

РФ



**Разработка приложений
с использованием WPF**

Урок № 5

Трансформации

Содержание

Введение	5
1. Ресурсы	6
1.1. Перекрывающиеся ресурсы.....	11
1.2. Статические и динамические ресурсы.....	14
1.3. Объединенные словари ресурсов.....	18
2. Стили.....	22
2.1. Переопределение свойств стиля	26
2.2. Стили по умолчанию	28
2.3. Наследование стилей	30
3. Шаблоны элементов управления.....	33
3.1. Расширение разметки TemplateBinding.....	37
3.2. Триггеры	41
3.2.1. Триггеры свойств.....	42
3.2.2. Триггер данных	51
3.3. Примеры переопределения шаблонов	55

3.3.1. Шаблон элемента управления Button	56
3.3.2. Шаблон элемента управления TextBox.....	62
4. Домашнее задание	66
4.1. Задание 1	66
4.1.1. Правила игры	66
4.2. Задание 2.....	67

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

Введение

Работая с WPF и XAML с использованием только рассмотренных до этого момента инструментов может показаться, что любое приложение разработанное на их базе будет содержать уйму практически идентичной разметки для описания каждого отдельного элемента управления или фрагмента пользовательского интерфейса. А если представить, что реальные приложения, как правило, содержат внушительное количество настроек для каждого из них, то данный подход может и вовсе показаться неоправданным. Однако, это не так. В данном разделе будут рассмотрены инструменты, которые позволяют описывать сложные наборы настроек для элементов пользовательского интерфейса при этом избегая дублирования программного кода и разметки. Также будет рассмотрена одна из основных отличительных особенностей WPF от других технологий – возможность описывать собственный внешний вид для существующих элементов управления, таких как кнопки, поля для ввода текста и т.д.

1. Ресурсы

При описании XAML-разметки, программистам часто приходится сталкиваться с одной проблемой: описание одного и того же фрагмента разметки в нескольких местах. Например, нескольким элементам управления необходимо установить одну и ту же градиентную кисть для заливки заднего фона. Данная задача требует описания абсолютно идентичного объекта в нескольких местах разметки. Помимо дублирования разметки также будет создано два разных объекта в памяти во время выполнения приложения, хотя можно было бы вполне обойтись одним. К сожалению, в XAML нет понятия переменных, которые очень подошли бы для решения данной проблемы.

Приведенный ниже фрагмент разметки демонстрирует ситуацию, в которой две кнопки требуют идентичной настройки свойств, что выливается в дублирование разметки (рис. 1) (полный пример находится в папке `Wpf. WithoutResources`):

XAML

```
<Window ...>
  <Grid Margin="5">

    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition Height="5"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
```

```

<Button Content="Button 1"
        FontSize="20"
        Foreground="White"
        Grid.Row="0"
        Padding="10">

    <Button.Background>
        <LinearGradientBrush EndPoint="1,1"
                               StartPoint="0,0">
            <GradientStop Color="Yellow"
                           Offset="0"/>
            <GradientStop Color="Red"
                           Offset="1"/>
        </LinearGradientBrush>
    </Button.Background>
</Button>

<Button Content="Button 2"
        FontSize="20"
        Foreground="White"
        Grid.Row="2"
        Padding="10">

    <Button.Background>
        <LinearGradientBrush EndPoint="1,1"
                               StartPoint="0,0">
            <GradientStop Color="Yellow"
                           Offset="0"/>
            <GradientStop Color="Red"
                           Offset="1"/>
        </LinearGradientBrush>
    </Button.Background>
</Button>

</Grid>
</Window>

```

```

<Grid Margin="5">
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="5"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Button Content="Button 1" FontSize="20" Foreground="White" Grid.Row="0" Padding="10">
    <Button.Background>
      <LinearGradientBrush EndPoint="1,1" StartPoint="0,0">
        <GradientStop Color="Yellow" Offset="0"/>
        <GradientStop Color="Red" Offset="1"/>
      </LinearGradientBrush>
    </Button.Background>
  </Button>
  <Button Content="Button 2" FontSize="20" Foreground="White" Grid.Row="2" Padding="10">
    <Button.Background>
      <LinearGradientBrush EndPoint="1,1" StartPoint="0,0">
        <GradientStop Color="Yellow" Offset="0"/>
        <GradientStop Color="Red" Offset="1"/>
      </LinearGradientBrush>
    </Button.Background>
  </Button>
</Grid>

```

Рисунок 1. Дублирование XAML-разметки

Для решения описанной выше проблемы XAML предоставляет так называемые ресурсы. Ресурсом является объект, который можно повторно использовать в нескольких местах приложения. В качестве ресурса может выступать практически любой объект, например, кисть, стиль или шаблон элемента управления. Доступ к ресурсам можно получить с программного кода и с XAML-разметки.

Приведенный ниже фрагмент разметки демонстрирует использование одних и тех же ресурсов несколькими элементами управления (полный пример находится в папке `Wpf.Resources`):

XAML

```

<Window ...
  xmlns:System= "clr-namespace:System;
                  assembly=mscorlib"
  ...>

```

```

<Window.Resources>
    <LinearGradientBrush x:Key="backgroundBrush"
                        EndPoint="1,1"
                        StartPoint="0,0">
        <GradientStop Color="Yellow" Offset="0"/>
        <GradientStop Color="Red" Offset="1"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="foregroundBrush">
        White</SolidColorBrush>
    <Thickness x:Key="padding">10</Thickness>
    <System.Double x:Key="fontSize">20
    </System.Double>
</Window.Resources>
<Grid Margin="5">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Background=
        "{StaticResource backgroundBrush}"
        Content="Button 1"
        FontSize="{StaticResource fontSize}"
        Foreground=
            "{StaticResource foregroundBrush}"
        Grid.Row="0"
        Padding="{StaticResource padding}"/>
    <Button Background=
        "{StaticResource backgroundBrush}"
        Content="Button 2"
        FontSize="{StaticResource fontSize}"
        Foreground=
            "{StaticResource foregroundBrush}"
        Grid.Row="2"
        Padding="{StaticResource padding}"/>
</Grid>
</Window>

```


Результат приведенной выше разметки показан на рис. 2.

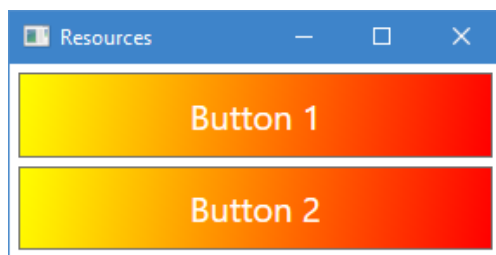


Рисунок 2. Использование ресурсов

Как можно заметить из приведенной выше разметки, для того, чтобы начать использовать ресурсы необходимо выполнить три условия:

- Объект-ресурс должен быть помещен в свойство `System.Windows.FrameworkElement.Resources`.
- Объект-ресурс должен обладать уникально заданным идентификатором, по которому будет осуществляться доступ к нему.
- Установка ресурса в качестве используемого значения для атрибута осуществляется посредством расширения разметки `System.Windows.StaticResourceExtension`.

Каждый элемент унаследованный от класса `System.Windows.FrameworkElement` обладает свойством `Resources`, которое содержит все его ресурсы в виде ассоциативной коллекции, где ключом является название ресурса, а значением сам ресурс. Каждый ресурс должен обладать уникальным идентификатором в пределах этой коллекции. Для того, чтобы указать идентификатор ресурса необходимо добавить ему атрибут `x:Key`. Обычно, в качестве иденти-

фикатора выступает строка. Однако, используя дополнительные расширения разметки, возможно использование объектов другого типа. Не строковые идентификаторы применяются некоторыми механизмами WPF, в особенности, стилями, о которых речь пойдет дальше.

После того, как ресурс был объявлен, его можно использовать в качестве значения свойства, используя расширение разметки, которое принимает в качестве аргумента идентификатор ресурса, например:

XAML

```
<Label Background="{StaticResource someBrush}"/>  
<Button FontSize="{StaticResource normalFontSize}"/>  
<ListBox Items="{StaticResource itemList}"  
          Margin="{StaticResource defaultMargin}"/>
```

1.1. Перекрывающиеся ресурсы

Довольно часто возникает необходимость определять ресурс по умолчанию с возможностью переопределения. Например, можно определить ресурс описывающий минимальную ширину для кнопки. Если такой ресурс будет помещен в коллекцию ресурсов приложения, то все кнопки приложения смогут использовать данный ресурс, но, что если в пределах одного из окон (или части окна) требуется другое значение для данного ресурса. Фактически в такой ситуации необходимо переопределить значение уже существующего ресурса.

В пределах коллекции ресурсов одного элемента не может быть двух объектов с одним и тем же идентификатором, но разные коллекции могут содержать повторе-

ния. При этом будет выбран ресурс, который объявлен наиболее «близко» к использующему его элементу.

