34

Windows Presentation Foundation

Windows Presentation Foundation (WPF) — одно из трех главных расширений .NET Framework 3.0. WPF представляет собой новую библиотеку, служащую для создания пользовательских интерфейсов в интеллектуальных клиентских приложениях. В то время как элементы управления Windows Forms основаны на "родных" элементах управления Windows, использующих оконные дескрипторы (Window handles), которые основаны в свою очередь на экранных пикселях, WPF базируется на DirectX. Приложение теперь не использует оконные дескрипторы, поэтому размеры пользовательского интерфейса легко изменять, а, кроме того, здесь включена поддержка звука и видео.

Ниже перечислены основные темы, которые мы рассмотрим в настоящей главе:

- обзор WPF;
- фигуры, как базовые элементы рисования;
- масштабирование, вращение и наклоны с помощью трансформаций;
- различные виды кистей для заливки элементов;
- элементы управления WPF и их возможности;
- как определяются компоновки панелей WPF;
- механизм обработки событий WPF;
- стили, шаблоны и ресурсы;

Обзор

Одно из важнейших преимуществ WPF заключается в том, что работа легко может быть разделена между дизайнерами и разработчиками. Результат работы дизайнера может быть непосредственно использован разработчиком. Чтобы такое стало возможно, вы должны понимать XAML. Рассматривая первую тему настоящей главы, мы представим обзор WPF и дадим вам достаточно информации для понимания принципов XAML, а также о способах взаимодействия дизайнеров и разработчиков. WPF состоит из нескольких сборок, содержащих в себе тысячи классов. Чтобы вы могли легко ориентироваться в огромном множестве классов и легко находить то, что вам нужно, наш обзор включает иерархию классов и пространств имен WPF.

XAML

XAML (XML for Application Markup Language – XML для языка разметки приложений) представляет собой XML-синтаксис, используемый для определения иерархической структуры пользовательского интерфейса. В следующей строке вы можете видеть объявление кнопки по имени button1 с меткой “Click Me!”. Элемент `<Button>` специфицирует использование класса `Button`:

```
<Button Name="button1">Click Me!</Button> "
```

За элементом XAML всегда стоит класс .NET. С помощью атрибутов и дочерних элементов вы устанавливаете значения свойств и определяете методы-обработчики событий.

Для тестирования примера кода XAML вы можете запустить утилиту `XAMLPad.exe` (рис. 34.1) и ввести код XAML в поле редактирования. Вы можете вписать элемент `<Button>` внутрь элементов `<Page>` и `<Grid>`, которые уже подготовлены для `XAMLPad`. Благодаря `XAMLPad` вы можете видеть выходной результат XAML немедленно.

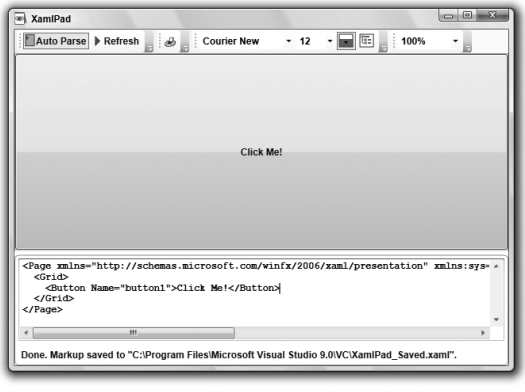

Рис. 34.1. Тестирование приложения в утилите `XAMLPad.exe`

Код XAML может быть интерпретирован исполняющей системой WPF, но также он может быть скомпилирован в BAML (Binary Application Markup Language – бинарный язык разметки приложений), что и делается по умолчанию в проектах Visual Studio WPF. BAML добавляется в исполняемый файл в виде ресурса.

Вместо написания XAML вы можете также создать кнопку в коде C#. Вы можете создать нормальное консольное приложение C#, добавить ссылки на сборки `WindowsBase`, `PresentationCore` и `PresentationFramework`, и написать следующий код. В методе `Main()` создается объект `Window` из пространства имен `System.Windows`, и устанавливается его свойство `Title`. Затем создается объект `Button` из пространства имен `System.Windows.Controls`, устанавливается его `Content`, а `Content` окна устанавливается в кнопку. Метод `Run()` класса `Application` отвечает за обработку сообщений Windows.

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace Wrox.ProCSharp.WPF
{
 class Program
 {
  [STAThread]
```

```
  static void Main()
  {
     Window mainWindow = new Window();
     mainWindow.Title = "WPF Application";
     Button button1 = new Button();
     button1.Content = "Click Me!";
     mainWindow.Content = button1;
     button1.Click +=
        (sender, e) => MessageBox.Show("Button clicked");
     Application app = new Application();
     app.Run(mainWindow);
  }
 }
}
```

Класс `Application` также может быть определен посредством XAML. WPF-проект Visual Studio включает файл `App.xaml`, содержащий в себе свойства и `StartupUri` класса `Application`.

Запустив приложение, вы получите окно, содержащее кнопку, как показано на рис. 34.2.

Рис. 34.2. Работа примера WPF-приложения

Как видите, программирование с использованием WPF очень похоже на программирование с применением Windows Forms, но с небольшим отличием, заключающимся в том, что `Button` имеет свойство `Content` вместо `Text`. Однако по сравнению с созданием форм пользовательского интерфейса в коде, XAML обладает некоторыми замечательными преимуществами. Благодаря XAML дизайнер и разработчик могут сотрудничать намного лучше. Дизайнер может работать в коде XAML и проектировать стильный пользовательский интерфейс, а разработчик — добавлять функциональность в отделенный код на C#. Благодаря XAML намного легче отделить пользовательский интерфейс от функциональности.

Вы можете непосредственно взаимодействовать с элементами, определенными в XAML из кода C#, используя отделенный код и XAML. Вам нужно только определить имя элемента и использовать то же имя как переменную для изменения свойств и вызова методов.

Кнопка имеет свойство `Content` вместо свойства `Text`, поскольку кнопка может показывать все что угодно. Вы можете добавлять к содержимому кнопки текст, но с тем же успехом и графику, и окно списка, и видеоклип — короче говоря, все, что пожелаете.

Свойства как атрибуты

Перед тем, как приступить к работе с XAML, вы должны знать важные характеристики синтаксиса XAML. Вы можете использовать атрибуты XML для специфицирования свойств классов. В следующем примере демонстрируется установка свойств `Content` и `Background` класса `Button`.

```
<Button Content="Click Me!" Background="LightGreen" />
```

Свойства как элементы

Вместо использования атрибутов XML свойства также могут специфицировать дочерние элементы. Значение содержимого может быть непосредственно установлено указанием дочернего элемента для `Button`. Для всех свойств `Button` имя дочернего элемента определяется именем внешнего элемента, следующего за именем свойства:

```
<Button>
<Button.Background>
  LightGreen
</Button.Background>
Click Me!
</Button>
```

В предыдущем примере не обязательно было применять дочерние элементы; используя атрибуты XML, можно достичь того же результата. Однако применение атрибутов теперь невозможно, если значение сложнее строки. Например, фон может быть установлен только в простой цвет, но также и задан кистью (brush); например, кистью с линейным градиентом.

```
<Button>
<Button.Background>
  <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
     <GradientStop Color="Yellow" Offset="0.0" />
     <GradientStop Color="Orange" Offset="0.25" />
     <GradientStop Color="Red" Offset="0.75" />
     <GradientStop Color="Violet" Offset="1.0" />
  </LinearGradientBrush>
</Button.Background>
Click Me!
</Button>
```

Свойство зависимости

Изучая программирование с использованием WPF, вы часто будете встречать термин *свойство зависимости* (dependency property). Элементы WPF представляют собой классы с методами, свойствами и событиями. Почти каждое свойство элемента WPF является свойством зависимости. Что это значит? Свойство зависимости может зависеть от других входных параметров, например, тем и пользовательских предпочтений. Свойства зависимости применяются со связыванием данных, анимацией, ресурсами и стилями.

С точки зрения грамматики свойство зависимости может быть прочитано и записано не только вызовом строго типизированного свойства, но также методов, получающих объект свойства зависимости.

Только класс, унаследованный от базового класса `DependencyObject`, может включать свойства зависимости. Следующий класс — `MyDependencyObject` — определяет свойство зависимости `SomeState`. `SomeStateProperty` — это статическое поле типа `DependencyProperty`, обеспечивающее свойство зависимости. Свойство зависимости зарегистрировано в системе свойств зависимости WPF с помощью метода `Register()`. Метод `Register()` получает имя свойства зависимости, его тип и тип его владельца. Вы можете устанавливать значение свойства зависимости посредством метода `SetValue()` базового класса `DependancyObject`, а получать значение методом `GetValue()`. Свойства зависимости обычно также имеет строго типизированный доступ. Вместо использования методов базового класса `DependancyObject` класс `MyDependancyObject` включает свойство `SomeState`, вызывающее методы базового класса из реализации средств доступа `set` и `get`. Вы не должны делать что-то еще в реализации средств доступа `set` и `get`, поскольку они могут и не вызываться для этого свойства.

```
public class MyDependencyObject : DependencyObject
{
 public static readonly DependencyProperty SomeStateProperty =
  DependencyProperty.Register("SomeState", typeof(String), typeof(MyDependencyObject));
 public string SomeState
 {
  get { return (string)this.GetValue(SomeStateProperty); }
  set { this.SetValue(SomeStateProperty, value); }
 }
}
```

В иерархии классов WPF класс `DependencyObject` находится очень высоко. Каждый элемент WPF наследуется от этого базового класса.

Присоединенное свойство

Элемент WPF также получает возможности родительского элемента. Например, если элемент `Button` находится внутри элемента `Canvas`, то кнопка имеет свойства `Top` и `Left`, снабженные префиксом — именем родительского элемента. Такое свойство называют *присоединенным (attached) свойством.*

```
<Canvas>
<Button Canvas.Top="30" Canvas.Left="40">
  Click Me!
</Button>
</Canvas>
```

Написание функциональности в отделенном коде слегка отличается, поскольку класс `Button` не имеет свойства `Canvas.Top` и `Canvas.Left`, даже если он содержится внутри класса `Canvas`.

Существует шаблон именования для установки присоединенных свойств, общий для всех классов. Класс, поддерживающий присоединенные свойства, имеет статические методы с именами `Set<Property>` и `Get<Property>`, где первый параметр — объект, к которому применено значение свойства. Класс `Canvas` определяет статические методы `SetLeft()` и `SetTop()` для получения того же результата, что и ранее показанный пример кода XAML.

```
[STAThread]
static void Main()
{
  Window mainWindow = new Window();
  Canvas canvas = new Canvas();
  mainWindow.Content = canvas;
  Button button1 = new Button();
  canvas.Children.Add(button1);
  button1.Content = "Click Me!";
  Canvas.SetLeft(button1, 40);
  Canvas.SetTop(button1, 30);
  Application app = new Application();
  app.Run(mainWindow);
}
```

Присоединенное свойство может быть реализовано как объект зависимости. Метод `DependencyProperty.RegisterAttached()` регистрирует присоединенное свойство.

Расширения разметки

При установке значений элементов вы можете делать это непосредственно, но иногда в этом очень пригодятся расширения разметки. Расширения разметки (markup extensions) состоят из фигурных скобок, внутри которых следует строковая лексема, определяющая тип расширения.

Ниже приведен пример расширения разметки StaticResource.

```
<Button Name="button1" Style="{StaticResource key}" Content="Click Me" />
```

Вместо использования расширения разметки вы можете написать то же самое через дочерние элементы:

```
<Button Name="button1">
<Button.Style>
     <StaticResource ResourceKey="key" />
</Button.Style>
Click Me!
</Button>
```

Расширения разметки в основном используются для обращения к ресурсам, необходимым для связывания данных. Обе эти темы мы обсудим ниже в настоящей главе.

Кооперация дизайнеров и разработчиков

Очень часто разработчики не только реализуют приложения Windows, но также отвечают и за дизайн. Это особенно верно в тех случаях, когда приложение создается только для внутреннего пользования. Если нанимается некто, обладающий квалификацией, необходимой для построения пользовательского интерфейса, то разработчик обычно получает файл JPEG, на котором представлено видение дизайнера того, как должен выглядеть пользовательский интерфейс.

Перед разработчиком затем встает проблема — воплотить этот дизайнерский план. Даже минимальные изменения, вносимые дизайнером, такие как другой внешний вид окон списков или кнопок, может привести к необходимости значительных затрат при использовании собственных рисуемых элементов управления. В результате пользовательский интерфейс, реализованный разработчиком, сильно отличается от того, который был изначально спроектирован.

Благодаря WPF эта ситуация меняется. Дизайнер и разработчик могут работать с одним и тем же кодом XAML. Дизайнер может использовать такой инструмент, как Expression Blend, в то время как разработчик — Visual Studio 2008. Оба работают над одними и теми же файлами проекта. В типичной схеме такого процесса кооперации дизайнер начинает проект в Expression Blend, используя те же файлы проекта, что и Visual Studio. Затем разработчик вступает в работу над отделенным кодом, в то время, пока дизайнер совершенствует пользовательский интерфейс. По мере того, как разработчик расширяет функциональность, дизайнер также может добавлять новые интерфейсные средства, использующие эту дополнительную функциональность, предоставленную разработчиком. Конечно, можно также запустить разработку приложения в Visual Studio и позднее усовершенствовать пользовательский интерфейс в EID. Единственное, о чем вам следует позаботиться — не разрабатывать интерфейс так, как вы привыкли делать это в Windows Forms, поскольку при этом не будут в полной мере использованы преимущества, представленные WPF.

На рис. 34.3 показано окно в Expression Blend, созданное с применением WPF. В этом приложении возможно масштабирование как рабочего пространства, так и документа, поскольку WPF базируется на векторной графике.

По сравнению с расширениями Expression Blend к Visual Studio, Expression Blend обладает великолепными средствами для определения стилей, создания анимаций, использования графики и т.п. Чтобы работать совместно, Expression Blend может использовать классы отделенного кода, созданные разработчиком, и дизайнер может специфицировать привязку данных от элементов WPF к классам .NET. Дизайнер может также протестировать завершенное приложение, начав с Expression Blend. Поскольку Expression Blend использует те же файлы MS-Build, что и Visual Studio, отделенного код C# компилируется для запуска приложения.

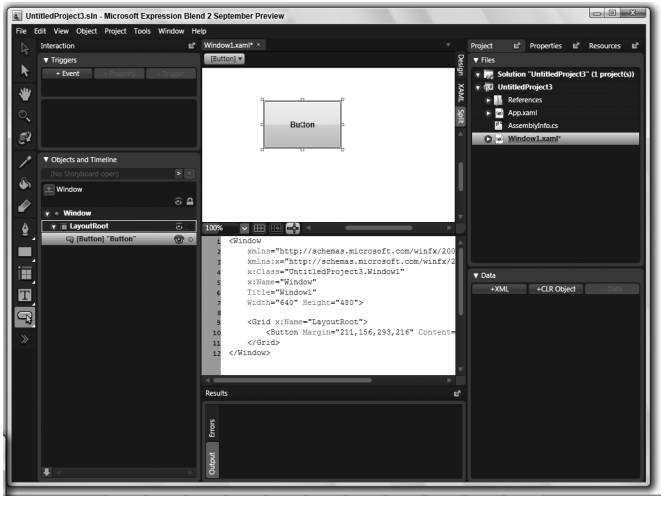

Рис. 34.3. Пример пользовательского интерфейса в Expression Blend

Иерархия классов

WPF состоит из тысяч классов, организованных в многоуровневую иерархию. Чтобы помочь понять отношения между этими классами, на рис. 34.4 показана часть диаграммы классов WPF. Некоторые классы и их функциональность описаны в табл. 34.1.

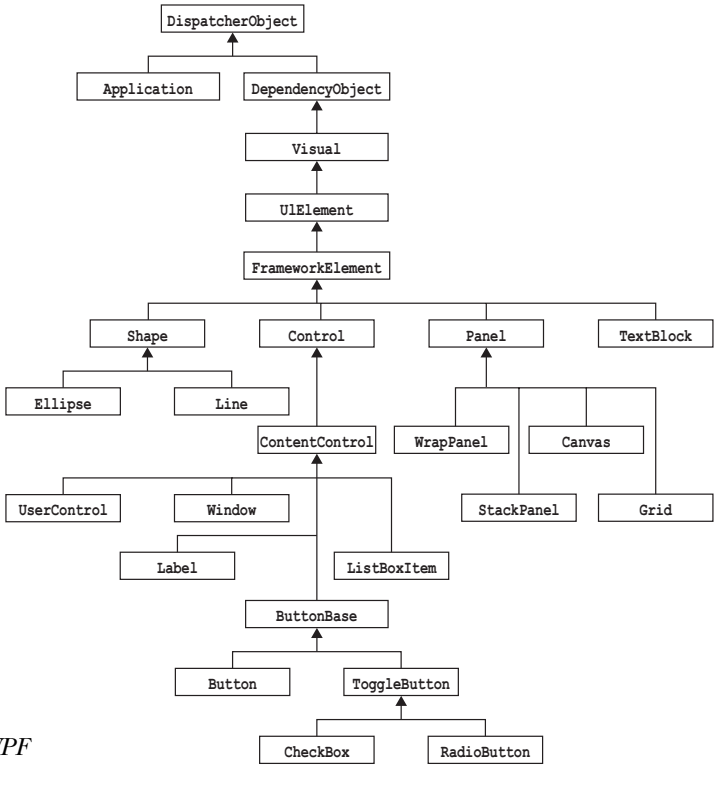

Рис. 34.4.
Часть диаграммы классов WPF

Таблица 34.1. Некоторые классы WPF

Класс	Описание
`DispatcherObject`	`DispatcherObject` — абстрактный базовый класс, являющийся предком всех классов, привязанных к одному потоку. WPF, подобно Windows Forms, требует, чтобы методы и свойства вызывались только из потока создателя. Классы, унаследованные от `DispatcherObject`, имеют ассоциированный объект `Dispatcher`, который может применяться для переключения потока.
`Application`	В приложении WPF может быть создан один экземпляр класса `Application`. Этот класс реализует шаблон Singleton для доступа к окнам приложения, его ресурсам и свойствам.
`DependencyObject`	`DependencyObject` — базовый класс для всех классов, поддерживающих свойства зависимости. Свойства зависимости мы обсуждали ранее.
`Visual`	`Visual` — базовый класс для всех визуальных элементов. Этот класс включает средства стрессового тестирования и трансформации.
`UIElement`	`UIElement` — абстрактный базовый класс для всех элементов WPF, которые нуждаются в базовых средствах представления `UIElement`. Этот класс обеспечивает туннелирование и пузырьковую передачу событий движения мыши, перетаскивания объектов и щелчков кнопками мыши, предоставляет виртуальные методы для визуализации, которые могут быть переопределены классами-наследниками, а также обеспечивает методы компоновки. Вы уже знаете, что WPF уже не использует оконные дескрипторы. Этот класс можно трактовать как эквивалент оконного дескриптора.
`FrameworkElement`	`FrameworkElement` унаследован от базового класса `UIElement` и реализует поведение по умолчанию методов, определенных в базовом классе.
`Shape`	`Shape` — базовый класс для всех элементов — фигур; например, `Line`, `Ellipse`, `Polygon`, `Rectangle`.
`Control`	`Control` наследуется от `FrameworkElement` и является базовым классом для всех интерактивных элементов, обеспечивающих взаимодействие с пользователем.
`Panel`	Класс `Panel` наследуется от `FrameworkElement` и является абстрактным базовым классом для всех панелей. Этот класс имеет свойство `Children` для всех элементов пользовательского интерфейса внутри панели и определяет методы для размещения дочерних элементов управления. Классы, унаследованные от `Panel`, определяют различное поведение организации дочерних элементов; например, `WrapPanel`, `StackPanel`, `Canvas`, `Grid`.
`ContentControl`	`ContentControl` — базовый класс для всех элементов управления, имеющих единственное содержимое (например, `Label`, `Button`). Стиль по умолчанию этого элемента управления может быть ограничен, но допускается изменение его внешнего вида посредством использования шаблонов.

Как видите, классы WPF действительно образуют глубокую иерархию. В настоящей и следующей главах вы увидите классы, составляющие лишь центральную функциональность, поскольку невозможно охватить все средства WPF в единственной главе.

Пространства имен

Классы из Windows Forms и WPF очень легко спутать. Классы Windows Forms находятся в пространстве имен `System.Windows.Forms`, в то время как классы WPF — внутри пространства имен `System.Windows` и вложенных в него пространств, за исклю-

чением System.Windows.Forms. Класс Button из Windows Forms имеет полное имя System.Windows.Forms.Button, а класс Button из WPF — System.Windows.Controls.Button. Тема Windows Forms раскрыта в главах 31 и 32.

Пространства имен WPF и их функциональность описаны в табл. 34.2.

Таблица 34.2. Пространства имен WPF

Пространство имен	Описание
System.Windows	Это центральное пространство имен WPF. Здесь вы найдете основные классы ядра WPF, такие как Application, классы объектов зависимости DependencyObject и DependencyProperty, а также базовый класс для всех элементов WPF — FrameworkElement.
System.Windows.Annotations	Классы из этого пространства имен применяются для пользовательских аннотаций и пометок (notes) данных приложения, которые хранятся отдельно от документа. Пространство имен System.Windows.Annotations.Storage содержит классы для хранения аннотаций.
System.Windows.Automation	Классы из этого пространства имен могут использоваться для автоматизации приложений WPF. Внутри него доступно несколько подпространств. System.Windows.Automation.Peers предоставляет элементы WPF для автоматизации — например, ButtonAutomationPeer и CheckBoxAutomationPeer. Пространство имен System.Windows.Automation.Provider необходимо, если вы создаете собственного поставщика автоматизации.
System.Windows.Controls	Это пространство имен, в котором вы можете найти все элементы управления WPF, такие как Button, Border, Canvas, ComboBox, Expander, Slider, ToolTip, TreeView и тому подобные. В пространстве имен System.Windows.Controls.Primitives вы найдете классы, используемые внутри сложных элементов управления, например, Popup, ScrollBar, StatusBar, TabPanel и другие.
System.Windows.Converters	Пространство имен System.Windows.Converters содержит классы, предназначенные для преобразования данных. Не ожидайте найти здесь все классы для преобразования. Основные классы, конвертирующие данные, определены в пространстве имен System.Windows.
System.Windows.Data	Пространство имен System.Windows.Data используется для привязки данных WPF. Важнейшим классом из этого пространства имен является Binding, служащий для определения привязки между целевым элементов WPF и источником CLR.
System.Windows.Documents	Если вы работаете с документами, то в этом пространстве имен найдете множество полезных классов. FixedDocument и FlowDocument — элементы содержимого, которые могут включать в себя другие элементы из этого пространства имен. С помощью классов из пространства имен System.Windows.Documents.Serialization вы можете записывать документы на диск.
System.Windows.Ink	Устройства вроде Windows Tablet PC и Ultra Mobile PC используются все шире и шире. На этих устройствах для организации пользовательского ввода могут применяться чернила. Пространство имен System.Windows.Ink содержит классы, имеющие дело с вводом подобного рода.

Окончание табл. 34.2

Пространство имен	Описание
`System.Windows.Input`	Пространство имен `System.Windows.Input` содержит несколько классов, предназначенных для обработки команд, клавиатурного ввода, работы с пером и тому подобным.
`System.Windows.Interop`	Классы, необходимые для интеграции Win32 и WPF вы найдете в пространстве имен `System.Windows.Interop`.
`System.Windows.Markup`	В этом пространстве имен содержатся вспомогательные классы для кода разметки XAML.
`System.Windows.Media`	Для работы с изображениями, аудио- и видеосодержимым вы можете использовать классы из пространства имен `System.Windows.Media`.
`System.Windows.Navigation`	Это пространство имен содержит классы, необходимые для навигации между окнами.
`System.Windows.Resources`	Это пространство имен включает классы, поддерживающие работу с ресурсами.
`System.Windows.Shapes`	Центральные классы пользовательского интерфейса расположены в пространстве имен `System.Windows.Shapes`: `Line`, `Ellipse`, `Rectangle` и другие.
`System.Windows.Threading`	Элементы WPF подобны элементам управления Windows Forms, привязанным к единственному потоку. В пространстве имен `System.Windows.Threading` вы найдете классы, предназначенные для работы с множеством потоков; например, в этом пространстве находится класс `Dispatcher`.
`System.Windows.Xps`	XML Paper Specification (XPS) — это новая спецификация документов, также поддерживаемая Microsoft Word. В пространствах имен `System.Windows.Xps.Packaging` и `System.Windows.Xps.Serialization` вы найдете классы, необходимые для создания потоковых документов XPS.

Фигуры

Фигуры (shapes) — центральные элементы WPF. С их помощью вы можете рисовать двумерную графику, используя прямоугольники, линии, эллипсы, многоугольники и ломаные — все эти фигуры представлены классами, унаследованными от абстрактного базового класса `Shape`. Фигуры определены в пространстве имен `System.Windows.Shapes`.

В следующем примере применения XAML рисуется рожица желтого цвета с ногами, состоящая из эллипса — лица, двух эллипсов — глаз, кривой линии рта и четырех прямых линий, изображающих ноги.

```
<Window x:Class="ProCSharp.WPF.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="WPF Samples" Height="260" Width="230">
 <Canvas>
     <Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
       Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
     <Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
       Stroke="Blue" StrokeThickness="3" Fill="White" />
     <Ellipse Canvas.Left="70" Canvas.Top="75" Width="5" Height="5" Fill="Black" />
```

```
    <Path Stroke="Blue" StrokeThickness="4" Data="M 62,125 Q 95,122 102,108" />
    <Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue" StrokeThickness="4" />
    <Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue" StrokeThickness="4" />
    <Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue" StrokeThickness="4" />
    <Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue" StrokeThickness="4" />
 </Canvas>
</Window>
```

Результат работы этого кода показан на рис. 34.5.

Все эти элементы WPF доступны программно, независимо от того — будь то кнопки или фигуры вроде линий и прямоугольников. Присваивание свойству `Name` элемента `Path` значения `mouth` (рот) позволяет вам получить программный доступ к данному элементу через переменную `mouth`:

```
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
  Data="M 62,125 Q 95,122 102,108" />
```

В отделенном коде для свойства `Data` элемента `Path` `mouth` устанавливается в новую геометрию. Чтобы задать форму кривой, класс `Path` поддерживает `PathGeometry` с синтаксисом разметки пути. Буква `M` определяет начальную точку пути, `Q` устанавливает контрольную точку и конечную точку для квадратичной кривой Безье. Запустив приложение, вы увидите окно в том виде, как оно показано на рис. 34.6.

