- Знакомство с объектом Binding
- Управление визуализацией
- Настройка представления коллекции
- Поставщики данных
- Дополнительные вопросы
- А теперь все вместе: клиент Twitter на чистом XAML

13

Привязка к данным

В WPF термин *данные* обычно употребляется для описания произвольного объекта .NET. Примерами такого соглашения могут служить словосочетания *привязка к данным*, *шаблоны данных* и *триггеры данных*. В роли данных может выступать объект-коллекция, XML-файл, веб-служба, таблица базы данных, объект написанного вами класса и даже элемент WPF, к примеру `Button`.

Поэтому под привязкой к данным понимается установление некоторой связи между произвольными объектами .NET. Классический пример – визуальное представление (например, в виде списка `ListBox` или сетки `DataGrid`) данных, хранящихся в XML-файле, в базе данных или в коллекции в памяти. Вместо того чтобы писать цикл обхода источника данных и вручную добавлять объекты `ListBoxItem` в `ListBox`, не проще ли было бы сказать нечто вроде: «Привет, `ListBox`! Возьми-ка элементы вон оттуда. И будь добр, следи за их актуальностью. Ах да, еще не забудь отформатировать вот так». Механизм привязки к данным позволяет все это и многое другое.

Знакомство с объектом Binding

Ключом к механизму привязки к данным является объект класса `System.Windows.Data.Binding`, который «склеивает» между собой два свойства и открывает коммуникационный канал между ними. Объект `Binding` можно один раз настроить и поручить ему дальнейшую заботу о синхронизации.

Использование объекта Binding в процедурном коде

Допустим, требуется добавить в приложение Photo Gallery, рассмотренное в предыдущих главах, элемент `TextBlock`, в котором будет отображаться имя текущей папки, расположив его над списком `ListBox`:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
  Background="AliceBlue" FontSize="16" />
```

Можно было бы обновлять этот текстовый блок вручную при каждом изменении свойства SelectedItem элемента TreeView:

```
void treeView_SelectedItemChanged(object sender,
  RoutedPropertyChangedEventArgs<object> e)
{
  currentFolder.Text = (treeView.SelectedItem as TreeViewItem).Header.ToString();
  Refresh();
}
```

Но, воспользовавшись объектом Binding, мы сможем избавиться от этой строки кода, заменив ее однократной инициализацией в конструкторе класса MainWindow:

```
public MainWindow()
{
  InitializeComponent();

  Binding binding = new Binding();
  // Задаем объект-источник
  binding.Source = treeView;
  // Задаем свойство-источник
  binding.Path = new PropertyPath("SelectedItem.Header");
  // Присоединяем к нему свойство-приемник
  currentFolder.SetBinding(TextBlock.TextProperty, binding);
}
```

После такого изменения свойство currentFolder.Text будет автоматически обновляться при каждом изменении свойства treeView.SelectedItem.Header. Если в дереве TreeView будет выбран элемент, не обладающий свойством Header (в приложении Photo Gallery такого быть не может), то привязка завершится неудачно (без каких-либо исключений) и вернет значение свойства, подразумеваемое по умолчанию (в данном случае пустую строку). Впрочем, есть способы получать уведомления о таком развитии события – мы рассмотрим их ниже.

Такая модификация кода вряд ли может считаться улучшением, потому что вместо одной строки нам пришлось написать аж четыре! Но ведь это очень простой случай использования привязки. В последующих примерах мы убедимся, что привязка позволяет значительно уменьшить объем кода, который пришлось бы написать вручную для достижения эквивалентного результата.

Для объекта Binding определены понятия свойства-*источника* и свойства-*приемника*. Свойство-источник (в нашем случае treeView.SelectedItem.Header) задается в два приема: запись ссылки на объект-источник в свойство Source и имени интересующего нас свойства (или цепочки, состоящей из имени свойства и его «субсвойств»), обернутого объектом PropertyPath, в свойство Path. Чтобы ассоциировать Binding со свойством-приемником (в нашем случае currentFolder.Text) Binding, нужно вызвать метод SetBinding (унаследованный от класса

FrameworkElement или FrameworkContentElement), передав ему нужное свойство зависимости и сам объект Binding.

СОВЕТ

На самом деле существует два способа настроить объект Binding в процедурном коде. Первый – вызвать метод экземпляра SetBinding от имени объекта FrameworkElement или FrameworkContentElement, как было показано выше. Второй – вызвать статический метод SetBinding класса BindingOperations. Ему передаются те же аргументы, что и методу экземпляра, плюс дополнительный первый аргумент, представляющий объект-приемник:

