Стили, триггеры и темы

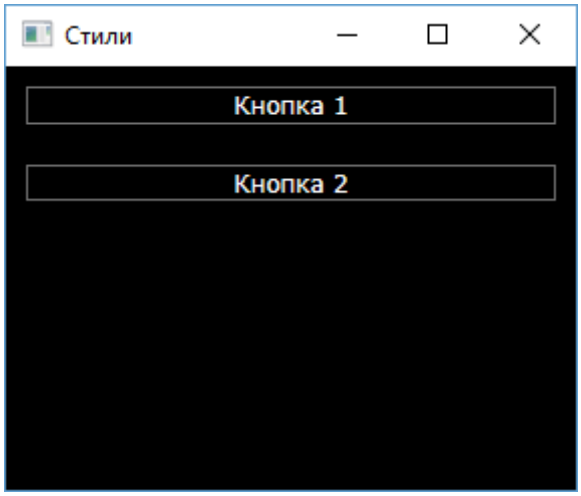
Стили

Стили позволяют определить набор некоторых свойств и их значений, которые потом могут применяться к элементам в xaml. Стили хранятся в ресурсах и отделяют значения свойств элементов от пользовательского интерфейса. Также стили могут задавать некоторые аспекты поведения элементов с помощью триггеров. Аналогом стилей могут служить каскадные таблицы стилей (CSS), которые применяются в коде html на веб-страницах.

Зачем нужны стили? Стили помогают создать стилевое единообразие для определенных элементов. Допустим, у нас есть следующий код xaml:

```
<StackPanel x:Name="buttonsStack" Background="Black" >
    <Button x:Name="button1" Margin="10" Content="Кнопка 1" FontFamily="Verdana" Foreground="Whi
    <Button x:Name="button2" Margin="10" Content="Кнопка 2" FontFamily="Verdana" Foreground="Whi
</StackPanel>
```

Здесь обе кнопки применяют ряд свойств с одними и теми же значениями:



Однако в данном случае мы вынуждены повторяться. Частично, проблему могло бы решить использование ресурсов:

```
<Window x:Class="StylesApp.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:StylesApp"
        mc:Ignorable="d"
        Title="Стили" Height="250" Width="300">
    <Window.Resources>
        <FontFamily x:Key="buttonFont">Verdana</FontFamily>
        <SolidColorBrush Color="White" x:Key="buttonFontColor" />
        <SolidColorBrush Color="Black" x:Key="buttonBackColor" />
        <Thickness x:Key="buttonMargin" Bottom="10" Left="10" Top="10" Right="10" />
    </Window.Resources>
    <StackPanel x:Name="buttonsStack" Background="Black" >
        <Button x:Name="button1" Content="Кнопка 1"
                Margin="{StaticResource buttonMargin}"
                FontFamily="{StaticResource buttonFont}"
                Foreground="{StaticResource buttonFontColor}"
                Background="{StaticResource buttonBackColor}" />
        <Button x:Name="button2" Content="Кнопка 2"
                Margin="{StaticResource buttonMargin}"
                FontFamily="{StaticResource buttonFont}"
                Foreground="{StaticResource buttonFontColor}"
                Background="{StaticResource buttonBackColor}"/>
    </StackPanel>
</Window>
```

Однако в реальности код раздувается, опть же приходится писать много повторяющейся информации. И в этом плане стили предлагают более элегантное решение:

```
<Window x:Class="StylesApp.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:StylesApp"
        mc:Ignorable="d"
        Title="Стили" Height="250" Width="300">
    <Window.Resources>
        <Style x:Key="BlackAndWhite">
            <Setter Property="Control.FontFamily" Value="Verdana" />
            <Setter Property="Control.Background" Value="Black" />
            <Setter Property="Control.Foreground" Value="White" />
            <Setter Property="Control.Margin" Value="10" />
        </Style>
    </Window.Resources>
    <StackPanel x:Name="buttonsStack" Background="Black" >
        <Button x:Name="button1" Content="Кнопка 1"
                Style="{StaticResource BlackAndWhite}" />
        <Button x:Name="button2" Content="Кнопка 2"
                Style="{StaticResource BlackAndWhite}"/>
    </StackPanel>
</Window>
```

Результат будет тот же, однако теперь мы избегаем ненужного повторения. Более того теперь мы можем управлять всеми нужными нам свойствами как единым целым - одним стилем.