Приведенный ниже фрагмент разметки демонстрирует перекрытие ресурсов с одинаковыми идентификаторами (полный пример находится в папке `Wpf.Resources.Overlap`):

XAML

```
<Window ...>
  <Window.Resources>
    <SolidColorBrush x:Key= "button1BackgroundBrush">
      Red </SolidColorBrush>
    <SolidColorBrush x:Key="button2BackgroundBrush">
      Green </SolidColorBrush>
    <SolidColorBrush x:Key="button3BackgroundBrush">
      Blue </SolidColorBrush>
  </Window.Resources>

  <Grid Margin="5">
    <Grid.Resources>
      <SolidColorBrush x:Key=
        "button2BackgroundBrush">
        Yellow </SolidColorBrush>
    </Grid.Resources>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition Height="5"/>
      <RowDefinition/>
      <RowDefinition Height="5"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Background=
      "{StaticResource button1BackgroundBrush}"
      Content="Button 1"
      Grid.Row="0"/>
    <Button Background=
      "{StaticResource button2BackgroundBrush}"
```

```

        Content="Button 2"
        Grid.Row="2"/>
<Button Background=
    "{StaticResource button3BackgroundBrush}"
    Content="Button 3"
    Grid.Row="4">
    <Button.Resources>
        <SolidColorBrush x:Key=
            "button3BackgroundBrush">
            Purple </SolidColorBrush>
    </Button.Resources>
</Button>
</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 3.

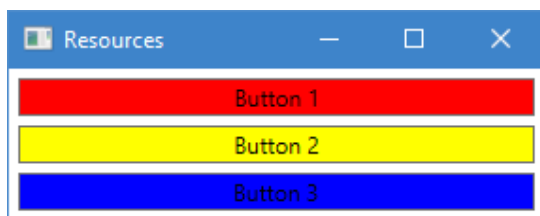


Рисунок 3. Перекрытие ресурсов

Когда во время обработки разметки XAML-анализатор обнаруживает расширение разметки `System.Windows.StaticResourceExtension`, то выполняется поиск ресурса с указанным идентификатором начиная от текущего элемента с продвижением вверх по логическому дереву. Если в элементе, для которого используется ресурс не было обнаружено требуемого идентификатора, то поиск осуществляется в родительском элементе и так далее, до тех

пор, пока алгоритм не дойдет до корневого элемента, которым является окно в данном случае. Если в коллекции ресурсов корневого элемента также не будет найдено желаемого ресурса, то поиск производится в ресурсах приложения, которые обычно описываются в пределах файла App.xaml.

Несмотря на то, что для третьей кнопки существует помещенный в нее ресурс с указанным идентификатором, будет выбран ресурс из родительского элемента. Это связано с тем, что при использовании расширения разметки `System.Windows.StaticResourceExtension`, учитываются только те ресурсы, которые были объявлены в файле выше.

1.2. Статические и динамические ресурсы

Ресурсы в WPF бывают двух видов: статические и динамические. Первый вид ресурсов описывает неизменные ресурсы, т.е. ресурс считывается единожды в момент присваивания его значения свойству элемента управления. Динамические ресурсы позволяют добиться эффекта «живого» обновления, т.е. если динамический ресурс будет использован для значения свойства какого-либо элемента управления, а после этого его значение будет изменено, то и значение свойства также изменится.

Коллекция, в которую помещаются ресурсы является динамической. Однако, использование расширения разметки `System.Windows.StaticResourceExtension`, не позволяет использовать данную ее функциональность практически никак.

Приведенный ниже фрагмент разметки демонстрирует изменение ресурса после его использования (полный пример находится в папке `Wpf.Resources.StaticProblem`):

XAML

```

<Window ...>
  <Window.Resources>
    <LinearGradientBrush
      x:Key="borderBackgroundBrush"
      EndPoint="1,1" StartPoint="0,0">
      <GradientStop Color="Yellow" Offset="0"/>
      <GradientStop Color="Green" Offset="0.5"/>
      <GradientStop Color="Purple" Offset="1"/>
    </LinearGradientBrush>
  </Window.Resources>
  <Grid Margin="5">
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition Height="5"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Border Background=
      "{StaticResource borderBackgroundBrush}"
      Grid.Row="0"/>
    <Button Click="Button_Click" Grid.Row="2">
      Change Resource </Button>
  </Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 4.

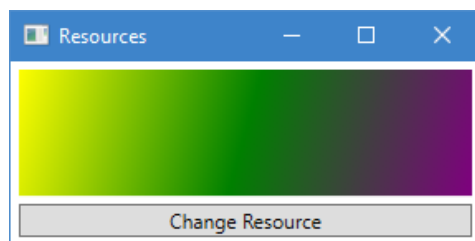


Рисунок 4. Изменение статического ресурса после его использования

Соответствующий файл с программным кодом (модель code-behind) содержит следующий фрагмент программного кода:

C#

```
private void Button_Click(object sender,
                           RoutedEventArgs e)
{
    Resources["borderBackgroundBrush"] = Brushes.Red;
}
```

При нажатии на кнопку ресурс с указанным идентификатором в коллекции ресурсов изменится на новый, но визуально ничего не изменится. Для того, чтобы получить визуальный отклик, т.е. добиться эффекта динамического ресурса (автоматическое обновление свойства вслед за обновлением ресурса) вместо статического, необходимо использовать другое расширение разметки для указания используемого ресурса — `System.Windows.DynamicResourceExtension` (полный пример находится в папке `Wpf.Resources.Dynamic`). Оба расширения разметки принимают название (идентификатор) ресурса в качестве аргумента.

При использовании статического ресурса (расширение разметки `System.Windows.StaticResourceExtension`) производится поиск ресурса во всех доступных местах. Это производится во время загрузки, в тот момент, когда необходимо получить значение для свойства, т.е. во время анализа XAML-разметки.

При использовании динамического ресурса (расширение разметки `System.Windows.DynamicResourceExtension`)

вместо поиска необходимого ресурса в момент загрузки создается специальный объект, который производит поиск ресурса непосредственно при необходимости, т.е. когда значение данного свойства потребует считывания.

Статические ресурсы предоставляют меньше возможностей, но при этом они работают немного быстрее. Поэтому динамические ресурсы стоит использовать если планируется динамическая подмена ресурсов в приложении. Самым наглядным примером такой подмены может быть переключение различных тем оформления. Также, из-за своего особо устройства, динамические ресурсы можно использовать только совместно со свойствами объектов, производных от класса `System.Windows.DependencyObject`.

Использование статических ресурсов наиболее уместно в следующих ситуациях:

- Большинство ресурсов приложения объявляются на уровне корневых элементов (например, окон) или на уровне приложения.
- Необходимость установки ресурса в качестве значения объекту не производному от класса `System.Windows.DependencyObject`.
- Ресурсы компилируются в виде отдельной библиотеки и поставляются вместе с приложением или используются несколькими приложениями совместно.
- При создании собственной темы оформления ресурсы используемые внутри темы должны быть «предсказуемыми» и не зависеть от возможного их переопределения.

Использование динамических ресурсов наиболее уместно в следующих ситуациях:

- Значение ресурса зависит от условий, значения которых не известны до момента выполнения приложения.
- Использование тем оформления, с возможностью переключения без перезапуска приложения.
- Коллекция ресурсов может быть изменена на этапе выполнения приложения.

1.3. Объединенные словари ресурсов

Использование ресурсов позволяет существенно сократить разметку и программный код за счет того, что убирается дублирование большей части кода. С ростом приложения будут увеличиваться и размеры файлов, содержащих ресурсы. Например, если приложение состоит из нескольких окон, которые используют совместно разнообразные ресурсы, то эти ресурсы в итоге будут вынесены на уровень ресурсов приложения, т.е. в один общий файл. При таком подходе размер этого файла может достигнуть очень больших размеров и ориентироваться в нем станет практически невозможно. К сожалению, XAML не поддерживает концепцию разделения одного файла на несколько отдельных. Однако, существует механизм, позволяющий вынести ресурсы в отдельный файл или несколько отдельных файлов.

WPF ресурсы поддерживают так называемую концепцию объединенных словарей. Для того, чтобы воспользоваться данным механизмом, необходимо создать XAML-файл, в котором корневым элементом должен быть элемент `ResourceDictionary`. Данный элемент описывает

словарь, который необходимо объединить с каким-то другим словарем ресурсов.

