выбрать элемент в **ComboBox**, введя текст в текстовом поле, необходимо присвоить свойству **IsEditable** значение **true**. Это свойство работает в комбинации со свойством **IsReadOnly**: оно указывает, что поле ввода доступно только для чтения. По умолчанию имеет значение **False**, поэтому если **IsEditable="True"**, то мы можем вводить туда произвольный текст. Еще одно свойство – **StaysOpenOnEdit** – при установке в **True** позволяет сделать список раскрытым на время ввода значений в поле ввода.

MainWindow.xaml

```
<Window x:Class="ComboBoxWPF.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:ComboBoxWPF"

        mc:Ignorable="d"

        Title="MainWindow" Height="350" Width="525">

    <Grid>

        <ComboBox x:Name="comboBox" HorizontalAlignment="Center"
VerticalAlignment="Center" Width="200"
SelectionChanged="comboBox_SelectionChanged">

            <ComboBoxItem>Элемент #1</ComboBoxItem>

            <ComboBoxItem>Элемент #2</ComboBoxItem>

            <ComboBoxItem>Элемент #3</ComboBoxItem>

        </ComboBox>

    </Grid>

</Window>
```

MainWindow.xaml.cs

```
using System.Windows;
using System.Windows.Controls;
namespace ComboBoxWPF
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private void comboBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
        {
            ComboBox comboBox = (ComboBox)sender;
            ComboBoxItem selectedItem = (ComboBoxItem)comboBox.SelectedItem;
            MessageBox.Show(selectedItem.Content?.ToString());
        }
    }
}
```

Window

Чтобы рассмотреть взаимодействия окон, создадим проект **WindowRelation**. По умолчанию он уже содержит одно главное окно **MainWindow**.

MainWindow.xaml

```
<Window x:Class="WindowRelation.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:WindowRelation"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Width="100" Height="30" Content="Дочернее окно"
Click="Button_Click" />
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System.Windows;
namespace WindowRelation
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            ChildWindow childWindow = new ChildWindow();
            childWindow.ViewModel = "ViewModel";
            childWindow.Show();
            childWindow.ShowViewModel();
        }
    }
}
```

Добавим еще одно окно. Для этого в окне добавления нового элемента надо выбрать тип **"Window (WPF)"**.

Назовем новое окно **ChildWindow**. Используя ссылку на окно, мы можем взаимодействовать с ним – передавать ему данные из главной формы или вызывать его методы. Например, у окна **ChildWindow** устанавливается свойство **ViewModel** и вызывается его метод.

ChildWindow.xaml

```
<Window x:Class="WindowRelation.ChildWindow"
        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:WindowRelation"
        mc:Ignorable="d"
        Title="ChildWindow" Height="300" Width="300">
    <Grid>
        <TextBlock x:Name="textBlock" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="TextBlock" VerticalAlignment="Center"/>
    </Grid>
</Window>
```

ChildWindow.xaml.cs

```
using System.Windows;
namespace WindowRelation
{
    /// <summary>
    /// Interaction logic for ChildWindow.xaml
    /// </summary>
    public partial class ChildWindow : Window
    {
        public ChildWindow()
        {
            InitializeComponent();
        }
        public string ViewModel { get; set; }
        public void ShowViewModel()
        {
            textBlock.Text = ViewModel;
        }
    }
}
```

Стоит отметить, что после открытия эти окна существуют независимо друг от друга. Можно закрыть главное окно **MainWindow**, а второе окно (**ChildWindow**) продолжит работу. Но такое поведение несложно изменить.

У всех окон есть свойство **Owner**, которое указывает на главное окно, владеющее текущим. Добавим в обработчик **Button_Click** в главном окне следующий текст:

```
// Теперь MainWindow главное окно для childWindow
childWindow.Owner = this;
```

Теперь мы можем обращаться из **ChildWindow** к его владельцу и менять его свойства:

```
this.Owner.Background = new SolidColorBrush(Colors.Crimson);
```

Одновременно с этим все дочерние окна доступны в родительском окне через свойство **OwnedWindows**:

```
private void Button_Click(object sender, RoutedEventArgs e)
        {
            ChildWindow childWindow = new ChildWindow();
            // Теперь MainWindow главное окно для childWindow
            childWindow.Owner = this;
            childWindow.ViewModel = "ViewModel";
            childWindow.Show();
            childWindow.ShowViewModel();
            foreach (Window window in this.OwnedWindows)
            {
                window.Background = new SolidColorBrush(Colors.Aquamarine);
                if (window is ChildWindow)
                    window.Title = "Новый заголовок";
            }
        }
```

Кроме этого, класс **App** (главный класс приложения) содержит свойство **Windows**, которое хранит информацию обо всех открытых окнах приложения. И в любом месте программы можно получить эту информацию:

```
foreach (Window window in App.Current.Windows)
            {
                window.Background = new SolidColorBrush(Colors.Aquamarine);

                // Если окно – объект TaskWindow
                if (window is ChildWindow)
                    window.Title = "Новый заголовок!";
            }
```

Практическое задание

Создать **WPF**-приложение для ведения списка сотрудников компании.

1. Создать сущности **Employee** и **Department** и заполнить списки сущностей начальными данными.
2. Для списка сотрудников и списка департаментов предусмотреть визуализацию (отображение). Это можно сделать, например, с использованием **ComboBox** или **ListView**.
3. Предусмотреть редактирование сотрудников и департаментов. Должна быть возможность изменить департамент у сотрудника. Список департаментов для выбора можно выводить в **ComboBox**, и все это можно выводить на дополнительной форме.
4. Предусмотреть возможность создания новых сотрудников и департаментов. Реализовать это либо на форме редактирования, либо сделать новую форму.

Дополнительные материалы

1. XAML Unleashed 1st Edition by Adam Nathan, December 2014.
2. WPF 4.5 Unleashed 1st Edition by Adam Nathan, July 2013.
3. Windows Presentation Foundation 4.5 Cookbook by Pavel Yosifovich, September 2012.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. WPF 4.5 Unleashed 1st Edition by Adam Nathan, July 2013.
2. Windows Presentation Foundation 4.5 Cookbook by Pavel Yosifovich, September 2012.
3. MSDN.