Стиль создается как ресурс с помощью объекта **Style**, который представляет класс **System.Windows.Style**. И как любой другой ресурс, он обязательно должен иметь ключ. С помощью коллекции **Setters** определяется группа свойств, входящих в стиль. В нее входят объекты **Setter**, которые имеют следующие свойства:

- **Property**: указывает на свойство, к которому будет применяться данный сеттер. Имеет следующий синтаксис: Property="Тип_элемента.Свойство_элемента". Выше в качестве типа элемента использовался Control, как общий для всех элементов. Поэтому данный стиль мы могли бы применить и к Button, и к TextBlock, и к другим элементам. Однако мы можем и конкретизировать элемент, например, Button:

```
<Setter Property="Button.FontFamily" Value="Arial" />
```

- **Value**: устанавливает значение

Если значение свойства представляет сложный объект, то мы можем его вынести в отдельный элемент:

```
<Style x:Key="BlackAndWhite">
    <Setter Property="Control.Background">
        <Setter.Value>
            <LinearGradientBrush>
                <LinearGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0" />
                    <GradientStop Color="Black" Offset="1" />
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
    <Setter Property="Control.FontFamily" Value="Verdana" />
    <Setter Property="Control.Foreground" Value="White" />
    <Setter Property="Control.Margin" Value="10" />
</Style>
```

TargetType

Нам необязательно прописывать для всех кнопок стиль. Мы можем в самом определении стиля с помощью свойства **TargetType** задать тип элементов. В этом случае стиль будет автоматически применяться ко всем кнопкам в окне:

```
<Window x:Class="StylesApp.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:StylesApp"
        mc:Ignorable="d"
        Title="Стили" Height="250" Width="300">
    <Window.Resources>
        <Style TargetType="Button">
            <Setter Property="FontFamily" Value="Verdana" />
            <Setter Property="Background" Value="Black" />
            <Setter Property="Foreground" Value="White" />
            <Setter Property="Margin" Value="10" />
        </Style>
    </Window.Resources>
    <StackPanel x:Name="buttonsStack" Background="Black" >
        <Button x:Name="button1" Content="Кнопка 1"  />
        <Button x:Name="button2" Content="Кнопка 2" />
    </StackPanel>
</Window>
```

Причем в этом случае нам уже не надо указывать у стиля ключ x:Key несмотря на то, что это ресурс.

Также если используем свойство TargetType, то в значении атрибута Property уже необязательно указывать тип, то есть Property="Control.FontFamily". И в данном случае тип можно просто опустить: Property="FontFamily"

Если же необходимо, чтобы к какой-то кнопке не применялся автоматический стиль, то ее стилю присваивают значение null

```
<Button x:Name="button2" Content="Кнопка 2" Style="{x:Null}" />
```

Определение обработчиков событий с помощью стилей

Кроме коллекции Setters стиль может определить другую коллекцию - **EventSetters**, которая содержит объекты **EventSetter**. Эти объекты позволяют связать события элементов с обработчиками. Например, подключим все кнопки к одному обработчику события Click:

```
<Style TargetType="Button">
    <Setter Property="Button.Background" Value="Black" />
    <Setter Property="Button.Foreground" Value="White" />
    <Setter Property="Button.FontFamily" Value="Andy" />
    <EventSetter Event="Button.Click" Handler="Button_Click" />
</Style>
```

Соответственно в файле кода c# у нас должен быть определен обработчик Button_Click:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Button clickedButton = (Button)sender;
    MessageBox.Show(clickedButton.Content.ToString());
}
```

Наследование стилей и свойство BasedOn

У класса Style еще есть свойство **BasedOn**, с помощью которого можно наследовать и расширять существующие стили:

```
<Window.Resources>
    <Style x:Key="ButtonParentStyle">
        <Setter Property="Button.Background" Value="Black" />
        <Setter Property="Button.Foreground" Value="White" />
        <Setter Property="Button.FontFamily" Value="Andy" />
    </Style>
    <Style x:Key="ButtonChildStyle" BasedOn="{StaticResource ButtonParentStyle}">
        <Setter Property="Button.BorderBrush" Value="Red" />
        <Setter Property="Button.FontFamily" Value="Verdana" />
    </Style>
</Window.Resources>
```

Свойство BasedOn в качестве значения принимает существующий стиль, определяя его как статический ресурс. В итоге он объединяет весь функционал родительского стиля со своим собственным.

Если в дочернем стиле есть сеттеры для свойств, которые также используются в родительском стиле, как в данном случае сеттер для свойства Button.FontFamily, то дочерний стиль переопределяет родительский стиль.

Стили в C#

В C# стили представляют объект System.Windows.Style. Используя его, мы можем добавлять сеттеры и устанавливать стиль для нужных элементов:

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace StylesApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            Style buttonStyle = new Style();
            buttonStyle.Setters.Add(new Setter { Property = Control.FontFamilyProperty, Value =
            buttonStyle.Setters.Add(new Setter { Property = Control.MarginProperty, Value = new
            buttonStyle.Setters.Add(new Setter { Property = Control.BackgroundProperty, Value =
            buttonStyle.Setters.Add(new Setter { Property = Control.ForegroundProperty, Value =
            buttonStyle.Setters.Add(new EventSetter { Event= Button.ClickEvent, Handler= new Rou

            button1.Style = buttonStyle;
            button2.Style = buttonStyle;
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            Button clickedButton = (Button)sender;
            MessageBox.Show(clickedButton.Content.ToString());
        }
    }
}
```

При создании сеттера нам надо использовать свойство зависимостей, например, Property = Control.FontFamilyProperty. Причем для свойства Value у сеттера должен быть установлен объект именно того типа, которое хранится в этом свойстве зависимости. Так, свойство зависимости MarginProperty хранит объект типа Thickness, поэтому определение сеттера выглядит следующим образом:

```
new Setter { Property = Control.MarginProperty, Value = new Thickness(10) }
```

Триггеры

Тригерры позволяют декларативно задать некоторые действия, которые выполняются при изменении свойств стиля. Существует три вида триггеров:

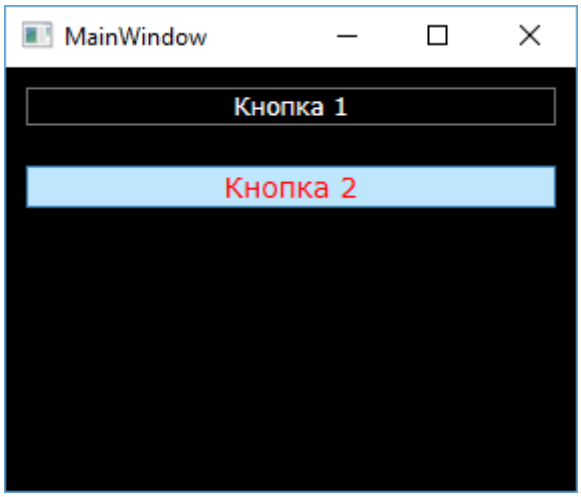
- **Триггеры свойств**: вызываются в ответ на изменения свойствами зависимостей своего значения
- **Триггеры** данных: вызываются в ответ на изменения значений любых свойств (они необязательно должны быть свойствами зависимостей)
- **Триггеры событий**: вызываются в ответ на генерацию событий
- **Мультитриггеры**: вызываются при выполнении ряда условий

Триггеры свойств

Простые триггеры свойств задаются с помощью объекта **Trigger**. Они следят за значением свойств и в случае их изменения с помощью объекта **Setter** устанавливают значение других свойств.

Например, в следующем примере по наведению на кнопку высота шрифта устанавливается в 14, а цвет шрифта становится красным:

```
<Window x:Class="TriggersApp.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:TriggersApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300">

    <Window.Resources>
        <Style TargetType="Button">
            <Style.Setters>
                <Setter Property="Button.Background" Value="Black" />
                <Setter Property="Button.Foreground" Value="White" />
                <Setter Property="Button.FontFamily" Value="Verdana" />
                <Setter Property="Button.Margin" Value="10" />
            </Style.Setters>
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="FontSize" Value="14" />
                    <Setter Property="Foreground" Value="Red" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <StackPanel Background="Black" >
        <Button x:Name="button1" Content="Кнопка 1"/>
        <Button x:Name="button2" Content="Кнопка 2" />
    </StackPanel>
</Window>
```



Здесь объект Trigger применяет два свойства: Property и Value. Свойство Property указывает на отслеживаемое свойство, а свойство Value указывает на значение, по достижении которого триггер начнет действовать. Например, в данном случае триггер отслеживает свойство IsMouseOver - если оно будет равно true, тогда сработает триггер.