Приведенный ниже фрагмент разметки демонстрирует использование словарей ресурсов для вынесения и группировки схожих ресурсов (цвета и кисти) в отдельные файлы, что позволяет добиться лучшего структурирования проекта, а также предоставляет возможность использования одних и тех же ресурсов в различных файлах, не вынося их на уровень ресурсов всего приложения (полный пример находится в папке `Wpf.Resources.MergedDictionary`):

XAML

```
<!-- Colors.xaml -->
<ResourceDictionary xmlns=
    "http://schemas.microsoft.com/
    winfx/2006/xaml/presentation"
    xmlns:x=
    "http://schemas.microsoft.
    com/winfx/2006/xaml">
    <Color x:Key="backgroundColor1">DarkGray</Color>
    <Color x:Key="backgroundColor2">Orange</Color>
</ResourceDictionary>

<!-- Brushes.xaml -->
<ResourceDictionary xmlns=
    "http://schemas.microsoft.com/
    winfx/2006/xaml/presentation"
    xmlns:x=
    "http://schemas.microsoft.com/
    winfx/2006/xaml">
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="/Colors.xaml"/>
    </ResourceDictionary.MergedDictionaries>
```

```

        <SolidColorBrush x:Key="borderBackgroundBrush1"
                        Color="{StaticResource
                                backgroundColor1}"/>
        <SolidColorBrush x:Key="borderBackgroundBrush2"
                        Color="{StaticResource
                                backgroundColor2}"/>
    </ResourceDictionary>

    <Window ...>
        <Window.Resources>
            <ResourceDictionary>
                <ResourceDictionary.MergedDictionaries>
                    <ResourceDictionary
                        Source="/Brushes.xaml"/>
                </ResourceDictionary.MergedDictionaries>
                <SolidColorBrush
                    x:Key="borderBackgroundBrush1"
                    Color="DarkGray"/>
            </ResourceDictionary>
        </Window.Resources>

        <Grid Margin="5">
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition Width="5"/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <Border Background="{StaticResource
                                borderBackgroundBrush1}"
                    Grid.Column="0"/>
            <Border Background="{StaticResource
                                borderBackgroundBrush2}"
                    Grid.Column="2"/>
        </Grid>
    </Window>

```

Результат приведенной выше разметки показан на рис. 5.



Рисунок 5. Использование объединенных словарей ресурсов

Как видно из приведенного примера, для того, чтобы использовать объединенный словарь ресурсов, необходимо поместить элемент `<ResourceDictionary>` в описание свойства `System.Windows.FrameworkElement.Resource` желаемого элемента (обычно, корневой элемент). Стоит заметить, что у данного элемента не указывается атрибут `x:Key`, который является обязательным при объявлении ресурса. Наличие еще одного элемента `<ResourceDictionary>`, помещенного в свойство `MergedDictionaries` является особым случаем для реализации данного сценария. Для того, чтобы указать где именно располагается словарь, который необходимо объединить с ресурсами, используется свойство `Source`.

В качестве альтернативы или в дополнение к использованию свойства `Source`, возможно объявить ресурсы непосредственно внутри элемента `<ResourceDictionary>`. Данный подход не является распространенным. В подобной ситуации проще было бы поместить ресурсы в основной словарь, а не присоединенный.

Во время поиска необходимого ресурса (статического или динамического) по идентификатору, объединенные словари проверяются сразу после основного словаря элемента, до перехода к проверке словаря ресурсов родительского элемента (в случае отсутствия необходимого ресурса). Хотя идентификатор ресурса должен быть уникальным в пределах одного словаря ресурсов, он может повторяться в различных объединенных словарях одного и того же элемента и это не будет генерировать ошибку. В такой ситуации будет использован ресурс, который был найден в последнем из словарей, которые его содержат. Если при этом ресурс с одним и тем же идентификатором будет обнаружен в основном словаре и в объединенном словаре, то будет использован ресурс из основного словаря, т.е. по-прежнему срабатывает правило «ближайшего» объявления ресурса к элементу, который его использует.

2. Стили

Рассмотренные выше ресурсы отлично справляются с проблемой дублирования значений для отдельных свойств, но при этом другая проблема не может быть решена лишь с применением ресурсов. Например, если группа кнопок пользовательского интерфейса приложения должны выглядеть идентично и при этом содержать несколько значений свойств, которые отличаются от значений по умолчанию, то, даже применяя ресурсы, придется каждой кнопке явным образом указывать какой именно ресурс использовать для какого свойства. Если в будущем понадобится добавить настройку еще одного свойства всем этим кнопкам (например, сделать вывод текста курсивом), то придется каждой кнопке явным образом дописать еще одно свойство.

Для упрощения подобного процесса WPF предлагает использование стилей. Стилй представляет из себя набор пар свойство-значение. После того как стилй будет описан, его можно установить любому количеству элементов или даже настроить таким образом, чтобы он использовался по умолчанию для всех элементов, у которых стилй не задан явным образом. Такая операция позволит настроить сразу все свойства описанные внутри стилиа всем связанным с ним элементам. Более того, при необходимости добавления, удаления или изменения свойства в стилие, это будет сделано в одном месте, а обновление получат все элементы применяющие его.

Приведенный ниже фрагмент разметки демонстрирует использование стилей для того, чтобы задать одинаковые настройки кнопкам, которые принадлежат к одной и той же логической группе (полный пример находится в папке `Wpf.Styles`):

XAML

```
<Window ...>
  <Window.Resources>
    <Style x:Key="darkButtonStyle"
      TargetType="{x:Type Button}">
      <Setter Property="Background"
        Value="#FF333333"/>
      <Setter Property="FontWeight"
        Value="Bold"/>
      <Setter Property="Foreground"
        Value="#FFCCCCCC"/>
      <Setter Property="Padding" Value="10"/>
    </Style>

    <Style x:Key="lightButtonStyle"
      TargetType="{x:Type Button}">
      <Setter Property="Background"
        Value="Yellow"/>
      <Setter Property="FontStyle"
        Value="Italic"/>
      <Setter Property="Foreground"
        Value="Black"/>
    </Style>
  </Window.Resources>

  <Grid Margin="5">
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition Height="5"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
  </Grid>
</Window>
```

```

        <RowDefinition Height="5"/>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <Button Grid.Row="0"
            Style="{StaticResource darkButtonStyle}">
        Dark Button 1
    </Button>

    <Button Grid.Row="2"
            Style="{StaticResource darkButtonStyle}">
        Dark Button 2
    </Button>

    <Button Grid.Row="4"
            Style="{StaticResource lightButtonStyle}">
        Light Button 1
    </Button>

    <Button Grid.Row="6"
            Style="{StaticResource lightButtonStyle}">
        Light Button 2
    </Button>

    <Button Grid.Row="8">
        Simple Button
    </Button>

</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 6.

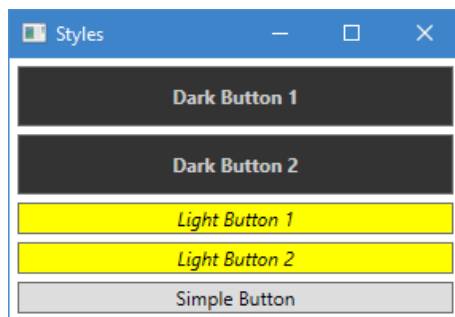


Рисунок 6. Использование стилей

Стиль может быть описан непосредственно при объявлении элемента, используя синтаксис элемента-свойства, но на практике это не применяется так как стиль создается для того, чтобы его можно было применить для нескольких элементов. При объявлении стиля внутри одного из элементов он не будет «виден» всем остальным, поэтому стили, обычно, помещают в ресурсы.

Из приведенного выше примера также можно заметить, что описание стиля сводится к описанию требуемых для настройки свойств и заданию им значений. Это выполняется при помощи элемента `<Setter>`, который содержит два атрибута: `Property` и `Value`, которые описывают название свойства и его значение соответственно.

Также при объявлении стиля необходимо указать значение его атрибуту `TargetType`, который указывает для какого целевого типа описывается данный стиль. Это необходимо для того, чтобы было понятно какой набор свойств доступен для настройки и чтобы не возникало ситуаций, подобных такой, при которой производится настройка свойства `Text` для элемента `System.Windows.Controls.ListBox`, которого у него нет. Для данного атри-

бута можно указать не конечный тип элемента, а базовый, например, `System.Windows.Controls.Control`.

2.1. Переопределение свойств стиля

Описывая стиль по сути создается группа настроек для элемента управления, к которому он будет применен. Однако стиль стоит рассматривать не просто, как группу настроек применяемых к элементу управления, а как настройки применяемые к нему «по умолчанию», т.е. стиль задает набор настроек, но каждый элемент в праве переопределить любую из них для себя. Это позволяет описывать в стиле значения для свойств, которые наиболее часто будут встречаться у элементов, к которым стиль будет применен, но при этом будет возможность более тонкой настройки для каждого отдельного элемента.

Приведенный ниже фрагмент разметки демонстрирует использование стиля для элемента с теми же явно указанными свойствами (полный пример находится в папке `Wpf.Styles.Overlap`):

XAML

```
<Window ...>
  <Window.Resources>
    <Style x:Key="textBlockStyle"
      TargetType="{x:Type TextBlock}">
      <Setter Property="Background"
        Value="Violet"/>
      <Setter Property="FontSize" Value="20"/>
      <Setter Property="FontWeight"
        Value="Bold"/>
      <Setter Property="HorizontalAlignment"
        Value="Center"/>
    </Style>
  </Window.Resources>
  <TextBlock/>
</Window>
```

```

        <Setter Property="Padding" Value="30"/>
        <Setter Property="VerticalAlignment"
            Value="Center"/>
    </Style>
</Window.Resources>

<Grid Margin="5" UseLayoutRounding="True">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <TextBlock Background="Yellow"
        Grid.Row="0"
        Style="{StaticResource textBlockStyle}"
        Text="Some Text"/>
    <TextBlock Grid.Row="2"
        Style="{StaticResource textBlockStyle}"
        Text="Some Text"/>