```
public Window1()
{
  InitializeComponent();
  mouth.Data = Geometry.Parse("M 62,125 Q 95,122 102,128");
}
```

Ранее в этой главе вы узнали, что кнопка может иметь любое содержимое. Внеся небольшое изменение в код XAML, и добавив элемент `Button` в качестве содержимого окна, вы заставите графику отображаться внутри кнопки (рис. 34.7).

```
<Window x:Class="ShapesDemo.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ShapesDemo" Height="260" Width="230">
<Button Margin="5">
  <Canvas Height="260" Width="230">
     <Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
            Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
     <Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
            Stroke="Blue" StrokeThickness="3" Fill="White" />
```

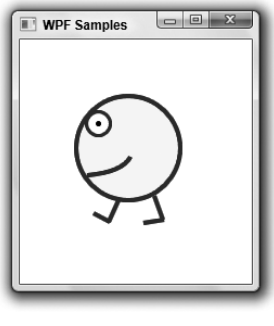

Рис. 34.5. Пример применения XAML

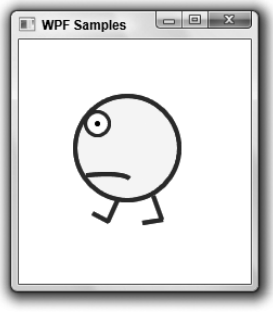

Рис. 34.6. Управление отдельным элементом

Рис. 34.7. Отображение графики внутри кнопки

```
      <Ellipse Canvas.Left="70" Canvas.Top="75" Width="5" Height="5" Fill="Black" />
      <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
            Data="M 62,125 Q 95,122 102,108" />
      <Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue" StrokeThickness="4" />
      <Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue" StrokeThickness="4" />
      <Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue" StrokeThickness="4" />
      <Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue" StrokeThickness="4" />
    </Canvas>
  </Button>
  </Window>
```

В табл. 34.3 перечислены фигуры, доступные в пространстве имен `System.Windows.Shapes`.

Таблица 34.3. Фигуры, доступные в пространстве имен `System.Windows.Shapes`

Класс фигуры	Описание
`Line`	Вы можете рисовать линию от координаты `X1.Y1` до `X2.Y2`.
`Rectangle`	С помощью класса `Rectangle` вы можете рисовать прямоугольник, задавая его ширину и высоту — `Width` и `Height`.
`Ellipse`	Этот класс позволит вам нарисовать эллипс.
`Path`	Класс `Path` можно использовать для рисования серии прямых и кривых линий. Свойство `Data` имеет тип `Geometry`. Вы можете выполнять рисование, используя классы, унаследованные от базового класса `Geometry`, либо применять синтаксис разметки пути для определения геометрии.
`Polygon`	Вы можете рисовать замкнутую фигуру, состоящую из соединенных линий посредством класса `Polygon`. Полигон определен серией объектов `Point`, присвоенной свойству `Points`.
`Polyline`	Подобно классу `Polygon`, вы можете рисовать соединенные линии с помощью `Polyline`. Отличие от полигона состоит в том, что ломаная не обязана формировать замкнутую фигуру.

Трансформация

Поскольку WPF основано на DirectX, который, в свою очередь, основан на векторной графике, вы можете масштабировать любой элемент. Векторная графика поддается масштабированию, вращению и сдвигу. Проверка попадания курсора мыши (hit testing) работает без необходимости в ручном вычислении позиции.

Добавление элемента `ScaleTransform` к свойству `LayoutTransform` элемента `Canvas`, как показано ниже, изменяет размер содержимого полного холста (canvas) вдвое в направлениях X и Y.

```
<Canvas.LayoutTransform>
  <ScaleTransform ScaleX="2" ScaleY="2" />
</Canvas.LayoutTransform>
```

Поворот может быть осуществлен способом, подобным масштабированию. Используя элемент `RotateTransform`, вы можете определить угол поворота (`Angle`):

```
<Canvas.LayoutTransform>
   <RotateTransform Angle="40" />
</Canvas.LayoutTransform>
```

Для скоса (skewing) вы можете применить элемент `SkewTransform`. При сдвиге вы можете назначать углы в направлениях X и Y.

```
<Canvas.LayoutTransform>
  <SkewTransform AngleX="20" AngleY="25" />
</Canvas.LayoutTransform>
```

На рис. 34.8 показывает результаты всех трансформаций. Рисунки размещены внутри `StackPanel`. Начиная слева, первая фигура масштабирована, вторая — повернута, а третья — сдвинута. Чтобы легче увидеть разницу, свойство `Background` элементов `Canvas` устанавливаются в разные цвета.

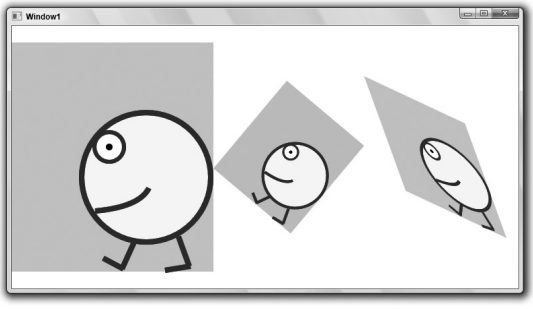

Рис. 34.8. Примеры различных трансформаций

Кисти

В этом разделе мы проиллюстрируем, как используются кисти (brushes), предоставленные WPF для закрашивания фона и переднего плана. На протяжении этого раздела мы будем ссылаться на рис. 34.9, который показывает эффекты от использования различных кистей для фона (`Background`) элементов `Button`.

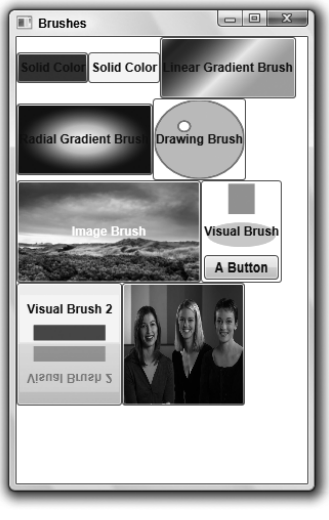

Рис. 34.9. Использование различных кистей для фона

SolidColorBrush

Первая кнопка на рис. 34.9 использует кисть `SolidColorBrush`, которая, как следует из ее названия, дает сплошной цвет. Вся область закрашена одним цветом.

Вы можете определить сплошной цвет, устанавливая атрибут `Background` в строку, определяющую этот самый сплошной цвет. Строка конвертируется в элемент `SolidColorBrush`.

```
<Button Height="30" Background="Purple">
Solid Color</Button>
```

Конечно, вы получите тот же эффект, устанавливая дочерний элемент `Background` и добавляя элемент `SolidColorBrush` в качестве его содержимого. Вторая кнопка приложения использует сплошной цвет `Yellow` (желтый) для фона:

```
<Button Height="30">
  <Button.Background>
    <SolidColorBrush>Yellow</SolidColorBrush>
  </Button.Background>
  Solid Color
</Button>
```

LinearGradientBrush

Для плавного изменения цвета вы можете использовать `LinearGradientBrush`, как показано в третьей кнопке. Эта кисть определяет свойства `StartPoint` и `EndPoint`. Этим свойствам вы можете присвоить двумерные координаты, чтобы задать линейный градиент. По умолчанию принят диагональный линейный градиент от точки 0,0 до 1,1. Определяя разные значения, градиент может принимать разные направления. Например, при `StartPoint` 0,0 и `EndPoint` 0,1 вы получите вертикальный градиент. При том же `StartPoint` и значении `EndPoint` 1,0 получается горизонтальный градиент.

В содержимом этой кисти вы можете определять значения цветов в указанных смещениях посредством элемента `GradientStop`. Между двумя стопами выполняется плавный переход цвета.

```
<Button Height="60">
  <Button.Background>
    <LinearGradientBrush StartPoint="0,0"
       EndPoint="0.5,1.2">
      <GradientStop Color="Red" Offset="0"></GradientStop>
      <GradientStop Color="Blue" Offset="0.2">
      </GradientStop>
      <GradientStop Color="BlanchedAlmond" Offset="0.7">
      </GradientStop>
      <GradientStop Color="DarkOrange" Offset="1">
      </GradientStop>
    </LinearGradientBrush>
  </Button.Background>
  Linear Gradient Brush
</Button>
```

RadialGradientBrush

С помощью `RadialGradientBrush` вы можете задать переход цвета в радиальном направлении. На рис. 34.9 четвертая кнопка использует `RadialGradientBrush`. Эта кисть определяет начальную точку цвета в свойстве `GradientOrigin`.

```
<Button Height="70" >
  <Button.Background>
    <RadialGradientBrush Center="0.5,0.5"
       GradientOrigin="0.5,0.5"
       RadiusX="0.5" RadiusY="0.5" SpreadMethod="Pad">
      <GradientStop Color="White" Offset="0" />
      <GradientStop Color="LightBlue" Offset="0.4" />
      <GradientStop Color="DarkBlue" Offset="1" />
    </RadialGradientBrush>
  </Button.Background>
  Radial Gradient Brush
</Button>
```

DrawingBrush

`DrawingBrush` позволяет вам определить рисунок, отображаемый кистью. Этот рисунок определен внутри элемента `GeometryDrawing`. Элемент `GeometryGroup`, который вы можете видеть внутри свойства `Geometry`, состоит из элементов `Geometry`, таких как `EllipseGeometry`, `LineGeometry`, `RectangleGeometry` и `CombinedGeometry`.

```
<Button Height="80">
  <Button.Background>
    <DrawingBrush>
      <DrawingBrush.Drawing>
```

```
        <GeometryDrawing Brush="LightBlue">
          <GeometryDrawing.Geometry>
            <GeometryGroup>
              <EllipseGeometry RadiusX="30" RadiusY="30"
                  Center="20,20" />
              <EllipseGeometry RadiusX="4" RadiusY="4"
                  Center="10,10" />
            </GeometryGroup>
          </GeometryDrawing.Geometry>
          <GeometryDrawing.Pen>
            <Pen>
              <Pen.Brush>Red
              </Pen.Brush>
            </Pen>
          </GeometryDrawing.Pen>
        </GeometryDrawing>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Button.Background>
  Drawing Brush
</Button>
```

ImageBrush

Чтобы загрузить графическое изображение в кисть, вы можете использовать элемент `ImageBrush`. С помощью этого элемента будет отображен образ, определенный свойством `ImageSource`.

```
<Button Height="100">
  <Button.Background>
    <ImageBrush
      ImageSource=" C:\Windows\Web\Wallpaper\img21.bmp"
    />
  </Button.Background>
  <Button.Foreground>White</Button.Foreground>
  Image Brush
</Button>
```

VisualBrush

`VisualBrush` позволяет использовать в кисти другие элементы WPF. Здесь вы можете добавить элемент WPF к свойству Visual. Седьмая кнопка на рис. 34.9 содержит в себе элементы `Rectangle`, `Ellipse` и `Button`.

```
<Button Height="100">
  <Button.Background>
    <VisualBrush >
      <VisualBrush.Visual>
        <StackPanel Background="White">
          <Rectangle Width="25" Height="25"
              Fill="LightCoral" Margin="2" />
          <Ellipse Width="65" Height="20"
              Fill="Aqua" Margin="5" />
          <Button Margin="2">A Button</Button>
        </StackPanel>
      </VisualBrush.Visual>
    </VisualBrush>
  </Button.Background>
  Visual Brush
</Button>
```

С помощью VisualBrush вы можете также создавать такие эффекты, как отражение. Показанная здесь кнопка содержит StackPanel, которая, в свою очередь, содержит Border и Rectangle. Элемент Border содержит StackPanel с Label и Rectangle. Но это не все. Второй Rectangle заполняется посредством VisualBrush. Кисть определяет значение размытости и трансформацию. Свойство Visual привязано к элементу Border. Трансформация выполняется установкой свойства RelativeTransform кисти VisualBrush. Эта трансформация использует относительные координаты. За счет установки ScaleY в -1 выполняется отражение по оси Y. TranslateTransform перемещает трансформацию по направлении Y, так что отражение оказывается под исходным объектом. Вы можете видеть результат на восьмой кнопке (“Visual Brush 2”) на рис. 34.9.

Привязка данных и элемент Binding, используемые здесь, подробно объясняются в следующей главе.

```
<Button Height="120">
  <StackPanel>
    <Border x:Name="reflected">
      <Border.Background>Yellow</Border.Background>
      <StackPanel>
        <Label>Visual Brush 2</Label>
        <Rectangle Width="70" Height="15" Margin="2"
             Fill="BlueViolet" />
      </StackPanel>
    </Border>
    <Rectangle Height="30">
      <Rectangle.Fill>
        <VisualBrush Opacity="0.35" Stretch="None"
             Visual="{Binding ElementName=reflected}">
          <VisualBrush.RelativeTransform>
            <TransformGroup>
              <ScaleTransform ScaleX="1" ScaleY="-1"
              />
              <TranslateTransform Y="1" />
            </TransformGroup>
          </VisualBrush.RelativeTransform>
        </VisualBrush>
      </Rectangle.Fill>
    </Rectangle>
  </StackPanel>
</Button>
```

Также вы можете использовать VisualBrush для воспроизведения видео — просто устанавливая свойство Visual в MediaElement. Свойство Source в MediaControl устанавливается в файл WMV. На рис. 34.9 девятая кнопка показывает пример отображения видео. Вы можете попробовать свое собственное, и если у вас версия Ultimate системы Windows Vista, то вы найдете то же видео на жестком диске. В противном случае просто выберите другой видеофайл.