Trigger имеет вложенную коллекцию Setters, в которой можно определить сеттеры, реализующие логику триггера.

MultiTrigger

При необходимости отслеживания не одного, а сразу нескольких свойств используют объект MultiTrigger. Он содержит коллекцию элементов Condition, каждый из которых, как и обычный триггер, определяет отслеживаемое свойство и его значение:

```
<Window x:Class="TriggersApp.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:TriggersApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300">

    <Window.Resources>
        <Style TargetType="Button">
            <Style.Setters>
                <Setter Property="Button.Background" Value="Black" />
                <Setter Property="Button.Foreground" Value="White" />
                <Setter Property="Button.FontFamily" Value="Verdana" />
                <Setter Property="Button.Margin" Value="10" />
            </Style.Setters>
            <Style.Triggers>
                <MultiTrigger>
                    <MultiTrigger.Conditions>
                        <Condition Property="IsMouseOver" Value="True" />
                        <Condition Property="IsPressed" Value="True" />
                    </MultiTrigger.Conditions>
                    <MultiTrigger.Setters>
                        <Setter Property="FontSize" Value="14" />
                        <Setter Property="Foreground" Value="Red" />
                    </MultiTrigger.Setters>
                </MultiTrigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <StackPanel Background="Black" >
        <Button x:Name="button1" Content="Кнопка 1"/>
        <Button x:Name="button2" Content="Кнопка 2" />
    </StackPanel>
</Window>
```

В данном случае если удовлетворены одновременно оба условия:

```
<Condition Property="IsMouseOver" Value="True" />
<Condition Property="IsPressed" Value="True" />
```

то срабатывают сеттеры мультитриггера:

```
<Setter Property="FontSize" Value="14" />
<Setter Property="Foreground" Value="Red" />
```

EventTrigger

Если простой триггер наблюдает за изменением свойства, то EventTrigger реагирует на определенные события совсем как обработчики событий. Правда, триггеры событий более ограничены в своих возможностях.

EventTrigger определяет анимацию и, если событие происходит, запускает ее на выполнение.

```
<Window x:Class="TriggersApp.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:TriggersApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300">

    <Window.Resources>
        <Style TargetType="Button">
            <Style.Setters>
                <Setter Property="Button.Background" Value="Black" />
                <Setter Property="Button.Foreground" Value="White" />
                <Setter Property="Button.FontFamily" Value="Verdana" />
                <Setter Property="Button.Margin" Value="10" />
            </Style.Setters>
            <Style.Triggers>
                <EventTrigger RoutedEvent="Click">
                    <EventTrigger.Actions>
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation Storyboard.TargetProperty="Width" Duration="0:0
                                <DoubleAnimation Storyboard.TargetProperty="Height" Duration="0:
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger.Actions>
                </EventTrigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <StackPanel Background="Black" >
        <Button x:Name="button1" Width="100" Height="30" Content="Кнопка 1"/>
    </StackPanel>
</Window>
```

В данном случае свойство **RoutedEvent** задает событие, на которое подписывается объект EventTrigger. Затем определяется свойство EventTrigger.Actions, которое задает анимацию, производимую в случае возникновения события.

Объект **BeginStoryboard** начинает анимацию, которая задается объектом **Storyboard**. Сама анимация определяется в объекте **DoubleAnimation**. Его свойство Storyboard.TargetProperty указывает на свойство элемента, изменяемое в процессе анимации, **Duration** задает время анимации, а свойство **To** - финальное значение свойства, на котором анимация заканчивается.