</Grid>
</Window>

```

Результат приведенной разметки показан на рис. 7.



Рисунок 7. Наложение свойств стиля и свойств элемента управления

В приведенном выше примере свойство, отвечающее за цвет заднего фона установлено в стиле, а также для одного из элементов указано явным образом при его объявлении. Явное указание значения при объявлении элемента считается более приоритетным и перекрывает значение установленное стилем.

Данную ситуацию можно представить иначе: стиль может определять набор настроек по умолчанию, с возможностью переопределения необходимых для целевого элемента управления.

2.2. Стили по умолчанию

При разработке приложений часто встречается ситуация, когда для всех экземпляров одного элемента управления (например, кнопки) необходимо определить один и тот же стиль. Явное указание при объявлении каждого элемента несомненно решит проблему, но WPF предоставляет более простой способ добиться желаемого результата. Можно воспользоваться особенностью стиля, которая позволяет при его объявлении указать, что его необходимо использовать для всех объектов определенного типа. Для этого достаточно опустить атрибут **x:Key**.

Приведенный ниже фрагмент разметки демонстрирует использование стиля по умолчанию (полный пример находится в папке `Wpf.Styles.Default`):

XAML

```
<Window ...>
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Background" Value="Lime"/>
    </Style>
  </Window.Resources>
</Window>
```

```

        <Setter Property="Padding" Value="10"/>
    </Style>
    <Style x:Key="specialButtonStyle"
        TargetType="{x:Type Button}">
        <Setter Property="Background"
            Value="Aqua"/>
        <Setter Property="FontStyle" Value="Italic"/>
    </Style>
</Window.Resources>
<Grid Margin="5">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Row="0">Button 1</Button>
    <Button Grid.Row="2">Button 2</Button>
    <Button Grid.Row="4"
        Style="{StaticResource
            specialButtonStyle}">
        Button 3 </Button>
</Grid>
</Window>

```

Результат приведенной разметки показан на рис. 8.

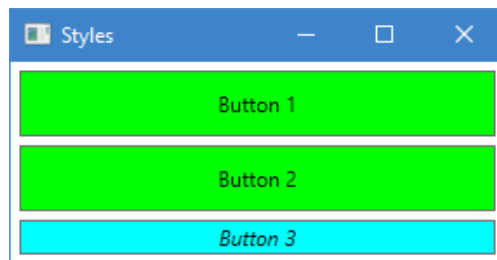


Рисунок 8. Использование стилей по умолчанию

Если объявить стиль по умолчанию, то он будет использоваться только для тех элементов управления, у которых стиль не задан явным образом. Более того, если объявить «именованный» стиль для такого же типа элементов управления, то он будет строиться не на базе первого, а на базе того стиля, который предоставляется WPF по умолчанию.

2.3. Наследование стилей

Создавая несколько вариантов стилей для одного и того же вида элементов управления часто можно столкнуться с ситуацией, что такие стили в основном содержат одинаковый набор свойств и отличаются всего лишь парой из них. Например, создавая несколько различных стилей для кнопок, в зависимости от их контекста использования, может случиться, что у них отличается всего лишь цветовое оформление, а все настройки отступов, выравниваний и шрифтов идентичны. Для того, чтобы избежать дублирования одинаковых свойств можно воспользоваться наследованием шрифтов. Данный механизм идентичен наследованию классов из ООП, при котором новый стиль получает все настройки базового стиля с возможностью переопределения всех или части из них а также с возможностью добавить новые.

Приведенный ниже фрагмент разметки демонстрирует наследование стилей (полный пример находится в папке `Wpf.Styles.Inheritance`):

XAML

```
<Window ...>
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
```

```

        <Setter Property="FontSize" Value="15"/>
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="Padding" Value="10"/>
        <Setter Property="Typography.Capitals"
            Value="SmallCaps"/>
    </Style>
    <Style x:Key="acceptButtonStyle"
        BasedOn="{StaticResource {x:Type Button}}"
        TargetType="{x:Type Button}">
        <Setter Property="Background" Value="Green"/>
        <Setter Property="Foreground" Value="White"/>
    </Style>
    <Style x:Key="declineButtonStyle"
        BasedOn="{StaticResource
            acceptButtonStyle}"
        TargetType="{x:Type Button}">
        <Setter Property="Background"
            Value="Red"/>
    </Style>
</Window.Resources>
<Grid Margin="5">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Row="0">Common Button</Button>
    <Button Grid.Row="2"
        Style="{StaticResource
            acceptButtonStyle}">
        Accept Button
    </Button>
    <Button Grid.Row="4"
        Style="{StaticResource
            declineButtonStyle}">

```

```

        Decline Button
    </Button>
</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 9.

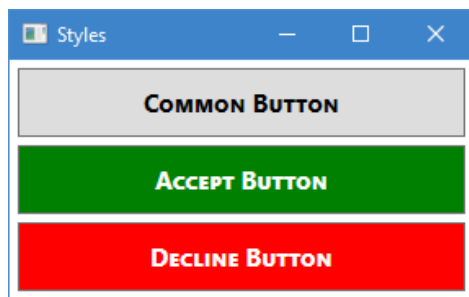


Рисунок 9. Наследование стилей

Для того, чтобы указать стиль на основании которого создается новый (наследуется), необходимо использовать атрибут **BasedOn**, где значением стоит указать базовый стиль. В большинстве ситуаций самым уместным решением будет указать ссылку на статический ресурс. Однако, если необходимо наследовать настройки стиля по умолчанию такой подход не сработает, так как стиль по умолчанию не обладает идентификатором. В такой ситуации, в качестве значения ресурса можно указать название типа данных элемента, используя расширение разметки `x:Type` или строковый литерал.

3. Шаблоны элементов управления

Одной из отличительных особенностей WPF от других технологий для разработки приложений с графическим пользовательским интерфейсом является возможность полного переопределения внешнего вида элементов управления не затрагивая при этом их поведение. Для того, чтобы получить новое визуальное оформление для элемента управления, необходимо описать для него шаблон. Шаблоны элементов управления описывают как именно должен выглядеть элемент управления, применяя при этом примитивные элементы или другие элементы управления. Пользуясь такой техникой, можно, к примеру, сделать все кнопки в приложении круглыми вместо прямоугольных, и они при этом продолжают генерировать события нажатия и вызывать привязанные к ним команды.

Элементы управления обладают широким набором свойств, значения которых можно изменить для того, чтобы получить новый внешний вид для них. Среди них есть свойства, которые позволяют настроить кисть для заднего фона, переднего плана, толщину и цвет рамки, стиль и размер шрифта. Например, можно настроить элемент управления переключатель таким образом, что текст отображался синим цветом и при этом был курсивным. Однако, возможности этих свойств в общем и целом ограничены. Без возможности переопределения структуры элемента управления, все они будут

обладать одинаковым внешним видом среди различных WPF-приложений, а это в свою очередь, ограничивает возможности по созданию приложений с уникальным стилем и ощущением.

По умолчанию все элементы управления переключатели отображаются одинаково: содержимое располагается справа, а включенное состояние идентифицируется видимой галочкой. Когда возникает необходимость выйти за границы переопределения стандартных свойств и придать совершенно новый вид элементу управления (рис. 10) необходимо использовать шаблоны элементов управления.

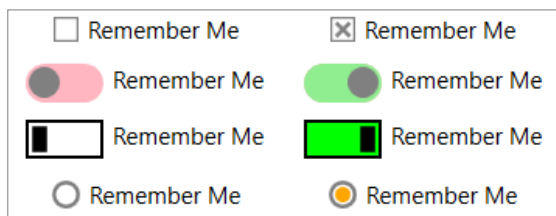


Рисунок 10. Переключатели с различными шаблонами элементов управления

Приведенный ниже фрагмент разметки демонстрирует создание собственного шаблона для кнопки (полный пример находится в папке `Wpf.ControlTemplates`):

XAML

```
<Window ...>
  <Grid Margin="5">
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition Height="5"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
```

```

<Button Content="Button with Default Template"
        Grid.Row="0"/>
<Button Content="Button with Custom Template"
        Grid.Row="2">
    <Button.Template>
        <ControlTemplate TargetType=
            "{x:Type Button}">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition/>
                    <ColumnDefinition/>
                    <ColumnDefinition/>
                </Grid.ColumnDefinitions>
                <Border Background="Red"
                        CornerRadius=
                            "10,0,0,10"
                        Grid.Column="0"/>
                <Border Background="Green"
                        Grid.Column="1"/>
                <Border Background="Blue"
                        CornerRadius=
                            "0,10,10,0"
                        Grid.Column="2"/>
                <Border Background="Yellow"
                        CornerRadius="10"
                        Grid.Column="0"
                        Grid.ColumnSpan="3"
                        HorizontalAlignment=
                            "Center"
                        Margin="0,5"
                        VerticalAlignment=
                            "Center">
                    <ContentPresenter Margin=
                        "5"/>
                </Border>
            </Grid>
        </ControlTemplate>