```
<Button Height="120">
  <Button.Background>
    <VisualBrush>
      <VisualBrush.Visual>
      <MediaElement x:Name="video"
          Source="C:\Windows\ehome\ColorTint.wmv" />
      </VisualBrush.Visual>
    </VisualBrush>
  </Button.Background>
</Button>
```

Элементы управления

В WPF вам доступны сотни различных элементов управления. Для того чтобы лучше понять их, эти элементы управления можно разбить на следующие категории:

- простые элементы управления;
- элементы управления с содержимым;
- озаглавленные элементы управления с содержимым;
- многоэлементные элементы управления;
- озаглавленные многоэлементные элементы управления.

Простые элементы управления

Простые элементы управления — это те, которые не имеют содержимого, т.е. свойства `Content`. На примере класса `Button` вы видели, что `Button` может содержать любую фигуру или элемент, который вам нравится. Такое невозможно для простых элементов управления. В табл. 34.4 перечислены простые элементы управления и их функциональность.

Таблица 34.4. Простые элементы управления

Элемент управления	Описание
`PasswordBox`	Элемент управления `PasswordBox` используется для ввода паролей. Этот элемент имеет специфические свойства для такого ввода; например, `PasswordChar` — для определения символа, который должен отображаться по мере ввода пользователем пароля, или `Password` — для доступа к введенному паролю. Событие `PasswordChanged` вызывается, как только пароль изменяется.
`ScrollBar`	`ScrollBar` (линейка прокрутки) — это элемент управления, содержащий в себе `Thumb` (бегунок), где пользователь может выбирать значение. Линейка прокрутки может использоваться, например, если документ не умещается на экране. Некоторые более сложные элементы управления содержат линейки прокрутки, чтобы можно было просматривать не умещающееся в них содержимое.
`ProgressBar`	Используя элемент управления `ProgressBar`, вы можете отображать ход долго выполняющейся операции.
`Slider`	С помощью элемента управления `Slider` пользователь может выбирать диапазон значений, перемещая `Thumb`. Классы `ScrollBar`, `ProgressBar` и `Slider` наследуются от одного базового класса — `RangeBase`.
`TextBox`	Элемент управления `TextBox` используется для отображения простого неформатированного текста.
`RichTextBox`	Элемент управления `RichTextBox` поддерживает форматированный текст с помощью класса `FlowDocument`. Классы `RichTextBox` и `TextBox` наследуются от одного базового класса — `TextBoxBase`.

Хотя простые элементы управления не имеют свойства `Content`, вы в полной мере можете управлять их внешним видом, определяя шаблоны. О шаблонах речь пойдет далее в главе.

Элементы управления с содержимым

Класс ContentProperty имеет свойство Content. Через это свойство вы можете добавлять любое содержимое к элементу управления. В предыдущем примере вы видели элемент Canvas с вставленным в него Button. Элементы управления с содержимым описаны в табл. 34.5.

Таблица 34.5. Элементы управления с содержимым

Элемент управления	Описание
Button RepeatButton ToggleButton CheckBox RadioButton	Классы Button, RepeatButton, ToggleButton и GridViewColumnHeader унаследованы от одного базового класса — ButtonBase. Все кнопки реагируют на событие Click. RepeatButton возбуждает событие Click многократно — до тех пор, пока кнопка не будет отпущена. ToggleButton — базовый класс для CheckBox и RadioButton. Эти кнопки имеют состояние “включено” и “выключено”. CheckBox может быть помечен и очищен пользователем; RadioButton может быть помечен (выбран) пользователем. Очистка RadioButton должна выполняться программно.
Label	Класс Label представляет текстовую метку для элемента управления. Этот класс также имеет поддержку клавиш доступа, например, команд меню.
Frame	Элемент управления Frame поддерживает навигацию. Вы можете перейти на содержимое страницы посредством метода Navigate(). Если содержимое представляет собой Web-страницу, для отображения используется элемент управления — браузер.
ListBoxItem	ListBoxItem — элемент внутри ListBox (окна списка).
StatusBarItem	StatusBarItem — элемент внутри StatusBar (панели состояния).
ScrollViewer	ScrollViewer — элемент управления с содержимым, включающий линейки прокрутки. Вы можете поместить любое содержимое в этот элемент; при необходимости появятся линейки прокрутки.
ToolTip	ToolTip создает всплывающее окно для отображения дополнительной информации для другого элемента управления.
UserControl	Использование класса UserControl в качестве базового класса обеспечивает простой путь для создания собственных специализированных элементов управления. Однако этот класс не имеет поддержки шаблонов.
Window	Класс Window позволяет вам создавать обычные и диалоговые окна. С помощью класса Window вы получаете рамку с кнопками для минимизации/максимизации/закрытия и системное меню. Для отображения диалогового окна вы можете использовать метод ShowDialog(); метод Show() открывает окно.
NavigationWindow	Класс NavigationWindow унаследован от класса Window и поддерживает навигацию по содержимому.

В следующем коде XAML внутри Window содержится только элемент управления Frame. Свойство Source установлено в http://www.wrox.com, так что элемент управления Frame выполняет переход на этот Web-сайт (рис. 34.10).

```
<Window x:Class="FrameSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="FrameSample" Height="400" Width="400">
<Frame Source="http://www.wrox.com" />
</Window>
```

Рис. 34.10. Работа элемента управления `Frame`

Элементы управления с содержимым и заголовками

Элементы управления с содержимым и заголовком унаследованы от класса `HeaderedContentControl`. Сам `HeaderedContentControl` наследуется от базового класса `ContentControl`. Класс `HeaderedContentControl` имеет свойство `Header` для определения заголовка и `HeaderTemplate` для полной настройки заголовка. Элементы управления, унаследованные от базового класса `HeaderedContentControl`, перечислены в табл. 34.6.

Таблица 34.6. Элементы управления, унаследованные от `HeaderedContentControl`

Элемент управления	Описание
`Expander`	С помощью элемента управления `Expander` можно организовать "дополнительный" (advanced) режим в диалоговом окне, которое по умолчанию не показывает всю информацию, но может быть развернуто пользователем для ее отображения. В неразвернутом режиме показана информация заголовка, а в развернутом — само содержимое.
`GroupBox`	Элемент управления `GroupBox` представляет рамку и заголовок для группы элементов управления.
`TabItem`	Элемент управления `TabItem` — это элементы внутри класса `TabControl`. Свойство `Header` в `TabItem` представляет содержимое заголовка, отображаемое во вкладках `TabControl`.

Ниже приведен пример использования элемента управления `Expander`. Элемент `Expander` имеет свойство `Header`, установленное в "Click for more". Этот текст отображается для развертывания. Содержимое элемента управления отображается только в том случае, если он развернут. На рис. 34.11 показано простое приложение со свернутым элементом управления `Expander`. На рис. 34.12 показано то же приложение с развернутым элементом управления `Expander`.

Рис. 34.11. Элемент управления `Expander` в свернутом состоянии

Рис. 34.12. Элемент управления `Expander` в развернутом состоянии

```
<Window x:Class="ExpanderSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Expander Sample" Height="300" Width="300">
 <StackPanel>
     <TextBlock>Short information</TextBlock>
     <Expander Header="Click for more">
       <Border Height="200" Width="200" Background="Yellow">
         <TextBlock HorizontalAlignment="Center"
              VerticalAlignment="Center">
              More information here!
         </TextBlock>
       </Border>
     </Expander>
 </StackPanel>
</Window>
```

Если вы хотите изменять заголовок элемента управления `Expander` в зависимости от того, развернуто или свернуто содержимое, вы можете создать триггер. О триггерах мы еще поговорим в этой главе.

Многоэлементные элементы управления

Класс `ItemsControl` содержит список элементов, доступ к которым можно получить через свойство `Items`. Классы, унаследованные от `ItemsControl`, перечислены в табл. 34.7.

Таблица 34.7. Элементы управления, унаследованные от `ItemsControl`

Элемент управления	Описание
`Menu` `ContextMenu`	Классы `Menu` и `ContextMenu` наследуются от абстрактного базового класса `MenuBase`. Предоставить меню пользователю можно, поместив элементы `MenuItem` в список и связав с ними соответствующие команды.
`StatusBar`	Элемент `StatusBar` обычно используется для отображения в нижней части окна приложения, предоставляя пользователю информацию о состоянии. Вы можете поместить элементы `StatusBarItem` внутрь списка `StatusBar`.

Окончание табл. 34.7

Элемент управления	Описание
`TreeView`	Элемент управления `TreeView` служит для отображения иерархических структур элементов.

`ListBox`
`ComboBox`
`TabControl`
	`ComboBox`, `ListBox` и `TabControl` имеют один и тот же абстрактный базовый класс — `Selector`. Этот базовый класс позволяет выбирать элементы из списка. `ListBox` отображает элементы списка. `ComboBox` имеет дополнительный элемент управления `Button` для отображения элементов, только если был совершен щелчок на кнопке. С помощью `TabControl` содержимое представляется в табличной форме.

Многоэлементные элементы управления с заголовками

`HeaderedItemsControls` является базовым классом для элементов управления, которые содержат не только элементы, но также и заголовок. Этот класс унаследован от `ItemsControls`.

Классы, унаследованные от `HeaderedItemsControls`, описаны в табл. 34.8.

Таблица 34.8. Элементы управления, унаследованные от `HeaderedItemsControls`

Элемент управления	Описание
`MenuItem`	Классы меню `Menu` и `ContextMenu` включают элементы типа `MenuItem`. Элементы меню могут быть ассоциированы с командами, поскольку класс `MenuItem` реализует интерфейс `ICommandSource`.
`TreeViewItem`	Класс `TreeView` может включать элементы типа `TreeViewItem`.
`ToolBar`	Элемент управления `ToolBar` — это контейнер для группы элементов управления, обычно включающий элементы `Button` и `Separator`. Вы можете поместить `ToolBar` внутри `ToolBarTray`, который обрабатывает переупорядочивание элементов управления `ToolBar`.

Компоновка

Для определения компоновки элементов, используемой приложением, вы можете применять класс, унаследованный от базового класса `Panel`. Доступно несколько контейнеров компоновки, о которых мы поговорим ниже. Контейнер компоновки должен выполнять две главные задачи: измерение и расстановка. При измерении контейнер опрашивает у своих дочерних элементов их предпочтительные размеры. Поскольку полный размер, который сообщают эти элементы, может оказаться недоступным, затем контейнер определяет размеры и расположение этих дочерних элементов.

`StackPanel`

`Window` может включать в себя в качестве содержимого только один элемент. Если вам нужно поместить в окно более одного элемента, вы можете использовать `StackPanel` как дочерний элемент `Window` и добавлять необходимые вам элементы уже внутрь `StackPanel`.

StackPanel представляет собой простой элемент-контейнер, который просто отображает содержащиеся в нем элементы один за другим. Ориентация StackPanel может быть горизонтальной или вертикальной. Класс ToolBarPanel унаследован от StackPanel.

```
<Window x:Class="LayoutSamples.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Layout Samples" Height="300" Width="283">
 <StackPanel>
     <Label>Label</Label>
     <TextBlock>TextBlock</TextBlock>
     <TextBox>TextBox</TextBox>
     <Button>Button</Button>
     <CheckBox>Checkbox</CheckBox>
     <ListBox>
       <ListBoxItem>ListBoxItem One</ListBoxItem>
       <ListBoxItem>ListBoxItem Two</ListBoxItem>
     </ListBox>
     <Button>Button</Button>
 </StackPanel>
</Window>
```

На рис. 34.13 вы можете видеть дочерние элементы управления StackPanel, организованные вертикально.

> В случае привязываемых к данным элементов StackPanel, если недостаточно пространства для отображения их всех, то вы можете использовать вместо нее VirtualizingStackPanel. В этой панели генерируются только отображаемые элементы.

WrapPanel

WrapPanel позиционирует дочерние элементы слева направо, друг за другом, до тех пор, пока они умещаются в одну строку, а затем продолжает со следующей строки. Ориентация панели может быть как горизонтальной, так и вертикальной.

```
<Window x:Class="LayoutSamples.WrapPanelDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Layout Samples" Height="160" Width="250">
 <WrapPanel>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
  <Button Width="100">Button</Button>
 </WrapPanel>
</Window>
```

На рис. 34.14 показан вывод панели WrapPanel. Если вы измените размер окна приложения, кнопки будут заново упорядочены так, чтобы умещаться в строке.

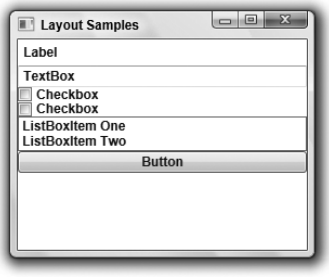

Рис. 34.13. Дочерние элементы управления панели `StackPanel`

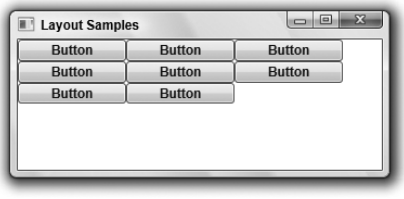

Рис. 34.14. Дочерние элементы управления панели `WrapPanel`

Canvas

`Canvas` — это панель, которая позволяет явно позиционировать элементы управления. `Canvas` определяет присоединенные свойства `Left`, `Right`, `Top` и `Bottom`, которые могут использоваться дочерними элементами для позиционирования внутри панели.