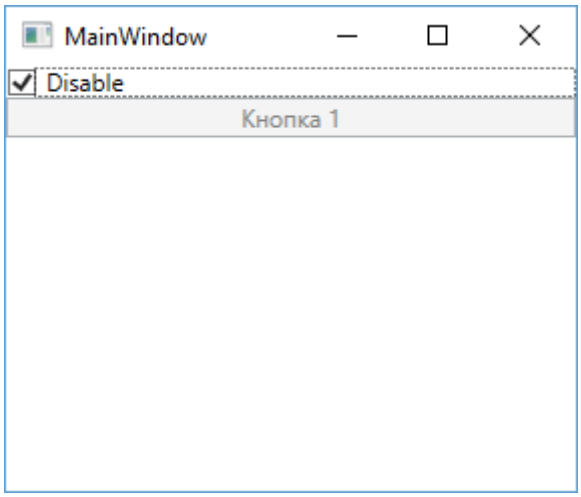
Суть данной анимации - значение свойства, указанного в To, сравнивается с текущим значением свойства и, если значение в To больше, то свойство увеличивает значение, иначе уменьшается.

Триггер данных DataTrigger

DataTrigger отслеживает изменение свойств, которые необязательно должны представлять свойства зависимостей. Для соединения с отслеживаемыми свойства триггеры данных используют выражения привязки:

```
<Window x:Class="TriggersApp.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:TriggersApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300">

    <Window.Resources>
        <Style TargetType="Button">
            <Style.Triggers>
                <DataTrigger Binding="{Binding ElementName=checkBox1, Path=IsChecked}"
                        Value="True">
                    <Setter Property="IsEnabled" Value="False"/>
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <StackPanel >
        <CheckBox x:Name="checkBox1" Content="Disable" />
        <Button x:Name="button1" Content="Кнопка 1" />
    </StackPanel>
</Window>
```

С помощью свойства Binding триггер данных устанавливает привязку к отслеживаемому свойству. Свойство **Value** задает значение отлеживаемого свойства, при котором сработает триггер. Так, в данном случае, если чекбокс будет отмечен, то сработает триггер, который установит у кнопки IsEnabled="False".



Темы

Стили позволяют задать стилевые особенности для определенного элемента или элементов одного типа. Но иногда возникает необходимость применить ко всем элементам какое-то общее стилевое единообразие. И в этом случае мы можем объединять стили элементов в темы. Например, все элементы могут выполнены в светлом стиле, или, наоборот, к ним может применяться так называемая "ночная тема". Более того может возникнуть необходимость не просто определить общую тему для всех элементов, но и иметь возможность динамически выбирать понравившуюся тему из списка тем. И в данной статье рассмотрим, как это сделать.

Пусть у нас есть окно приложения с некоторым набором элементов:

```
<Window x:Class="ThemesApp.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:ThemesApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300" Style="{DynamicResource WindowStyle}">
    <StackPanel>
        <ComboBox x:Name="styleBox" />
        <Button Content="Hello WPF" Style="{DynamicResource ButtonStyle}" />
        <TextBlock Text="Windows Presentation Foundation" Style="{DynamicResource TextBlockStyle
    </StackPanel>
</Window>
```

Для примера здесь определены кнопка, текстовый блок и выпадающий список, в котором позже будут выбираться темы.

К элементам окна уже применяются некоторые стили. Причем следует отметить, что стили указывают на динамические (не статические) ресурсы. Однако сами эти ресурсы еще не заданы. Поэтому зададим их.

Для этого добавим в проект новый файл словаря ресурсов, который назовем light.xaml, и определим в нем некоторый набор ресурсов:

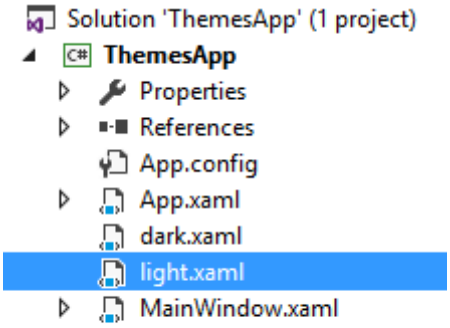
```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:local="clr-namespace:ThemesApp">
    <Style x:Key="TextBlockStyle" TargetType="TextBlock">
        <Setter Property="Background" Value="White" />
        <Setter Property="Foreground" Value="Gray" />
    </Style>
    <Style x:Key="WindowStyle" TargetType="Window">
        <Setter Property="Background" Value="White" />
    </Style>
    <Style x:Key="ButtonStyle" TargetType="Button">
        <Setter Property="Background" Value="White" />
        <Setter Property="Foreground" Value="Gray" />
        <Setter Property="BorderBrush" Value="Gray" />
    </Style>
</ResourceDictionary>
```

Здесь указаны все те стили, которые применяются элементами окна.