```

```

        </Button.Template>
    </Button>
</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 11.

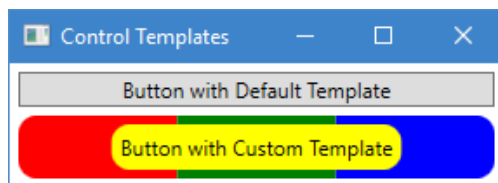


Рисунок 11. Использование шаблона элемента управления для кнопки

Шаблон элемента управления представляет из себя совокупность объектов `System.Windows.FrameworkElement`. При описании нового шаблона элемента управления они комбинируются для достижения желаемого результата. Элемент `<ControlTemplate>` может содержать только один дочерний элемент, который должен обладать типом производным от `System.Windows.FrameworkElement`. Чаще всего в роли корневого элемента шаблона используется одна из панелей. В конечном итоге, комбинация элементов определяет результирующий внешний вид элемента управления. Данный процесс похож на верстку пользовательского интерфейса всего приложения, только в немного меньшем масштабе.

В приведенном выше примере описывается шаблон для кнопки, который определяет ее новый внешний вид. Самым важным элементом в данном случае является эле-

мент **<ContentPresenter>**. Он берет на себя все обязанности по отображению содержимого кнопки. Стоит напомнить, что в WPF многие элементы предоставляют особую модель управления содержимым, что позволяет помещать в них любые объекты, а не только строки. Элемент **<ContentPresenter>** отвечает за то, чтобы помещенное в элемент управления содержимое отображалось согласно этой модели.

3.1. Расширение разметки TemplateBinding

Создавая собственные шаблоны для элементов управления многие разработчики сталкиваются с одной проблемой: стандартные свойства элемента управления, отвечающие за цвет заднего фона, переднего плана, рамку и т.д. остаются не у дел. Так получается в следствии того, что описывая шаблон элемента управления приходится явным образом указывать значения для аналогичных свойств элементам из которых состоит шаблон.

Приведенный ниже фрагмент разметки демонстрирует ситуацию, при которой настройка цвета заднего фона для элемента управления при его объявлении не имеет никакого эффекта (полный пример находится в папке `Wpf.ControlTemplates.WithoutTemplateBinding`):

XAML

```
<Window ...>
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType=
            "{x:Type Button}">
```

```

        <Border Background="Lime"
                CornerRadius="10">
            <ContentPresenter
                HorizontalAlignment=
                    "Center"
                Margin="5"
                VerticalAlignment=
                    "Center"/>
        </Border>
    </ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Window.Resources>
<Grid Margin="5">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Background="Aqua" Grid.Row="0">
        Button 1</Button>
    <Button Grid.Row="2">Button 2</Button>
</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 12.

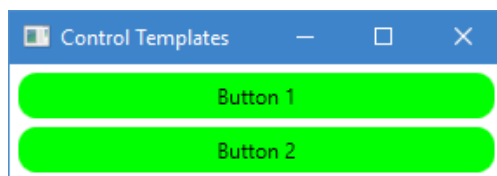


Рисунок 12. Шаблон элемента управления без привязки к свойствам элемента управления

В приведенном выше примере создается две кнопки, имеющих один и тот же шаблон элемента управления. Цвет заднего фона установлен в шаблоне явным образом для элемента `<Border>`. Несмотря на то, что для одной из кнопок цвет заднего фона устанавливается явным образом в другое значение, это не дает абсолютно никакого эффекта, так как описанный шаблон элемента управления не использует значения из свойств элемента управления. Такое поведение может немного запутывать, так как установить значение для свойства возможность есть, а результата нет.

Для того, чтобы добиться нужного эффекта, необходимо использовать механизм привязки данных рассмотренный ранее, но привязываться потребуется не к свойствам источника данных, помещенного в `System.Windows.FrameworkElement.DataContext`, а к свойствам элемента управления, для которого будет применен шаблон. Для этого необходимо задать источник при описании привязки следующим образом:

XAML

```
{Binding RelativeSource={RelativeSource TemplatedParent},  
    Path=SomeProperty}
```

Так как такой сценарий использования привязки данных применяется крайне часто, существует специальный сокращенный вариант:

XAML

```
{TemplateBinding Path=SomeProperty}
```

Расширение разметки `System.Windows.TemplateBinding` является особой версией расширения разметки `System.Windows.Data.Binding` с определенными оптимизациями, нацеленными на сценарии с участием шаблонов элементов управления. Одной из таких оптимизаций является то, что такой вид привязки всегда однонаправленный, даже если свойство поддерживает двунаправленный вариант.

Приведенный ниже фрагмент разметки демонстрирует использование привязки к свойствам элемента управления внутри шаблона (полный пример находится в папке `Wpf.ControlTemplates.WithTemplateBinding`):

XAML

```
<Window ...>
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Background"
        Value="Lime"/>
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType=
            "{x:Type Button}">

            <Border Background=
              "{TemplateBinding
                Background}"
              CornerRadius="10">
              <ContentPresenter
                HorizontalAlignment=
                  "Center"
                Margin="5"
                VerticalAlignment=
                  "Center"/>
            </Border>
```



```

        </ControlTemplate>
    </Setter.Value>
</Setter>
</Style>
</Window.Resources>

<Grid Margin="5">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Background="Aqua" Grid.Row="0">
        Button 1</Button>
    <Button Grid.Row="2">Button 2</Button>
</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 13.

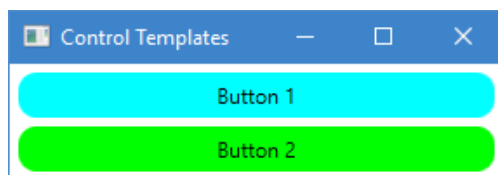


Рисунок 13. Шаблон элемента управления с привязкой к свойствам элемента управления

3.2. Триггеры

Во всех рассмотренных ранее примерах, связанных со стилями и шаблонами элементов управления присутствовал один недочет: вся визуальная настройка затрагивала только одно состояние элемента — основное. При

этом никаким образом не описывалось то, как элемент управления должен выглядеть, например, при наведении на него курсора мыши или при нажатии на него.

Шаблоны элементов управления могут содержать так называемые триггеры. Триггер представляет из себя механизм, который способен изменить набор свойств элемента (наподобие как это делает стиль) при изменении одного или нескольких свойств элемента. Триггеры можно применять для того, чтобы менять визуализацию элемента управления в зависимости от значений его свойств или данных, которые он хранит.

Концепция триггеров применима к шаблонам элементов управления, стилям и шаблонам данных. При этом они обладают практически одинаковым набором возможностей. Исключением является то, что при описании триггеров шаблона элемента управления или шаблона данных есть возможность взаимодействовать с отдельными элементами шаблона. При использовании триггера внутри стиля есть возможность взаимодействовать лишь со всем элементом целиком, т.е. с логической точки зрения, а не со структурной.

3.2.1. Триггеры свойств

Первый и наиболее часто встречающийся вид триггера срабатывает, когда указанное свойство элемента управления получает требуемое значение. Например, можно создать триггер, который будет «слушать» свойство, которое указывает расположен ли курсор мыши в области над кнопкой. Если данное свойство изменит свое значение с **false** на **true**, то триггер сработает и из-

меняет требуемые свойства элемента, например, цвет рамки и заднего фона, чтобы получить эффект «подсветки» при наведении курсора. После того, как свойство вернется в прежнее состояние эффект триггера (цвет рамки и заднего фона) будут возвращены в прошлое их состояние.

Приведенный ниже фрагмент разметки демонстрирует использование триггер свойства (полный пример находится в папке `Wpf.Triggers.PropertyTrigger`):

XAML

```
<Window ...>
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Background"
        Value="#FF0055FF"/>
      <Setter Property="BorderBrush"
        Value="DarkGray"/>
      <Setter Property="BorderThickness"
        Value="2"/>
      <Setter Property="FontWeight"
        Value="Bold"/>
      <Setter Property="Foreground"
        Value="White"/>
      <Setter Property="Padding" Value="5"/>
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType=
            "{x:Type Button}">
            <Border Background=
              "{TemplateBinding
                Background}"
              BorderBrush=
                "{TemplateBinding
                  BorderBrush}"
```

```

        BorderThickness=
            "{TemplateBinding
            BorderThickness}"
        CornerRadius="5">
<ContentPresenter Content=
    "{TemplateBinding
    Content}"
    HorizontalAlignment=
    "{TemplateBinding
    HorizontalContentAlignment}"
    Margin=
    "{TemplateBinding Padding}"
    VerticalAlignment=
    "{TemplateBinding
    VerticalContentAlignment}"/>
</Border>

<ControlTemplate.Triggers>
    <Trigger Property= "IsMouseOver"
        Value="True">
        <Setter Property= "Background"
            Value= "#FF00AAFF"/>
    </Trigger>
    <Trigger Property="IsPressed"
        Value="True">
        <Setter Property="Background"
            Value="#FF0000FF"/>
    </Trigger>
</ControlTemplate.Triggers>

</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Window.Resources>
<Grid Margin="5">
    <Grid.RowDefinitions>

```

```

        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Row="0">Button 1</Button>
    <Button Grid.Row="2">Button 2</Button>

</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 14.