```
<Window x:Class="LayoutSamples.CanvasDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Layout Samples" Height="300" Width="300">
 <Canvas Background="LightBlue">
  <Label Canvas.Top="30" Canvas.Left="20">Enter here:</Label>
  <TextBox Canvas.Top="30" Canvas.Left="130" Width="100"></TextBox>
  <Button Canvas.Top="70" Canvas.Left="130">Click Me!</Button>
 </Canvas>
</Window>
```

На рис. 34.15 показан вывод панели `Canvas` с позиционированными дочерними элементами `Label`, `TextBox` и `Button`.

DockPanel

`DockPanel` очень похожа на функциональность стыковки Windows Forms. Здесь вы можете специфицировать область, где должны быть организованы дочерние элементы управления. `DockPanel` определяет присоединенное свойство `Dock`, которое вы можете установить для дочерних элементов в значения `Left`, `Right`, `Top` и `Bottom`. На рис. 34.16 показан результат размещения текстовых блоков в панели `DockPanel`.

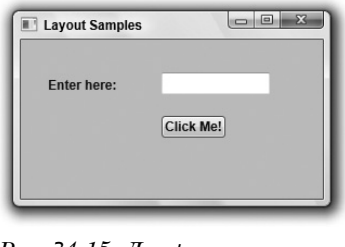

Рис. 34.15. Дочерние элементы управления панели `Canvas`

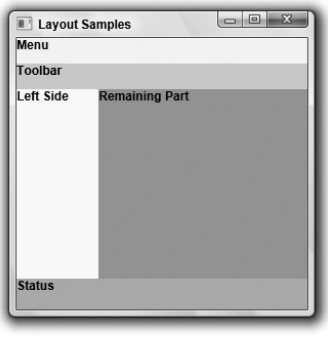

Рис. 34.16. Дочерние элементы управления панели `DockPanel`

Для облегчения восприятия различий разные области выделены разными цветами.

```
<Window x:Class="LayoutSamples.DockPanelDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Layout Samples" Height="300" Width="300">
 <DockPanel Background="LightBlue">
  <Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">
     <TextBlock>Menu</TextBlock>
  </Border>
  <Border Height="25" Background="Aqua" DockPanel.Dock="Top">
     <TextBlock>Toolbar</TextBlock>
  </Border>
  <Border Height="30" Background="LightSteelBlue" DockPanel.Dock="Bottom">
     <TextBlock>Status</TextBlock>
  </Border>
  <Border Width="80" Background="Azure" DockPanel.Dock="Left">
     <TextBlock>Left Side</TextBlock>
  </Border>
  <Border Background="HotPink">
     <TextBlock>Remaining Part</TextBlock>
  </Border>
 </DockPanel>
</Window>
```

Grid

Используя `Grid`, вы можете упорядочить элементы управления по строкам и столбцам. Для каждого столбца вы можете специфицировать `ColumnDefinition`, а для каждой строки — `RowDefinition`. В примере кода отображаются два столбца и три строки. Для каждой строки и столбца вы можете специфицировать ширину и высоту. `ColumnDefinition` имеет свойство `Width`, а `RowDefinition` — свойство `Height`. Высоту и ширину можно определять в пикселях, сантиметрах, дюймах или точках, либо установить `Auto` для определения размеров в зависимости от содержимого. `Grid` также позволяет "звездное" изменение размера (star sizing), когда пространство для строк и столбцов вычисляется в соответствии с доступным пространством и относительно других строк и столбцов. При выделении доступного пространства для столбца вы можете установить свойство `Width` в значение `*`, а чтобы получить удвоенное значение для другого столбца, указывайте `2*`. Пример кода, определяющего два столбца и три строки, не задает дополнительных установок в определениях столбца и строки; по умолчанию работает "звездная" настройка.

`Grid` содержит несколько элементов `Label` и `TextBox`. Поскольку родителем этих элементов является `Grid`, вы можете установить присоединенные свойства `Column`, `ColumnSpan`, `Row` и `RowSpan`.

```
<Window x:Class="LayoutSamples.GridDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Layout Samples" Height="300" Width="283">
 <Grid ShowGridLines="True">
  <Grid.ColumnDefinitions>
     <ColumnDefinition />
     <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
     <RowDefinition />
     <RowDefinition />
     <RowDefinition />
  </Grid.RowDefinitions>
```

```
  <Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0"
       VerticalAlignment="Center" HorizontalAlignment="Center">Title</Label>
  <Label Grid.Column="0" Grid.Row ="1" VerticalAlignment="Center">
       Firstname:</Label>
  <TextBox Grid.Column="1" Grid.Row="1" Width="100" Height="30"></TextBox>
  <Label Grid.Column="0" Grid.Row ="2" VerticalAlignment="Center">
       Lastname:</Label>
  <TextBox Grid.Column="1" Grid.Row="2" Width="100" Height="30"></TextBox>
 </Grid>
</Window>
```

В итоге получим упорядоченные в сетку элементы управления, как показано на рис. 34.17. Для того чтобы были видны столбцы и строки, свойство `ShowGridLines` установлено в `true`.

Для `Grid` с одинаковыми ячейками вы можете использовать класс `UniformGrid`.

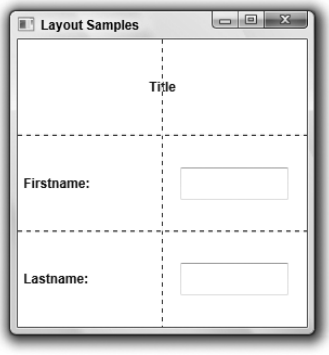

Рис. 34.17. Дочерние элементы управления панели `Grid`

Обработка событий

Классы WPF определяют события, для которых вы можете добавить собственные обработчики, например, `MouseEnter`, `MouseLeave`, `MouseMove`, `Click` и тому подобные. Все это базируется на механизме событий и делегатов .NET. Описание архитектуры событий и делегатов вы найдете в главе 7.

В WPF вы можете присвоить обработчик событий либо в XAML, либо в отделенном коде. Для `button1` назначен XML-атрибут `Click`, используемый для присваивания метода `OnButtonClick` событию щелчка на кнопке. Для `button2` никакого обработчика событий в XAML не назначено.

```
<Button Name="button1" Click="OnButtonClick">Button 1</Button>
<Button Name="button2" Button 2</Button>
```

Событие `Click` для `button2` присваивается в отделенном коде посредством создания экземпляра делегата `RoutedEventHandler` и передачи метода `OnButtonClick` делегату. Метод `OnButtonClick()`, вызываемый из обеих кнопок, имеет аргументы, определенные делегатом `RoutedEventHandler`.

```
public Window1()
{
  InitializeComponent();
  button2.Click += new RoutedEventHandler(OnButtonClick);
}
void OnButtonClick(object sender, RoutedEventArgs e)
{
  MessageBox.Show("Click Event");
}
```

Механизм обработки событий для WPF базируется на событиях .NET, но расширенный средствами "пузырькового" распространения и туннелирования. Как вы уже знаете, `Button` может содержать графику, окна списков, другую кнопку и т.д. Что случится, если внутри `Button` будет элемент `CheckBox`, и вы щелкните на этом `CheckBox`? Где возникнет событие? Ответ заключается в том, что событие распространяется подобно пузырьку. Сначала событие `Click` возникает в `CheckBox`, а затем распространяется к `Button`. Таким образом, вы можете обработать событие `Click` во всех элементах, находящихся внутри `Button`, наряду с самим `Button`.

Некоторые события являются туннелируемыми, другие пузырьковыми. Пузырьковые события начинаются с внутреннего элемента и распространяются к внешним. Обычно туннелируемые и пузырьковые события идут парами. Туннелируемые события снабжены префиксом `Preview`, например, `PreviewMouseMove`. Это событие распространяется от внешнего элемента управления к внутренним. После события `PreviewMouseMove` возникает событие `MouseMove`. Это событие распространяется, как пузырек — от внутренних элементов управления к внешним.

Вы можете остановить туннелирование или пузырьковое распространение, задав свойству `Handled` аргумента события значение `true`. Свойство `Handled` — член класса `RoutedEventArgs`, и все обработчики событий, участвующие в процессе туннелирования или пузырькового распространения, имеют аргумент типа `RoutedEventArgs` или наследника `RoutedEventArgs`.

Если вы останавливаете туннелирование события, установив свойство `Handled` в `true`, то пузырьковое событие, следующее за туннельным, уже не возникает.

Стили, шаблоны и ресурсы

Вы можете определить вид и поведение элементов WPF, устанавливая такие свойства, как `FontSize` и `BackGround` для элемента `Button`, как показано ниже:

```
<StackPanel>
   <Button Name="button1" Width="150" FontSize="12" Background="AliceBlue">
      Click Me!
   </Button>
</StackPanel>
```

Но вместо определения вида каждого элемента в отдельности можно определить стили, сохраняемые в ресурсах. Для полной настройки внешнего вида элементов управления вы можете использовать также шаблоны и сохранять их в ресурсах.

Стили

Для определения стилей вы можете использовать элемент `Style`, содержащий в себе элементы `Setter`. Посредством `Setter` вы специфицируете значение стиля, например, свойство `Button.Background` и значение `AliceBlue`.

Чтобы присвоить стили определенным элементам, вы можете назначить стиль всем элементам типа или использовать ключ для стиля. Чтобы назначить стиль всем элементам типа, используйте свойство `TargetType` класса `Style` и присвойте его `Button`, специфицируя расширение разметки `x:Type` следующим образом:

```
<Window.Resources>
<Style TargetType="{x:Type Button}">
  <Setter Property="Button.Background" Value="LemonChiffon" />
  <Setter Property="Button.FontSize" Value="18" />
</Style>
<Style x:Key="ButtonStyle">
  <Setter Property="Button.Background" Value="AliceBlue" />
  <Setter Property="Button.FontSize" Value="18" />
</Style>
</Window.Resources>
```

В приведенном ниже XAML-коде элемент `button2`, не имеющий стиля, определенного в свойствах элемента, получает стиль, определенный для всего типа `Button`. Для `button3` свойство `Style` устанавливается с расширением разметки `StaticResource`

в {StaticResource ButtonStyle}, где ButtonStyle специфицирует значение ключа ресурса стиля, определенного ранее, поэтому button3 имеет фон цвета aliceblue.

```
<Button Name="button2" Width="150">Click Me!</Button>
<Button Name="button3" Width="150" Style="{StaticResource ButtonStyle}">
   Click Me, Too!
</Button>
```

Вместо установки Background кнопки в единственное значение, вы можете сделать нечто большее — установить свойство Background в LinearGradientBrush, с определением градиентного цвета, как показано ниже:

```
<Style x:Key="FancyButtonStyle">
<Setter Property="Button.FontSize" Value="22" />
<Setter Property="Button.Foreground" Value="White" />
<Setter Property="Button.Background">
  <Setter.Value>
     <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
       <GradientStop Offset="0.0" Color="LightCyan" />
       <GradientStop Offset="0.14" Color="Cyan" />
       <GradientStop Offset="0.7" Color="DarkCyan" />
     </LinearGradientBrush>
  </Setter.Value>
</Setter>
</Style>
```

button4 получает приятный стиль, окрашиваясь в градиентный переход цвета cyan:

```
<Button Name="button4" Width="200" Style="{StaticResource FancyButtonStyle}">
   Fancy!
</Button>
```

То, что получится в результате, представлено на рис. 34.18.

Рис. 34.18. Пример использования стилей

Ресурсы

Как вы уже видели на примере использования стилей, обычно стили сохраняются в ресурсах. Вы можете определить любой элемент внутри ресурса; например, кисть (brush), созданную нами ранее для стиля фона кнопки, можно также определить в виде ресурса, так что вы сможете использовать ее в любом месте, где потребуется кисть.

Следующий пример определяет LinearGradient Brush с ключом по имени MyGradientBrush внутри ресурсов StackPanel. Элемент button1 получает свойство Background посредством расширения разметки StaticResource в ресурс MyGradientBrush. На рис. 34.19 показан результат, который даст следующий код XAML:

Рис. 34.19. Пример использования ресурса

```
<Window x:Class="ResourcesSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Resources Sample" Height="100" Width="300"
  >
<Window.Resources>
</Window.Resources>
```

```
 <StackPanel>
  <StackPanel.Resources>
     <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0.5,0" EndPoint="0.5,1">
       <GradientStop Offset="0.0" Color="LightCyan" />
       <GradientStop Offset="0.14" Color="Cyan" />
       <GradientStop Offset="0.7" Color="DarkCyan" />
     </LinearGradientBrush>
  </StackPanel.Resources>
  <Button Name="button1" Width="200" Height="50" Foreground="White"
     Background="{StaticResource MyGradientBrush}">
     Click Me!
  </Button>
 </StackPanel>
</Window>
```

Здесь ресурсы определены в `StackPanel`. В предыдущем примере ресурсы были определены в элементе `Window`. Базовый класс `FrameworkElement` определяет свойство `Resources` типа `ResourceDictionary`. Вот почему ресурсы могут быть определены для каждого класса, унаследованного от `FrameworkElement`, т.е. для любого элемента WPF.

Поиск ресурсов осуществляется иерархически. Если вы определите ресурс в окне, он применяется к каждому дочернему элементу окна. Если `Window` содержит `Grid`, а `Grid` содержит `StackPanel`, и вы определили ресурс для `StackPanel`, то этот ресурс применяется ко всем элементам управления внутри `StackPanel`. Если `StackPanel` содержит `Button`, и вы определяете ресурсы только для `Button`, то стиль будет распространяться только на кнопку.

Из-за иерархии вам следует быть внимательными, если вы используете `TargetType` без `Key` для стилей. Если вы определяете ресурсы для элемента `Canvas` и установите `TargetType` для стиля так, что он будет применяться к элементам `TextBox`, то стиль будет распространяться на все элементы `TextBox` в пределах `Canvas`. Этот стиль даже будет касаться элементов `TextBox`, содержащихся в `ListBox`, если `ListBox` находится в том же `Canvas`.

Если вам нужен одинаковый стиль более чем для одного элемента `Window`, вы можете определить стиль на уровне приложения. В WPF-проекте Visual Studio создается файл `App.xml` для определения глобальных ресурсов приложения. Стили приложения действуют для всех окон приложения; каждый элемент может обращаться к ресурсам, определенным в приложении. Если ресурсы не найдены в родительском окне, поиск ресурсов продолжается в `Application`.