Но теперь также добавим еще один словарь ресурсов, который назовем dark.xaml и в котором определим следующий набор ресурсов:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:local="clr-namespace:ThemesApp">
    <Style x:Key="TextBlockStyle" TargetType="TextBlock">
        <Setter Property="Background" Value="Gray" />
        <Setter Property="Foreground" Value="White" />
    </Style>
    <Style x:Key="WindowStyle" TargetType="Window">
        <Setter Property="Background" Value="Gray" />
    </Style>
    <Style x:Key="ButtonStyle" TargetType="Button">
        <Setter Property="Background" Value="Gray" />
        <Setter Property="Foreground" Value="White" />
        <Setter Property="BorderBrush" Value="White" />
    </Style>
</ResourceDictionary>
```

Здесь определены те же самые стили, только их значения уже отличаются. То есть фактически мы создали две темы: для светлого и темного стилей.



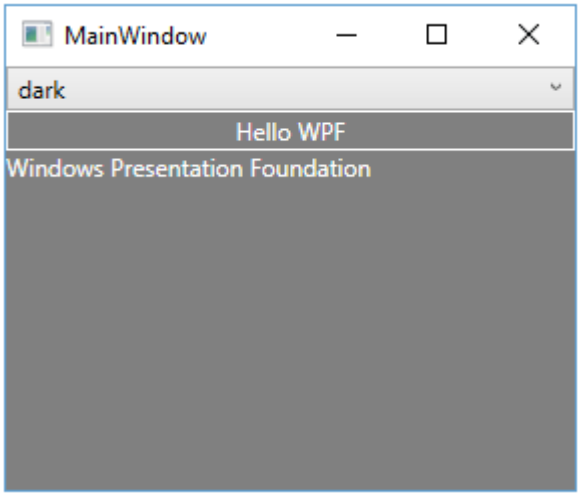
Теперь применим эти стили. Для этого изменим файл MainWindow.xaml.cs следующим образом:

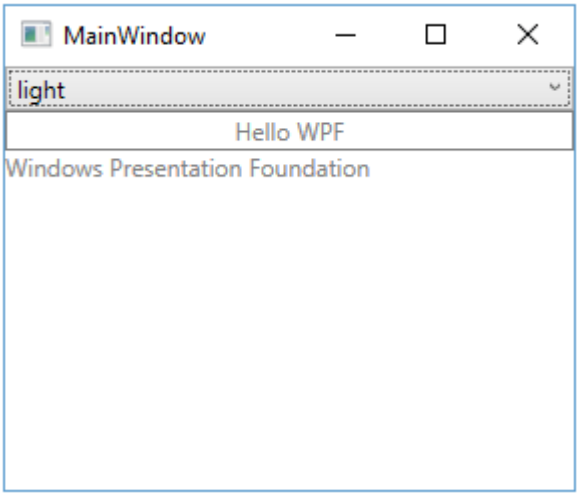
```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;

namespace ThemesApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            List<string> styles = new List<string> { "light", "dark" };
            styleBox.SelectionChanged += ThemeChange;
            styleBox.ItemsSource = styles;
            styleBox.SelectedItem = "dark";
        }

        private void ThemeChange(object sender, SelectionChangedEventArgs e)
        {
            string style = styleBox.SelectedItem as string;
            // определяем путь к файлу ресурсов
            var uri = new Uri(style + ".xaml", UriKind.Relative);
            // загружаем словарь ресурсов
            ResourceDictionary resourceDict = Application.LoadComponent(uri) as ResourceDictiona
            // очищаем коллекцию ресурсов приложения
            Application.Current.Resources.Clear();
            // добавляем загруженный словарь ресурсов
            Application.Current.Resources.MergedDictionaries.Add(resourceDict);
        }
    }
}
```

К элементу ComboBox цепляется обработчик ThemeChange, который срабатывает при выборе элемента в списке.

В методе ThemeChange получаем выделенный элемент, который представляет название темы. По нему загружаем локальный словарь ресурсов и добавляем этот словарь в коллекцию ресурсов приложения.

В итоге при выборе элемента в списке будет меняться применяемая к приложению тема:


MainWindow
dark
Hello WPF
Windows Presentation Foundation


MainWindow
light
Hello WPF
Windows Presentation Foundation