Рисунок 14. Триггер свойства

В приведенном выше примере триггер свойства используется для мониторинга свойств `System.Windows.UIElement.IsMouseOver` и `System.Windows.Controls.Primitives.ButtonBase.IsPressed`, которые отвечают за то, находится ли курсор мыши в области над элементом управления и нажата ли кнопка соответственно. Оба триггера срабатывают при каждом изменении указанных свойств, и если значение свойства равняется тому, что было указано в триггере (`true` в данном случае), то триггер срабатывает. Срабатывание триггера влечет за собой установку новых значений всем свойствам, перечисленным внутри триггера. Если же новое значение свойства не равняется тому, что указано в триггере, то эффект триггера «откатывается»,

т.е. все значения возвращаются на те, что были раньше. Это существенно облегчает работу, потому что не приходится каждый раз описывать по несколько триггеров для взаимоисключающих ситуаций.

Стоит также заметить, что срабатывают все возможные триггеры. Имеется ввиду, что если выполняются условия равенства для более, чем одного триггера, то применяются эффекты каждого из них в порядке их объявления. Это может привести к нежелательным последствиям, если активированные триггеры изменяют одно и то же свойство, так как по факту останется только эффект от последнего из них.

Если в прошлом примере поменять два триггера местами, то при нажатии на кнопку ее внешний вид останется таким же, как и при наведении курсора. За счет того, что при нажатии оба свойства имеют значение `true`, срабатывают оба триггера. Сперва, триггер, отвечающий за нажатие, меняет цвет заднего фона, а затем цвет заднего фона меняется еще раз, потому что при нажатии курсор мыши все еще находится в области над элементом управления. Последнее срабатывание полностью нивелирует эффект первого (полный пример находится в папке `Wpf.Triggers.Overlap`).

Помимо рассмотренного выше триггера WPF предоставляет аналогичный, но с возможностью задать более одного свойства для наблюдения.

Приведенный ниже фрагмент разметки демонстрирует использование триггера свойства для наблюдения за изменением сразу нескольких свойств (полный пример находится в папке `Wpf.Triggers.MultiPropertyTrigger`):

XAML

```

<Window ...>
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="MinWidth" Value="75"/>
    </Style>

    <Style TargetType="{x:Type TextBox}">
      <Style.Triggers>
        <MultiTrigger>
          <MultiTrigger.Conditions>
            <Condition Property="IsEnabled"
              Value="True"/>
            <Condition Property="IsMouseOver"
              Value="True"/>
          </MultiTrigger.Conditions>
          <MultiTrigger.Setters>
            <Setter Property="Background"
              Value="Violet"/>
          </MultiTrigger.Setters>
        </MultiTrigger>
      </Style.Triggers>
    </Style>

  </Window.Resources>
  <Grid Margin="5">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="5"/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition Height="5"/>
      <RowDefinition/>
      <RowDefinition Height="5"/>
  </Grid>

```

```

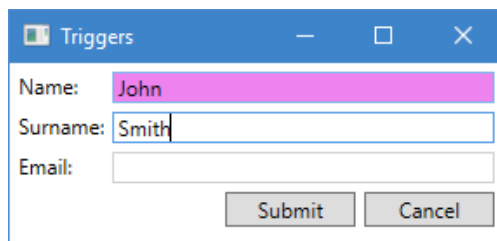
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <TextBlock Grid.Column="0"
               Grid.Row="0"
               Text="Name:"/>
    <TextBox Grid.Column="2"
             Grid.Row="0"/>
    <TextBlock Grid.Column="0"
               Grid.Row="2"
               Text="Surname:"/>
    <TextBox Grid.Column="2"
             Grid.Row="2"/>
    <TextBlock Grid.Column="0"
               Grid.Row="4"
               Text="Email:"/>
    <TextBox Grid.Column="2"
             Grid.Row="4"
             IsEnabled="False"/>
    <StackPanel Grid.Column="0"
                Grid.ColumnSpan="3"
                Grid.Row="6"
                HorizontalAlignment="Right"
                Orientation="Horizontal">
        <Button>Submit</Button>
        <Button Margin="5,0,0,0">Cancel</Button>
    </StackPanel>

</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 15.



The image shows a Windows-style dialog box titled "Triggers". It has a blue title bar with standard minimize, maximize, and close buttons. Inside the dialog, there are three text input fields. The first field is labeled "Name:" and contains the text "John". The second field is labeled "Surname:" and contains the text "Smith". The third field is labeled "Email:" and is currently empty. Below the input fields, there are two buttons: "Submit" and "Cancel".

Рисунок 15. Триггер свойства для наблюдения за несколькими свойствами

При использовании триггера способного наблюдать за изменением нескольких свойств необходимо заполнить коллекцию условий, в которой каждый элемент должен описывать отдельное условие равенства с желаемым свойством. Такой триггер сработает только в случае выполнения всех равенств из данной коллекции.

В стиле или шаблоне элемента управления можно применять различные триггеры вперемешку. Главным правилом, обычно, является наличие у них взаимоисключающих условий.

Описывая триггер, который располагается внутри шаблона элемента управления часто возникает необходимость изменять свойства конкретного элемента, который является частью структуры логического элемента. Обычное изменение свойства в данном случае не подойдет так как оно может использоваться для других целей или элемент может в принципе не располагать необходимым свойством в своем интерфейсе.

Приведенный ниже фрагмент разметки демонстрирует обращение к элементу шаблона элемента управления при изменении свойства на указанное значение (полный пример находится в папке `Wpf.Triggers.SetterTarget`):

XAML

```

<Window ...>
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Background"
        Value="#FF0055FF"/>
      <Setter Property="BorderBrush"
        Value="DarkGray"/>
      <Setter Property="BorderThickness"
        Value="2"/>
      <Setter Property="FontWeight"
        Value="Bold"/>
      <Setter Property="Foreground"
        Value="White"/>
      <Setter Property="Padding"
        Value="5"/>
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType=
            "{x:Type Button}">
            <Border Background=
              "{TemplateBinding
                Background}"
              BorderBrush=
                "{TemplateBinding
                  BorderBrush}"
              BorderThickness=
                "{TemplateBinding
                  BorderThickness}"
              CornerRadius="10">
            <Grid>
              <Border x:Name= "border"
                BorderBrush= "Aqua"
                BorderThickness= "2"
                CornerRadius="7"
                Margin="3"
                Visibility="Hidden"/>

```

```

        <ContentPresenter Content=
            "{TemplateBinding Content}"
            HorizontalAlignment=
            "{TemplateBinding
            HorizontalContentAlignment}"
            Margin=
            "{TemplateBinding Padding}"
            VerticalAlignment=
            "{TemplateBinding
            VerticalContentAlignment}"/>
    </Grid>
</Border>
<ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver"
        Value="True">
        <Setter TargetName="border"
            Property="Visibility"
            Value="Visible"/>
    </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Window.Resources>

<Grid Margin="5">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Row="0">Button 1</Button>
    <Button Grid.Row="2">Button 2</Button>
</Grid>

</Window>

```

Результат приведенной выше разметки показан на рис. 16.



Рисунок 16. Обращение к элементу шаблона внутри триггера

Стоит отметить, что имена присвоенные элементам внутри шаблона элемента управления видны только в пределах этого шаблона. Попытка обращения к ним за пределами шаблона сгенерирует ошибку компиляции. Для того, чтобы получить именованный элемент шаблона необходимо воспользоваться следующим программным кодом:

C#

```
Border borderInTemplate = (Border)someButton.  
    Template.FindName("border", someButton);
```

3.2.2. Триггер данных

Помимо триггера, который умеет наблюдать за изменением свойств элемента управления существует триггер способный применять стандартный механизм привязки данных. Это дает возможность наблюдать за свойствами любого другого объекта, который умеет генерировать соответствующие оповещения об их изменении.

Приведенный ниже фрагмент разметки демонстрирует использование триггера данных (полный пример находится в папке `Wpf.Triggers.DataTrigger`):

XAML

```

<Window ...>
  <Grid Margin="5">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="5"/>
      <RowDefinition/>
    </Grid.RowDefinitions>

    <TextBlock Grid.Row="0"
      Text="Transations"
      Typography.Capitals="SmallCaps"/>
    <ListBox Grid.Row="2"
      HorizontalContentAlignment="Stretch"
      ItemsSource="{Binding Transactions}">
      <ListBox.ItemTemplate>
        <DataTemplate>
          <Grid>
            <Grid.ColumnDefinitions>
              <ColumnDefinition/>
              <ColumnDefinition Width=
                "Auto"/>
            </Grid.ColumnDefinitions>

            <TextBlock Grid.Column="0"
              Text="{Binding
                Description}"/>
            <TextBlock x:Name=
              "moneyTextBlock"
              FontWeight="Bold"
              Grid.Column="1"
              HorizontalAlignment=
                "Right"
              Text="{Binding Money,
                Mode=OneWay,
                StringFormat={
                  {0:0.00}}"/>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>
    </Grid>
  </Window>

```

```

        </Grid>

        <DataTemplate.Triggers>

            <DataTrigger Binding="{Binding
                                IsExpense}"
                            Value="True">

                <Setter TargetName=
                    "moneyTextBlock"
                    Property=
                        "TextBlock.Foreground"
                    Value="Red"/>
            </DataTrigger>