```
<Application x:Class="ResourcesSample.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="Window1.xaml">
  <Application.Resources>
  </Application.Resources>
</Application>
```

Системные ресурсы

Существует также несколько ресурсов уровня системы, используемых для установки цветов и шрифтов, которые доступны всем приложениям. Эти ресурсы определены в классах `SystemColors`, `SystemFonts` и `SystemParameters`.

- С `SystemColors` вы получаете, настройки цвета для рамок, элементов управления, рабочего стола и окон, например, `ActiveBorderColor`, `ControlBrush`, `DesktopColor`, `WindowColor`, `WindowBrush` и тому подобного.

- Класс `SystemFonts` возвращает настройки шрифтов меню, линейки статуса и окна сообщений, например, `CaptionFont`, `DialogFont`, `MenuFont`, `MessageBoxFont`, `StatusFont` и т.д.
- Класс `SystemParameters` представляет настройки размеров кнопок меню, курсоров, пиктограмм, заголовков рамок, информации о времени и настройки клавиатуры, например, `BorderWidth`, `CaptionHeight`, `CaptionWidth`, `MenuButtonWidth`, `MenuPopupAnimation`, `MenuShowDelay`, `SmallIconHeight`, `SmallIconWidth` и т.д.

На рис. 34.20 показан диалог, в котором пользователь может конфигурировать эти настройки. Вы можете получить доступ к диалоговому окну Appearance (Внешний вид) через настройки Personalization (Персонализация) панели управления.

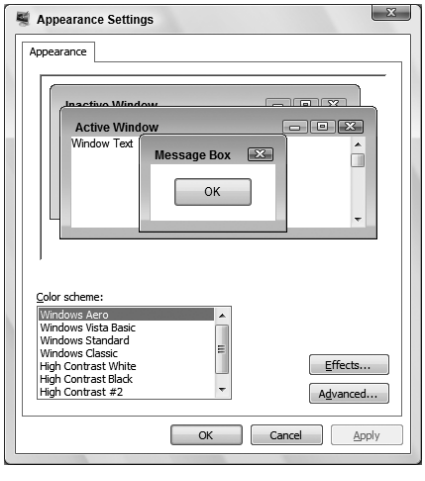

Рис. 34.20. Диалоговое окно Appearance

Обращение к ресурсам из кода

Чтобы обратиться к ресурсам из отделенного кода, базовый класс `FrameworkElement` реализует метод `FindResource()`, так что вы можете вызывать этот метод с каждым объектом WPF.

Чтобы показать, как это делается, мы не будем специфицировать фон для `button2`, но событию `Click` назначим метод `OnApplyResource`.

```
<StackPanel Name="MyContainer">
<StackPanel.Resources>
  <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0.5,0" EndPoint="0.5,1">
     <GradientStop Offset="0.0" Color="LightCyan" />
     <GradientStop Offset="0.14" Color="Cyan" />
     <GradientStop Offset="0.7" Color="DarkCyan" />
  </LinearGradientBrush>
</StackPanel.Resources>
<Button Name="button2" Width="200" Height="50" Click="button1_Click">
  Apply Resource Programmatically
</Button>
```

В реализации метода `OnAPplyResource()` воспользуемся методом `FindResource()` для кнопки, на которой был совершен щелчок. При этом произойдет иерархический поиск ресурса `MyGradientBrush`, и кисть будет применена к свойству `Background` элемента управления.

```
public void OnApplyResource(object sender, RoutedEventArgs e)
{
  Control ctrl = sender as Control;
  ctrl.Background = ctrl.FindResource("MyGradientBrush") as Brush;
}
```

Если `FindResource()` не находит ключа ресурса, возбуждается исключение. Если вы не знаете точно, доступен ли ресурс, то можете использовать вместо этого метод `TryFindResource()`. Метод `TryFindResource()` возвратит `null`, если не сможет найти ресурс.

Динамические ресурсы

С расширением разметки `StaticResource` поиск ресурсов осуществляется во время загрузки. Если ресурс должен измениться во время работы программы, вам следует использовать вместо этого расширение разметки `DynamicResource`.

В следующем примере используется тот же ресурс, определенный ранее. `button1` использует ресурс как `StaticResource`, а `button3` — как `DynamicResource` с расширением разметки `DynamicResource`. Кнопка `button2` применяется для программного изменения ресурса. У нее есть обработчик события `Click`, которым назначен метод `button2_Click`.

```
<Button Name="button1" Width="200" Height="50"
  Background="{StaticResource MyGradientBrush}">
  Static Resource
</Button>
<Button Name="button2" Width="200" Height="50"
  Click="button2_Click">
  Change Resource
</Button>
<Button Name="button3" Width="200" Height="50"
  Background="{DynamicResource MyGradientBrush}">
  Dynamic Resource
</Button>
```

Реализация `button2_Click()` очищает ресурсы `StackPanel` и добавляет новый ресурс с тем же именем — `MyGradientBrush`. Этот новый ресурс очень похож на ресурс, определенный в коде XAML; он просто определяет разные цвета.

```
public void OnChangeResource(object sender, RoutedEventArgs e)
{
  MyContainer.Resources.Clear();
  LinearGradientBrush brush = new LinearGradientBrush();
  brush.StartPoint = new Point(0.5, 0);
  brush.EndPoint = new Point(0.5, 1);
  GradientStopCollection stops = new GradientStopCollection();
  stops.Add(new GradientStop(Colors.White, 0.0));
  stops.Add(new GradientStop(Colors.Yellow, 0.14));
  stops.Add(new GradientStop(Colors.YellowGreen, 0.7));
  brush.GradientStops = stops;
  MyContainer.Resources.Add("MyGradientBrush", brush);
}
```

Если вы запустите это приложение и измените ресурс динамически — щелчком на третьей кнопке, то `button4` немедленно получит новый ресурс. При этом кнопка `button1`, использующая `StaticResource`, останется со старым ресурсом, который был загружен первоначально.

DynamicResource требует более высокой производительности, чем StaticResource, поскольку ресурс всегда загружается по необходимости. Применяйте DynamicResource только для тех ресурсов, изменения которых следует ожидать во время выполнения программы.

Триггеры

С помощью триггеров вы можете динамически изменять внешний вид ваших элементов управления, когда происходят некоторые события или изменяются значения некоторых свойств. Например, когда пользователь перемещает указатель мыши над кнопкой, внешний вид кнопки может изменяться. Обычно вы должны делать это в коде C#; но, имея дело с WPF, вы также можете воспользоваться XAML, когда затрагивается только пользовательский интерфейс.

Класс Style имеет свойство Triggers, которому можно назначить триггеры свойств. Следующий пример включает два элемента TextBox внутри панели Canvas. В ресурсах Window определен стиль TextBoxStyle, на который ссылаются элементы TextBox с использованием свойства Style. Стиль TextBoxStyle специфицирует, что Background устанавливается в LightBlue, а FontSize — в 17. Это — стиль элементов TextBox на момент запуска приложения. Использование триггеров изменяет стиль элементов управления. Триггеры определены внутри элементов Style.Triggers посредством элемента Trigger. Один триггер назначается свойству IsMouseOver, другой — свойству IsKeyboardFocused. Оба эти свойства определены с классом TextBox, к которому применяется стиль. Если IsMouseOver получает значение true, возбуждается триггер и устанавливает красный цвет фона, а размер шрифта — 22. Если TextBox имеет клавиатурный фокус, то свойство IsKeyboardFocused равно true, и тогда возбуждается второй триггер и устанавливает свойство Background элемента TextBox в Yellow (желтый фон).

```
<Window x:Class="TriggerSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="TriggerSample" Height="200" Width="400"
  >
<Window.Resources>
  <Style x:Key="TextBoxStyle" TargetType="{x:Type TextBox}">
  <Setter Property="Background" Value="LightBlue" />
  <Setter Property="FontSize" Value="17" />
  <Style.Triggers>
     <Trigger Property="IsMouseOver" Value="True">
       <Setter Property="Background" Value="Red" />
       <Setter Property="FontSize" Value="22" />
     </Trigger>
     <Trigger Property="IsKeyboardFocused" Value="True">
       <Setter Property="Background" Value="Yellow" />
       <Setter Property="FontSize" Value="22" />
     </Trigger>
  </Style.Triggers>
  </Style>
</Window.Resources>
<Canvas>
  <TextBox Canvas.Top="80" Canvas.Left="30" Width="300"
     Style="{StaticResource TextBoxStyle}" />
  <TextBox Canvas.Top="120" Canvas.Left="30" Width="300"
     Style="{StaticResource TextBoxStyle}" />
</Canvas>
</Window>
```

Вы не должны сбрасывать значения свойств в первоначальные, когда условие триггера перестает быть актуальным; например, вам не нужно определять триггер для `IsMouseOver=true` и `IsMouseOver=false`. Как только условие триггера становится не актуальным, изменения, внесенные действием триггера, сбрасываются в исходные значения автоматически.

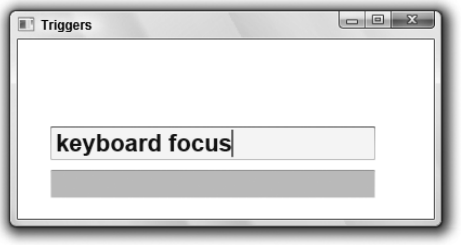

Рис. 34.21. Пример приложения с триггером

На рис. 34.21 показан пример приложения с триггером, где первое текстовое поле имеет фокус клавиатурного ввода, а второй — значения стиля по умолчанию для фона и размера шрифта.

> При использовании триггеров свойств очень легко изменять внешний вид элементов управления, шрифты, цвета, прозрачность и тому подобное; когда мышь проходит над ними, устанавливается клавиатурный фокус... и ни строчки программного кода для этого не требуется.

В классе `Trigger` определены свойства для спецификации действия триггера, которые перечислены в табл. 34.9.

Таблица 34.9. Свойства класса `Trigger`

Свойство	Описание
`PropertyValue`	С триггерами свойств свойства `Property` и `Value` используются для спецификации того, когда должен срабатывать триггер. Например, `Property="IsMouseOver"`, `Value="True"`.
`Setters`	Как только триггер сработал, вы можете применять `Setters` для определения коллекции элементов `Setter` для изменения значений свойств. В классе `Setter` определены свойства `Property`, `TargetName` и `Value` для изменения свойств объекта.
`EnterActions` `ExitActions`	Вместо определения `Setters` вы можете определить `EnterActions` и `ExitActions`. С обоими этими свойствами можно определить коллекцию элементов `TriggerAction`. Действие `EnterActions` возбуждается при запуске триггера (для триггеров свойств — когда применяется комбинация `Property`/`Value`), `ExitActions` возбуждается перед его завершением (точно в момент, когда комбинация `Property`/`Value` уже не применяется). Определенные вами действия триггеров порождены от базового класса `TriggerAction`, такого как `SoundPlayerAction` и `BeginStoryBoard`. С помощью `SoundPlayerAction` вы можете начать воспроизведение звука. `BeginStoryBoard` используется вместе с анимацией, как объясняется далее в этой главе.

Триггеры свойств — это лишь один из возможных в WPF типов триггеров. Другой тип триггеров — триггеры событий. Триггеры событий мы обсудим позднее в настоящей главе, когда речь пойдет об анимации.

Шаблоны

В этой главе вы уже видели, что элемент управления `Button` может иметь любое содержимое. Это может быть простой текст, но также вы можете добавить к кнопке элемент `Canvas`; этот элемент может содержать фигуры. Вы можете добавить к кнопке `Grid` или видеоклип. С простой кнопкой вы еще много чего можете сделать!

Функциональность и внешний вид элементов управления полностью разделены в WPF. Кнопка имеет некоторый внешний вид по умолчанию, но вы можете изменять его по своему усмотрению с помощью шаблонов.

WPF предоставляет несколько типов шаблонов, унаследованных от базового класса `FrameWorkTemplate` (табл. 34.10).

Таблица 34.10. Типы шаблонов, предлагаемые WPF

Тип шаблона	Описание
`ControlTemplate`	С помощью `ControlTemplate` вы можете специфицировать визуальную структуру элемента управления и переопределить его внешность.
`ItemsPanelTemplate`	Для `ItemsControl` вы можете специфицировать компоновку элементов, присваивая `ItemsPanelTemplate`. Каждый `ItemsControl` имеет `ItemsPanelTemplate` по умолчанию. Для `MenuItem` это `WrapPanel`. Элемент `StatusBar` использует `DockPanel`, а `ListBox` — `VirtualizingStackPanel`.
`DataTemplate`	`DataTemplate` очень полезны для графического отображения объектов. Стилизуя `ListBox`, вы увидите, что по умолчанию элементы `ListBox` отображаются согласно выводу метода `ToString()`. Применяя `DataTemplate`, вы можете переопределить это поведение и определить собственное представление элементов списка.
`HierarchicalDataTemplate`	`HierarchicalDataTemplate` используется для упорядочивания дерева объектов. Этот элемент управления поддерживает `HeaderedItemsControl`, такие как `TreeViewItem` и `MenuItem`.

В следующем примере показано несколько кнопок и позднее — пошаговая настройка окон списков, так что вы сможете увидеть промежуточные результаты таких изменений. Начнем с двух очень простых кнопок, первая из которых вообще не будет иметь никакого стиля, а вторая — ссылается на стиль `ButtonStyle1` с изменениями `Background` и `FontSize`. Первоначальный результат показан на рис. 34.22.

```
<Window x:Class="TemplateSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Template Sample" Height="300" Width="300">
<Window.Resources>
  <Style x:Key="ButtonStyle1" TargetType="{x:Type Button}">
     <Setter Property="Background" Value="Yellow" />
     <Setter Property="FontSize" Value="18" />
  </Style>
</Window.Resources>
<StackPanel>
  <Button Name="button1" Height="50" Width="150">Default Button</Button>
  <Button Name="button2" Height="50" Width="150"
       Style="{StaticResource ButtonStyle1}">Styled Button
  </Button>
</StackPanel>
</Window>
```

Теперь добавим в ресурсы новый стиль `ButtonStyle2`. Этот стиль опять устанавливает `TargetType` в тип `Button`. Класс `Setter` теперь специфицирует свойство `Template`. Указывая свойство `Template`, вы можете полностью изменить внешний вид кнопки. Значение свойства `Template` определяется элементом `ControlTemplate`.

Класс ControlTemplate определяет содержимое элемента управления и разрешает доступ к содержимому из самого элемента управления, в чем вы вскоре убедитесь. Здесь ControlTemplate определяет Grid из двух строк. Строки используют звездообразное изменение размеров, причем первая строка по высоте вдвое больше второй. Затем определяются два элемента Rectangle. Первый прямоугольник охватывает обе строки, устанавливает свойство Stroke в Green (для подчеркивания зеленым), а также значения RoundX и RoundY для скругленных углов. Второй прямоугольник, расположенный внутри первой строки, имеет свойство Fill, установленное в кисть линейного градиента. button3 с содержимым Template Button, ссылается на стиль ButtonStyle2. На рис. 34.23 показана кнопка button3 с новым стилем, но отсутствующим содержимым.

Рис. 34.22. Пример применения шаблона с двумя кнопками

```
<Window x:Class="TemplateSample.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Template Sample" Height="300" Width="300"
    >
<Window.Resources>
<!-- другие стили -->
  <Style x:Key="ButtonStyle2" TargetType="{x:Type Button}">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate>
           <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="2*" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <Rectangle Grid.RowSpan="2" RadiusX="4" RadiusY="8" Stroke="Green"
            />
            <Rectangle RadiusX="4" RadiusY="8" Margin="2">
              <Rectangle.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                  <GradientStop Offset="0" Color="LightBlue" />
                  <GradientStop Offset="0.5" Color="#afff" />
                  <GradientStop Offset="1" Color="#6faa" />
                </LinearGradientBrush>
              </Rectangle.Fill>
            </Rectangle>
           </Grid>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</Window.Resources>
<StackPanel>
<!-- другие кнопки -->
<Button Name="button3" Background="Yellow" Height="100" Width="220"
FontSize="24"
    Style="{StaticResource ButtonStyle2}">Template Button
</Button>
</StackPanel>
</Window>
```

Теперь кнопка выглядит совершенно по-другому, но определенное в ней содержимое на рис. 34.23 полностью отсутствует. Ранее созданный шаблон необходимо расширить. Первый прямоугольник в шаблоне теперь имеет свойство `Fill` установленное в `{TemplateBinding Background}`. Расширение разметки `TemplateBinding` позволяет шаблону элемента управления использовать содержимое шаблонного элемента. Здесь прямоугольник заполняется фоном, определенным для кнопки. Для `button3` определен желтый фон, так что этот фон комбинируется с фоном из второго прямоугольника шаблона элемента управления. После определения второго прямоугольника используется элемент `ContentPresenter`. Элемент `ContentPresenter` берет содержимое из шаблонного элемента управления и помещает его, как определено в данном случае, в обе строки, поскольку `Grid.RawSpan` имеет значение 2. Если определен `ContentPresenter`, то `TargetType` с `ControlTemplate` также должен быть установлен. Содержимое позиционируется установкой свойств `HorizontalAlignment`, `VerticalAlignment` и `Margin` в значения, определенные самой кнопкой посредством расширений разметки `TemplateBinding`. С помощью `ControlTemplate` вы можете также определить триггеры в ресурсах, как было показано ранее. На рис. 34.24 представлен новый вид кнопки, включая содержимое и фон, скомбинированные с шаблоном.

```
<Style x:Key="ButtonStyle2" TargetType="{x:Type Button}">
  <Setter Property="Template">
     <Setter.Value>
     <ControlTemplate TargetType="{x:Type Button}" >
       <Grid>
         <Grid.RowDefinitions>
              <RowDefinition Height="2*" />
              <RowDefinition Height="*" />
         </Grid.RowDefinitions>
         <Rectangle Grid.RowSpan="2" RadiusX="4" RadiusY="8" Stroke="Green"
                Fill="{TemplateBinding Background}" />
         <Rectangle RadiusX="4" RadiusY="8" Margin="2">
           <Rectangle.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                     <GradientStop Offset="0" Color="LightBlue" />
                     <GradientStop Offset="0.5" Color="#afff" />
                     <GradientStop Offset="1" Color="#6faa" />
                </LinearGradientBrush>
           </Rectangle.Fill>
         </Rectangle>
         <ContentPresenter Grid.RowSpan="2"
                HorizontalAlignment="{TemplateBinding
                HorizontalContentAlignment}"
                VerticalAlignment="{TemplateBinding VerticalContentAlignment}"
                Margin="{TemplateBinding Padding}" />
         </Grid>
         <ControlTemplate.Triggers>
           <Trigger Property="IsMouseOver" Value="True">
             <Setter Property="Foreground" Value="Aqua" />
           </Trigger>
           <Trigger Property="IsPressed" Value="True">
             <Setter Property="Foreground" Value="Black" />
           </Trigger>
         </ControlTemplate.Triggers>
       </ControlTemplate>
     </Setter.Value>
  </Setter>
</Style>
</Window.Resources>
```

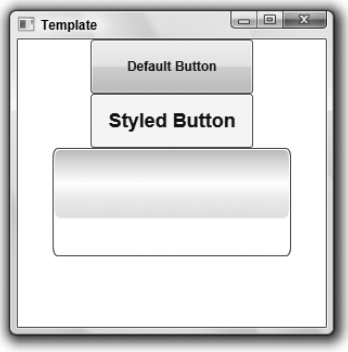

Рис. 34.23. Третья кнопка в примере применения шаблона

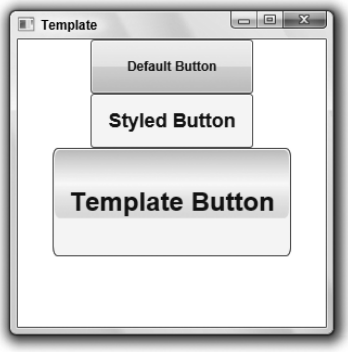

Рис. 34.24. Новый вид кнопки

Давайте попробуем сделать нашу кнопку еще симпатичнее: воспользуемся средством прозрачности. Стиль `GetButton` устанавливает свойства `Background`, `Height`, `Foreground`, `Margin` и `Template`. Шаблон — наиболее интересный аспект этого стиля. Шаблон специфицирует `Grid` с единственной строкой и столбцом.

Внутри этой ячейки вы можете найти прямоугольник по имени `GelBackground`. Этот прямоугольник имеет скругленные углы и кисть линейного градиента для штрихов. Скругленные углы определены установками `RadiusX` и `RadiusY`. Штрих, очерчивающий прямоугольник, очень тонкий, поскольку свойство `StrokeThickness` установлено в `0.35`.

Второй прямоугольник `GetShine` — это просто маленький прямоугольник высотой 15 пикселей, и из-за настройки `Margin` он видим внутри первого прямоугольника. Штрих прозрачен, так что нет линии, очерчивающей прямоугольник. Этот прямоугольник просто использует кисть заполнения линейного градиента, которая дает переход от легкого, частично прозрачного цвета до полной прозрачности. Это создает некоторый "мерцающий" (shimmering) эффект.

После этих двух прямоугольников следует элемент `ContentPresenter`, определяющий выравнивание содержимого и принимающий содержимое кнопки для отображения.

Такая стилизованная кнопка выглядит на экране очень симпатично. Однако пока нет никакого действия в ответ на щелчок кнопкой мыши или перемещение курсора мыши над поверхностью кнопки. Для шаблонно-стилизованной кнопки вы должны иметь триггеры, изменяющие отображение кнопки в ответ на щелчки. Триггер свойства `IsMouseOver` определяет новое значение свойства `Rectangle.Fill` с отличающимся цветом для кисти радиального градиента. На прямоугольник, получающий новое заполнение, ссылается свойство `TargetName`. Триггер свойства `IsPressed` очень похож; здесь просто используется другая кисть радиального градиента для заполнения прямоугольника. На рис. 34.25 вы можете видеть кнопку, ссылающуюся на стиль `GelButton`. На рис. 34.26 показана та же кнопка в момент перемещения над ней курсора мыши, так что видим эффект от кисти радиального градиента.

```
<Style x:Key="GelButton" TargetType="{x:Type Button}">
<Setter Property="Background" Value="Black" />
<Setter Property="Height" Value="40" />
<Setter Property="Foreground" Value="White" />
<Setter Property="Margin" Value="3" />
<Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="{x:Type Button}">
```

```
    <Grid>
      <Rectangle Name="GelBackground" RadiusX="9" RadiusY="9"
           Fill="{TemplateBinding Background}" StrokeThickness="0.35">
      <Rectangle.Stroke>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
          <GradientStop Offset="0" Color="White" />
          <GradientStop Offset="1" Color="#666666" />
        </LinearGradientBrush>
      </Rectangle.Stroke>
      </Rectangle>
      <Rectangle Name="GelShine" Margin="2,2,2,0" VerticalAlignment="Top"
           RadiusX="6" RadiusY="6" Stroke="Transparent" Height="15px">
        <Rectangle.Fill>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="#ccffffff" />
            <GradientStop Offset="1" Color="Transparent" />
          </LinearGradientBrush>
        </Rectangle.Fill>
      </Rectangle>
      <ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
            HorizontalAlignment="Center"
            Content="{TemplateBinding Content}" />
     </Grid>
       <ControlTemplate.Triggers>
         <Trigger Property="IsMouseOver" Value="True">
           <Setter Property="Rectangle.Fill" TargetName="GelBackground">
             <Setter.Value>
               <RadialGradientBrush>
                 <GradientStop Offset="0" Color="Lime" />
                 <GradientStop Offset="1" Color="DarkGreen" />
               </RadialGradientBrush>
             </Setter.Value>
           </Setter>
           <Setter Property="Foreground" Value="Black" />
         </Trigger>
         <Trigger Property="IsPressed" Value="True">
           <Setter Property="Rectangle.Fill" TargetName="GelBackground">
             <Setter.Value>
               <RadialGradientBrush>
                 <GradientStop Offset="0" Color="#ffcc00" />
                 <GradientStop Offset="1" Color="#cc9900" />
               </RadialGradientBrush>
             </Setter.Value>
           </Setter>
         </Trigger>
       </ControlTemplate.Triggers>
    </ControlTemplate>
  </Setter.Value>
</Setter>
</Style>
```

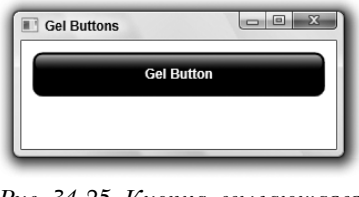

Рис. 34.25. Кнопка, ссылающаяся на стиль `GelButton`

Рис. 34.26. Кнопка, ссылающаяся на стиль `GelButton`*, в момент перемещения над ней курсора мыши*

Вместо обычного прямоугольного кнопка может иметь вид эллипса. В следующем примере вы также увидите, как один стиль может базироваться на другом стиле.

Рис. 34.27. Кнопка с формой, отличной от прямоугольной

Стиль `RoundedGelButton` может быть построен на основе стиля `GelButton` установкой свойства `BasedOn` с элементом `Style`. Если один стиль основывается на другом, то новый стиль получает все установки базового стиля, если только какие-либо из настроек не переопределяются. Например, `RoundedGelButton` получает установки `ForeGround` и `Margin` из `GelButton`, поскольку эти настройки не переопределены. Если вы измените установки в базовом стиле, то все стили, основанные на нем, автоматически получат эти новые значения.

Переопределим в новом стиле свойства `Height` и `Template`. Здесь шаблон определит два элемента `Ellipse` вместо прямоугольников. Внешний эллипс `GelBackground` определит черный эллипс с градиентной линией вокруг него. Второй эллипс будет поменьше, с небольшим полем (`5`) вверху и полем побольше (`50`) внизу. Этот эллипс, опять же, имеет линейный градиент, от светлого цвета до прозрачного, имитирующий эффект блеска. Снова будут присутствовать триггеры `IsMouseOver` и `IsPressed`, которые изменят значение свойства `Fill` первого из эллипсов.

В результате получим новую кнопку, основанную на стиле `RoundedGelButton`, и это по-прежнему будет именно кнопка, как показано на рис. 34.27.