            <DataTrigger Binding= "{Binding
                                IsIncome}"
                            Value="True">

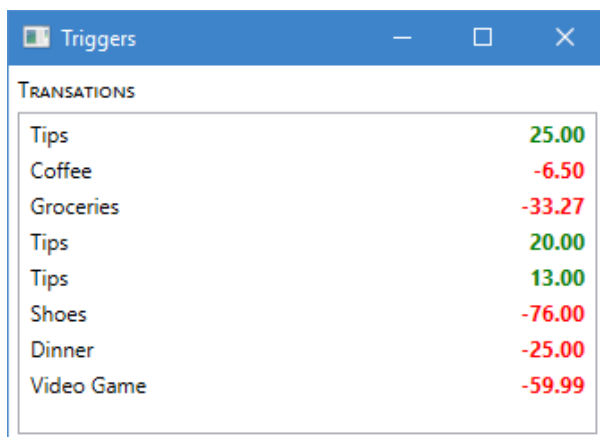
                <Setter TargetName=
                    "moneyTextBlock"
                    Property= "TextBlock.
                                Foreground"
                    Value="Green"/>
            </DataTrigger>

        </DataTemplate.Triggers>
    </DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
</Grid>

</Window>

```

Результат приведенной выше разметки показан на рис. 17.



TRANSACTIONS	
Tips	25.00
Coffee	-6.50
Groceries	-33.27
Tips	20.00
Tips	13.00
Shoes	-76.00
Dinner	-25.00
Video Game	-59.99

Рисунок 17. Триггер данных

В приведенном выше фрагменте программного кода, в качестве источника данных используется объект типа [DataSource](#), объявление которого выглядит следующим образом:

C#

```
internal sealed class DataSource
{
    private readonly IEnumerable <Transaction>
        transactions;

    public DataSource()
    {
        transactions = new[]
        {
            new Transaction("Tips", 25.0),
            new Transaction("Coffee", -6.5),
            new Transaction("Groceries", -33.27),
            new Transaction("Tips", 20.0),
            new Transaction("Tips", 13.0),
            new Transaction("Shoes", -76.0),

```

```

        new Transaction("Dinner", -25.0),
        new Transaction("Video Game", -59.99)
    };
}

public IEnumerable<Transaction> Transactions =>
    transactions;
}

```

Тип данных **Transaction**, который используется в источнике данных выглядит следующим образом:

C#

```

internal sealed class Transaction
{
    private readonly string description;
    private readonly double money;

    public Transaction(string description, double money)
    {
        this.description = description;
        this.money = money;
    }

    public string Description => description;

    public bool IsExpense => money < 0.0;

    public bool IsIncome => money > 0.0;

    public double Money => money;
}

```

Принцип использование триггера данных практически идентичен принципу триггера свойства. Для триггера

данных стоит указать привязку данных вместо названия свойства и значение с которым необходимо выполнять проверку на равенство.

Также как и с триггером свойства в WPF присутствует вариант триггера данных, который позволяет наблюдать сразу за несколькими свойствами (полный пример находится в папке `Wpf.Triggers.MultiDataTrigger`).

3.3. Примеры переопределения шаблонов

При переопределении шаблонов для некоторых простых элементов, таких как, например, кнопка или метка, единственным правилом является необходимость отображения содержимого в результирующей структуре элемента управления. Однако, существует много элементов управления, которые накладывают определенные требования на переопределенный шаблон. Например, элемент управления поле для ввода текста, должен содержать элемент, в который будет выводиться текст. Так как структура шаблона элемента управления может быть крайне сложной и содержать много компонентов, WPF необходимо будет идентифицировать, какой из них предназначен для отображения текста. Для подобных целей, некоторые элементы управления вводят концепцию именованных частей. По сути, при создании шаблона элемента управления требуется указать определенное имя (какое именно указывается в документации) одному из внутренних элементов. Во время применения шаблона к элементу управления, указанный именованный объект будет идентифицирован и будет использоваться автоматически.

3.3.1. Шаблон элемента управления *Button*

Элемент управления `System.Windows.Controls.Button` является одним из самых простых при создании нового шаблона элемента управления. Он не обладает ни одной обязательной именованной частью и, чтобы добиться адекватного его отображения, достаточно лишь учесть его содержимое, которое не известно на этапе описания шаблона. Эта задача решается крайне просто, применяя элемент `System.Windows.Controls.ContentPresenter`.

Приведенный ниже фрагмент разметки демонстрирует создание шаблона для элемента управления `System.Windows.Controls.Button` (полный пример находится в папке `Wpf.ControlTemplates.Controls.Button`):

XAML

```
<Window ...>
  <Window.Resources>
    <ControlTemplate x:Key="buttonTemplate"
                      TargetType="{x:Type Button}">
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="5"/>
          <ColumnDefinition/>
          <ColumnDefinition Width="5"/>
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
          <RowDefinition Height="5"/>
          <RowDefinition/>
          <RowDefinition Height="5"/>
        </Grid.RowDefinitions>

        <Border Background="{TemplateBinding
                                Background}"
```

```

        BorderBrush="{TemplateBinding
                        BorderBrush}"
        BorderThickness="1,1,1,0"
        Grid.Column="1"
        Grid.Row="0"/>

<Border Background="{TemplateBinding
                    Background}"
        BorderBrush="{TemplateBinding
                    BorderBrush}"
        BorderThickness="1,1,0,1"
        Grid.Column="0"
        Grid.Row="1"/>

<Border Background="{TemplateBinding
                    Background}"
        BorderBrush="{TemplateBinding
                    BorderBrush}"
        Grid.Column="1"
        Grid.Row="1">

<Grid>
    <Border Background=
            "{TemplateBinding
             BorderBrush}"
            Height="1"
            HorizontalAlignment=
                "Left"
            VerticalAlignment=
                "Top"
            Width="1"/>
    <Border Background=
            "{TemplateBinding
             BorderBrush}"
            Height="1"
            HorizontalAlignment=
                "Right"

```

```

        VerticalAlignment=
            "Top"
        Width="1"/>
<Border Background=
    "{TemplateBinding
        BorderBrush}"
    Height="1"
    HorizontalAlignment=
        "Left"
    VerticalAlignment=
        "Bottom"
    Width="1"/>
<Border Background=
    "{TemplateBinding
        BorderBrush}"
    Height="1"
    HorizontalAlignment=
        "Right"
    VerticalAlignment=
        "Bottom"
    Width="1"/>
</Grid>
</Border>

<Border Background= "{TemplateBinding
    Background}"
    BorderBrush= "{TemplateBinding
        BorderBrush}"
    BorderThickness= "0,1,1,1"
    Grid.Column="2"
    Grid.Row="1"/>
<Border Background= "{TemplateBinding
    Background}"
    BorderBrush= "{TemplateBinding
        BorderBrush}"
    BorderThickness="1,0,1,1"
    Grid.Column="1"

```

```

        Grid.Row="2"/>
    <ContentPresenter Content=
        "{TemplateBinding
        Content}"
        Grid.Column="1"
        Grid.Row="1"
        HorizontalAlignment=
            "{TemplateBinding
            HorizontalContentAlignment}"
        Margin= "{TemplateBinding
        Padding}"
        VerticalAlignment=
            "{TemplateBinding
            VerticalContentAlignment}"/>
</Grid>
</ControlTemplate>
<Style x:Key="buttonStyle"
    TargetType="{x:Type Button}">
    <Setter Property="FontWeight"
        Value="Bold"/>
    <Setter Property="Padding" Value="0"/>
    <Setter Property="Template"
        Value= "{StaticResource
        buttonTemplate}"/>
    <Setter Property="Typography.Capitals"
        Value="SmallCaps"/>
    <Style.Triggers>
        <Trigger Property="IsMouseOver"
            Value="True">
            <Setter Property="Foreground"
                Value="White"/>
        </Trigger>
    </Style.Triggers>
</Style>
<Style x:Key="acceptButtonStyle"
    BasedOn="{StaticResource buttonStyle}"
    TargetType="{x:Type Button}">

```

```

        <Setter Property= "Background"
            Value="#FF8FBC8F"/>
        <Setter Property="BorderBrush"
            Value="#FF556B2F"/>
        <Style.Triggers>
            <Trigger Property="IsMouseOver"
                Value="True">
                <Setter Property="Background"
                    Value="#FFA3D0A3"/>
            </Trigger>
            <Trigger Property="IsPressed"
                Value="True">
                <Setter Property="Background"
                    Value="#FF7BA87B"/>
            </Trigger>
        </Style.Triggers>
    </Style>

    <Style x:Key="declineButtonStyle"
        BasedOn="{StaticResource buttonStyle}"
        TargetType="{x:Type Button}">
        <Setter Property="Background"
            Value="#FFEB8080"/>
        <Setter Property="BorderBrush"
            Value="#FFB22222"/>
        <Style.Triggers>
            <Trigger Property="IsMouseOver"
                Value="True">
                <Setter Property="Background"
                    Value="#FFFF9494"/>
            </Trigger>
            <Trigger Property="IsPressed"
                Value="True">
                <Setter Property="Background"
                    Value="#FFD76C6C"/>
            </Trigger>
        </Style.Triggers>

```

```

</Style>

<Style x:Key="regularButtonStyle"
      BasedOn="{StaticResource buttonStyle}"
      TargetType="{x:Type Button}">
  <Setter Property="Background"
    Value="#FF87CEEB"/>
  <Setter Property="BorderBrush"
    Value="#FF191970"/>
  <Style.Triggers>
    <Trigger Property="IsMouseOver"
      Value="True">
      <Setter Property="Background"
        Value="#FF9BE2FF"/>
    </Trigger>
    <Trigger Property="IsPressed"
      Value="True">
      <Setter Property="Background"
        Value="#FF73BAD7"/>
    </Trigger>
  </Style.Triggers>
</Style>
</Window.Resources>

<Grid Margin="5">
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="5"/>
    <RowDefinition/>
    <RowDefinition Height="5"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Button Content="Regular Button"
    Grid.Row="0"
    Style="{StaticResource
      regularButtonStyle}"/>
  <Button Content="Accept Button"

```

```

        Grid.Row="2"
        Style="{StaticResource
            acceptButtonStyle}"/>
<Button Content="Decline Button"
        Grid.Row="4"
        Style= "{StaticResource
            declineButtonStyle}"/>

</Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 18.

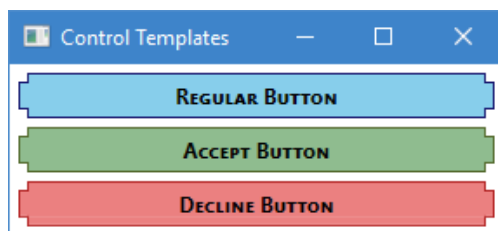


Рисунок 18. Кнопка

3.3.2. Шаблон элемента управления *TextBox*

Элемент управления `System.Windows.Controls.TextBox` отличается от рассмотренных ранее элементов управления тем, что он обладает именованной частью, которая отвечает за отображаемый текст. Для того, чтобы текст, ассоциируемый с полем для ввода текста отображался корректно в нужном месте, шаблон элемента управления должен содержать элемент `System.Windows.FrameworkElement`, обладающим именем `PART_ContentHost`.

Приведенный ниже фрагмент разметки демонстрирует создание шаблона для элемента управления `System.`

Windows.Controls.[TextBox](#) (полный пример находится в папке Wpf.ControlTemplates.Controls.TextBox):

XAML

```
<Window ...>
  <Window.Resources>
    <ControlTemplate x:Key="mailTextBoxTemplate"
      TargetType="{x:Type TextBox}">
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto"/>
          <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Border Background="{TemplateBinding
          BorderBrush}"
          BorderBrush="{TemplateBinding
            BorderBrush}"
          BorderThickness=
            "{TemplateBinding
              BorderThickness}"
          CornerRadius="3,0,0,3"
          Grid.Column="0">
          <Image Height="24"
            Margin="5"
            RenderOptions.
              BitmapScalingMode=
                "HighQuality"
            Source="/Icons/Mail.png"/>
        </Border>
        <Border BorderBrush=
          "{TemplateBinding
            BorderBrush}"
          BorderThickness=
            "{TemplateBinding
              BorderThickness}"
          CornerRadius="0,3,3,0"
          Grid.Column="1">
```

```

        <ScrollViewer x:Name=
            "PART_ContentHost"
            Margin="5,0"
            VerticalAlignment=
                "Center"/>
    </Border>
</Grid>
</ControlTemplate>

<ControlTemplate x:Key="phoneTextBoxTemplate"
    TargetType="{x:Type TextBox}">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Border Background= "{TemplateBinding
            BorderBrush}"
            BorderBrush="{TemplateBinding
                BorderBrush}"
            BorderThickness=
                "{TemplateBinding
                    BorderThickness}"
            CornerRadius="3,0,0,3"
            Grid.Column="0">
            <Image Height="24"
                Margin="5"
                RenderOptions.
                    BitmapScalingMode=
                        "HighQuality"
                Source="/Icons/Phone.png"/>
        </Border>
        <Border BorderBrush= "{TemplateBinding
            BorderBrush}"
            BorderThickness=
                "{TemplateBinding
                    BorderThickness}"

```

```

        CornerRadius="0,3,3,0"
        Grid.Column="1">
        <ScrollView x:Name=
            "PART_ContentHost"
            Margin="5,0"
            VerticalAlignment=
                "Center"/>
    </Border>
</Grid>
</ControlTemplate>

<Style x:Key="textBoxStyle"
    TargetType="{x:Type TextBox}">
    <Setter Property="BorderBrush"
        Value="Black"/>
    <Setter Property="BorderThickness"
        Value="3"/>
    <Setter Property="FontSize" Value="20"/>

    <Style.Triggers>
        <Trigger Property="IsKeyboardFocused"
            Value="True">
            <Setter Property="BorderBrush"
                Value="#FF1E90FF"/>
        </Trigger>
    </Style.Triggers>
</Style>

<Style x:Key="mailTextBoxStyle"
    BasedOn="{StaticResource
        textBoxStyle}"
    TargetType="{x:Type TextBox}">
    <Setter Property="Template"
        Value="{StaticResource
            mailTextBoxTemplate}"/>
</Style>

```

```

        <Style x:Key="phoneTextBoxStyle"
            BasedOn="{StaticResource textBoxStyle}"
            TargetType="{x:Type TextBox}">
            <Setter Property="Template"
                Value="{StaticResource
                    phoneTextBoxTemplate}"/>
        </Style>
    </Window.Resources>

    <Grid Margin="5">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition Height="5"/>
            <RowDefinition/>
        </Grid.RowDefinitions>

        <TextBox Grid.Row="0"
            Style="{StaticResource
                mailTextBoxStyle}"/>
        <TextBox Grid.Row="2"
            Style="{StaticResource
                phoneTextBoxStyle}"/>

    </Grid>
</Window>

```

Результат приведенной выше разметки показан на рис. 19.

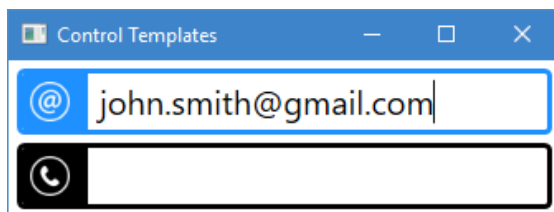


Рисунок 19. Поле для ввода текста

4. Домашнее задание

4.1. Задание 1

Необходимо разработать игру 2048 (рис. 20).

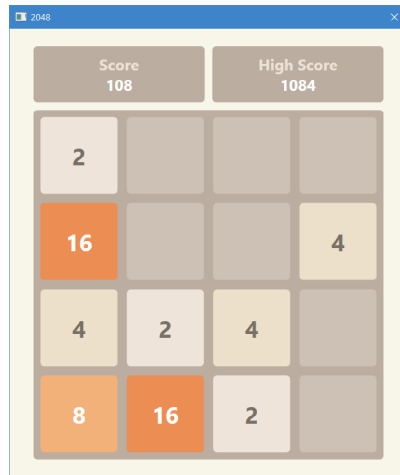


Рисунок 20

Игровое поле имеет форму квадрата 4x4. Целью игры является получение плитки номиналом 2048.

4.1.1. Правила игры

- В каждом раунде появляется плитка номиналом 2 (с вероятностью 90%) или 4 (с вероятностью 10%).
- Нажатием стрелки игрок может скинуть все плитки игрового поля в одну из 4-х сторон. Если при сбрасывании две плитки одного номинала «налетают» одна на другую, то они слипаются в одну, номинал которой равен сумме соединившихся плиток. После

каждого хода на свободной секции поля появляется новая плитка номиналом 2 или 4. Если при нажатии кнопки местоположение плиток или их номинал не изменится, то ход не совершается.

- Если в одной строчке или в одном столбце находится более двух плиток одного номинала, то при сбрасывании они начинают слипаться с той стороны, в которую были направлены. Например, находящиеся в одной строке плитки (4, 4, 4) после хода влево они превратятся в (8, 4), а после хода вправо — в (4, 8).
- За каждое соединение игровые очки увеличиваются на номинал получившейся плитки.
- Игра заканчивается поражением, если после очередного хода невозможно совершить действие.

При разработке приложения необходимо использовать архитектурный шаблон проектирования MVVM.

4.2. Задание 2

Необходимо разработать окно аутентификации пользователя (рис. 21). Для внешнего оформления элементов управления следует использовать стили и шаблоны элементов управления.

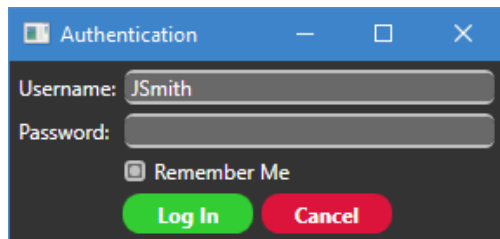


Рисунок 21

Урок № 5

Трансформации

© Павел Дубский

© Компьютерная Академия «Шаг».

www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.