```
<Style x:Key="RoundedGelButton" BasedOn="{StaticResource GelButton}"
  TargetType="Button">
<Setter Property="Width" Value="100" />
<Setter Property="Height" Value="100" />
<Setter Property="Grid.Row" Value="2" />
<Setter Property="Template">
  <Setter.Value>
     <ControlTemplate TargetType="{x:Type Button}">
       <Grid>
           <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
            <Ellipse.Stroke>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                   <GradientStop Offset="0" Color="#ff7e7e7e" />
                   <GradientStop Offset="1" Color="Black" />
                </LinearGradientBrush>
            </Ellipse.Stroke>
           </Ellipse>
           <Ellipse Margin="15,5,15,50">
              <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                   <GradientStop Offset="0" Color="#aaffffff" />
                   <GradientStop Offset="1" Color="Transparent" />
                </LinearGradientBrush>
              </Ellipse.Fill>
            </Ellipse>
            <ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
                HorizontalAlignment="Center" Content="{TemplateBinding Content}"
            />
       </Grid>
       <ControlTemplate.Triggers>
           <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Rectangle.Fill" TargetName="GelBackground">
                <Setter.Value>
```

```
                    <RadialGradientBrush>
                      <GradientStop Offset="0" Color="Lime" />
                      <GradientStop Offset="1" Color="DarkGreen" />
                    </RadialGradientBrush>
                </Setter.Value>
            </Setter>
            <Setter Property="Foreground" Value="Black" />
           </Trigger>
           <Trigger Property="IsPressed" Value="True">
            <Setter Property="Rectangle.Fill" TargetName="GelBackground">
                <Setter.Value>
                <RadialGradientBrush>
                    <GradientStop Offset="0" Color="#ffcc00" />
                    <GradientStop Offset="1" Color="#cc9900" />
                </RadialGradientBrush>
                </Setter.Value>
            </Setter>
            <Setter Property="Foreground" Value="Black" />
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Стилизация `ListBox`

Изменение стиля кнопки или метки — простая задача. А как насчет изменения стиля элемента, содержащего список элементов, например, `ListBox`? Кроме того, окно списка имеет поведение и внешний вид. Окно списка может отображать список элементов; вы можете выбирать в нем один или более элементов. Для обеспечения требуемого поведения в классе `ListBox` определены методы, свойства и события. Внешний вид `ListBox` отделен от его поведения. Элемент `ListBox` имеет внешний вид по умолчанию, но вы можете изменить его, создав соответствующий шаблон.

Чтобы отобразить некоторые элементы в списке, мы создадим класс `Country`, представляющий наименование и флаг, с путем к файлу его изображения. Класс `Country` определяет свойства `Name` и `ImagePath`, а также имеет переопределенный метод `ToString()` для строкового представления по умолчанию:

```
public class Country
{
  public Country(string name)
    : this(name, null)
  {
  }

  public Country(string name, string imagePath)
  {
    this.name = name;
    this.imagePath = imagePath;
  }

  public string Name { get; set; }
  public string ImagePath { get; set; }
  public override string ToString()
  {
    return Name;
  }
}
```

Статический класс Countries возвратит список нескольких стран, которые будут отображены:

```
public static class Countries
{
  public static IEnumerable<Country> GetCountries()
  {
    List<Country> countries = new List<Country>();
    countries.Add(new Country("Austria", "Images/Austria.bmp"));
    countries.Add(new Country("Germany", "Images/Germany.bmp"));
    countries.Add(new Country("Norway", "Images/Norway.bmp"));
    countries.Add(new Country("USA", "Images/USA.bmp"));
    return countries;
  }
}
```

Внутри файла, содержащего отделенный код, в конструкторе класса Window1 свойство DataContext экземпляра Window1 устанавливается в список стран, возвращенных методом Countries.GetCountries(). (Свойство DataContext — это средство привязки данных, о котором пойдет речь в следующей главе.)

```
public partial class Window1 : System.Windows.Window
{
  public Window1()
  {
    InitializeComponent();
    this.DataContext = Countries.GetCountries();
  }
}
```

Внутри кода XAML определяется ListBox по имени countryList. Элемент countryList не имеет отдельного стиля; он использует внешний вид по умолчанию элемента ListBox. Свойство ItemSource устанавливается в расширение разметки Binding, используемое средством привязки данных, и из отделенного кода видно, что привязка осуществляется к массиву объектов Country. На рис. 34.28 показан внешний вид ListBox, устанавливаемый по умолчанию.

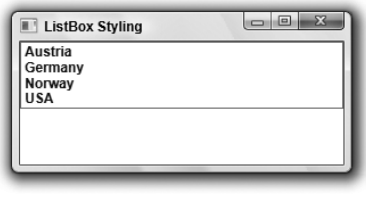

Рис. 34.28. Внешний вид ListBox, устанавливаемый по умолчанию

Как видим, по умолчанию он отображает лишь названия стран, возвращенные методом ToString(), в виде простого списка.

```
<Window x:Class="ListboxStyling.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Listbox Styling" Height="300" Width="300">
<StackPanel>
  <ListBox Name="countryList1" ItemsSource="{Binding}" />
</StackPanel>
</Window>
```

Объекты Country несут в себе как имя, так и изображение флага. Конечно, вы также можете отображать оба значения в окне списка. Но чтобы добиться этого, придется определить шаблон.

Элемент ListBox содержит элементы ListBoxItem. Вы можете определить содержимое элемента с ItemTemplate. Стиль listBoxStyle1 определяет ItemTemplate со значением DataTemplate. Шаблон DataTemplate применяется для привязки данных к элементам. Вы можете использовать расширение разметки Binding с элементами DataTemplate.

DataTemplate содержит Grid с тремя столбцами. Первый столбец содержит строку Country:, второй — название страны, а третий — флаг этой страны. Поскольку имена стран имеют разную длину, но в списке должны быть представлены строками равной длины, для определения второго столбца устанавливается свойство SharedSizeGroup. Информация разделения размера для столбцов используется только потому, что также установлено свойство Grid.IsSharedSizeScope.

После определений столбца и строки вы можете видеть два элемента TextBlock. Первый элемент TextBlock содержит текст Country:, а второй элемент TextBlock привязывает свойство Name, определенное в классе Country.

Содержимое третьего столбца — элемент Border, содержащий Grid. Элемент Grid содержит Rectangle с линейной "обтекающей" кистью и элемент Image, привязанный к свойству ImagePath класса Country. На рис. 34.29 показаны страны в ListBox, который выглядит совершенно по-другому, чем до этого.

```
<Window.Resources>
<Style x:Key="listBoxStyle1" TargetType="{x:Type ListBox}" >
  <Setter Property="ItemTemplate">
     <Setter.Value>
       <DataTemplate>
         <Grid>
            <Grid.ColumnDefinitions>
               <ColumnDefinition Width="Auto" />
               <ColumnDefinition Width="*" SharedSizeGroup="MiddleColumn" />
               <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
               <RowDefinition Height="60" />
            </Grid.RowDefinitions>
            <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                 FontStyle="Italic" Grid.Column="0">Country:</TextBlock>
            <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                 Text="{Binding Name}" FontWeight="Bold" Grid.Column="1" />
            <Border Margin="4,0" Grid.Column="2" BorderThickness="2"
            CornerRadius="4">
               <Border.BorderBrush>
                 <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                   <GradientStop Offset="0" Color="#aaa" />
                   <GradientStop Offset="1" Color="#222" />
                 </LinearGradientBrush>
               </Border.BorderBrush>
               <Grid>
                 <Rectangle>
                    <Rectangle.Fill>
                      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                           <GradientStop Offset="0" Color="#444" />
                           <GradientStop Offset="1" Color="#fff" />
                      </LinearGradientBrush>
                    </Rectangle.Fill>
                 </Rectangle>
                 <Image Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
               </Grid>
            </Border>
         </Grid>
       </DataTemplate>
     </Setter.Value>
   </Setter>
  <Setter Property="Grid.IsSharedSizeScope" Value="True" />
</Style>
</Window.Resources>
```

Совершенно не обязательно, чтобы ListBox содержал элементы, следующие друг за другом в вертикальном направлении. Вы можете придать ему совершенно другой вид, но с той же функциональностью. Следующий стиль, listBoxStyle2, определяет шаблон, в котором элементы отображаются горизонтально с помощью линейки прокрутки.

Рис. 34.29. Измененный внешний вид ListBox

В предыдущем примере для определения того, как элементы списка должны выглядеть в ListBox по умолчанию был создан только ItemTemplate. Теперь создадим шаблон, определяющий другой ListBox. Шаблон будет содержать элемент ControlTemplate для определения элементов ListBox. Элементом теперь будет ScrollViewer — представление с линейкой прокрутки, содержащее внутри себя StackPanel. Поскольку элементы списка теперь должны располагаться горизонтально, Orientation экземпляра StackPanel устанавливаем в Horizontal. Панель StackPanel будет содержать элементы, определенные посредством ItemTemplate, поэтому IsItemHost экземпляра StackPanel устанавливаем в true. Свойство IsItemHost — это свойство, доступное в каждом элементе Panel, который может содержать список элементов.

ItemTemplate определяет вид элементов панели StackPanel. Здесь создадим сетку (grid) из двух строк, первая из которых будет содержать элементы Image, привязанные к свойству ImagePath, а вторая — элементы TextBlock, привязанные к свойству Name.

На рис. 34.30 показан ListBox, стилизованный с помощью listBoxStyle2, в котором линейка прокрутки появляется автоматически, когда представление оказывается слишком маленьким, чтобы отобразить сразу весь список.

```
<Style x:Key="listBoxStyle2" TargetType="{x:Type ListBox}">
<Setter Property="Template">
  <Setter.Value>
     <ControlTemplate TargetType="{x:Type ListBox}">
       <ScrollViewer HorizontalScrollBarVisibility="Auto">
          <StackPanel Name="StackPanel1" IsItemsHost="True" Orientation="Horizontal" />
       </ScrollViewer>
     </ControlTemplate>
  </Setter.Value>
</Setter>
<Setter Property="VerticalAlignment" Value="Center" />
<Setter Property="ItemTemplate">
  <Setter.Value>
     <DataTemplate>
       <Grid>
       <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
       </Grid.ColumnDefinitions>
       <Grid.RowDefinitions>
            <RowDefinition Height="60" />
            <RowDefinition Height="30" />
       </Grid.RowDefinitions>
       <Image Grid.Row="0" Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
       <TextBlock Grid.Row="1" FontSize="14" HorizontalAlignment="Center"
            Margin="5" Text="{Binding Name}" FontWeight="Bold" />
       </Grid>
     </DataTemplate>
  </Setter.Value>
</Setter>
</Style>
```

Несомненно, на этом примере вы видите преимущества отделения внешнего вида элементов управления от его поведения. Возможно, у вас уже появилось множество идей относительно способов отображения ваших элементов в списке, которые больше отвечают требованиям, предъявляемым к вашему приложению. Может быть, вы просто хотите отобразить столько элементов, сколько уместится в окне, позиционируя их горизонтально, а затем продолжая на следующей строке вертикально. И здесь на помощь приходит `WrapPanel`. И, конечно, вы можете поместить `WrapPanel` внутрь шаблона `ListBox`, как показано в `listBoxStyle3`. На рис. 34.31 продемонстрирован результат применения `WrapPanel`.

```
<Style x:Key="listBoxStyle3" TargetType="{x:Type ListBox}">
<Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="{x:Type ListBox}">
      <ScrollViewer VerticalScrollBarVisibility="Auto"
           HorizontalScrollBarVisibility="Disabled">
       <WrapPanel IsItemsHost="True" />
      </ScrollViewer>
    </ControlTemplate>
  </Setter.Value>
</Setter>
<Setter Property="ItemTemplate">
  <Setter.Value>
    <DataTemplate>
      <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="140" />
          </Grid.ColumnDefinitions>
          <Grid.RowDefinitions>
            <RowDefinition Height="60" />
            <RowDefinition Height="30" />
          </Grid.RowDefinitions>
          <Image Grid.Row="0" Width="48" Margin="2,2,2,1"
                Source="{Binding ImagePath}" />
          <TextBlock Grid.Row="1" FontSize="14" HorizontalAlignment="Center"
                Margin="5" Text="{Binding Name}" />
        </Grid>
      </DataTemplate>
    </Setter.Value>
   </Setter>
</Style>
```

В следующей главе вы узнаете больше о `DateTemplate` с функциональностью привязки данных.

Рис. 34.30. Внешний вид `ListBox`, стилизованный с помощью `listBoxStyle2`

Рис. 34.31. Результат применения панели `WrapPanel`

Резюме

В этой главе был проведен небольшой экскурс по невероятно мощным средствам WPF. WPF позволяет легко разделить работу между разработчиками и дизайнерами. Как Microsoft Expression Blend, так и Visual Studio позволяют работать с кодом XAML. По сравнению со старыми приложениями Windows Forms код XAML обеспечивает лучшее отделение пользовательского интерфейса от стоящей за ним функциональности. Все интерфейсные средства могут быть созданы в XAML, а функциональность — в отделенном коде.

Вы видели много элементов управления и контейнеров, основанных на векторной графике. Благодаря векторной графике, элементы WPF могут масштабироваться, деформироваться и вращаться. За счет гибкости элементов содержимого механизм обработки событий основан на пузырьковом и туннельном распространении событий.

В следующей главе мы продолжим рассмотрением таких средств WPF, как анимации, 3D, привязка данных и ряда других.