```
BindingOperations.SetBinding(currentFolder, TextBlock.TextProperty, binding);
```

Достоинство статического метода в том, что первый параметр имеет тип DependencyObject, то есть открывается возможность осуществлять привязку к объектам, не наследующим ни FrameworkElement, ни FrameworkContentElement (например, класса Freezable).

КОПНЕМ ГЛУБЖЕ

Удаление объекта Binding

Если вы не хотите, чтобы объект Binding существовал на протяжении всего времени работы приложения, то его можно в любой момент «отключить» с помощью статического метода BindingOperations.ClearBinding. (Правда, это делают редко.) Методу передается объект-приемник и его свойство зависимости. Например:

```
BindingOperations.ClearBinding(currentFolder, TextBlock.TextProperty);
```

Если к объекту-приемнику присоединено более одного объекта Binding, то можно очистить их все за один прием, вызвав метод BindingOperations.ClearAllBindings:

```
BindingOperations.ClearAllBindings(currentFolder);
```

Еще один способ убрать привязку – напрямую записать в свойство-приемник новое значение, например:

```
currentFolder.Text = "I am no longer receiving updates.";
```

Но такой способ очищает только односторонние привязки. (Различные типы объектов Binding обсуждаются ниже в разделе «Настройка потока данных».) Подход на основе метода ClearBinding в любом случае более гибкий, поскольку оставляет свойству зависимости возможность принимать данные из других источников с более низким приоритетом (триггеров стилей, механизма наследования значений свойств и т. д.). Напомним, что приоритеты, учитываемые при определении значения свойства, рассматривались в главе 3 «Основные принципы WPF». У объекта Binding, заданного с помощью метода SetBinding, такой же приоритет, как у локального значения, а метод ClearBinding просто исключает этот источник поставки значений точно так же, как метод ClearValue делает это для *любого* локального значения. (На самом деле реализация ClearBinding всего лишь вызывает метод ClearValue для объекта-приемника!)

Использование объекта Binding в XAML

Поскольку вызвать метод SetBinding из XAML-кода невозможно, в WPF включено расширение разметки, позволяющее использовать объект Binding декларативно. На самом деле класс Binding сам является классом расширения разметки (несмотря на то, что в его имени нет стандартного суффикса Extension).

Чтобы использовать объект Binding в XAML, нужно записать в свойство-приемник сам этот объект, а затем с помощью стандартного синтаксиса расширения разметки настроить его свойства. Следовательно, показанный выше процедурный код можно заменить таким дополнением к объявлению элемента currentFolder:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
  Text="{Binding ElementName=treeView, Path=SelectedItem.Header}"
  Background="AliceBlue" FontSize="16" />
```

Ну что, привязка к данным уже начинает выглядеть более привлекательной, чем кодирование вручную? Соединение между источником и приемником не только записывается лаконично, но еще и вынесено из процедурного кода.

СОВЕТ

Помимо конструктора по умолчанию в классе Binding имеется конструктор, принимающий один аргумент Path. Следовательно, можно оформить расширение разметки и по-другому, передав путь Path конструктору, а не устанавливая его явно в качестве свойства. Иными словами, приведенный выше фрагмент XAML можно было бы переписать и в таком виде:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
  Text="{Binding SelectedItem.Header, ElementName=treeView}"
  Background="AliceBlue" FontSize="16" />
```

Оба подхода почти идентичны, за исключением некоторых тонких различий в разрешении префиксов пространств имен в путях к свойствам. В целом, явное задание свойства Path надежнее.

Отметим, что в этом XAML-коде для установки объекта-источника применяется свойство ElementName, а не Source, как в предыдущем разделе. И то и другое допустимо в обоих контекстах, но ElementName удобнее в XAML, потому что для него нужно указать только имя элемента-источника. Впрочем, с появлением в WPF 4 расширения разметки x:Reference свойство Source можно задать следующим образом:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
  Text="{Binding Source={x:Reference TreeView}, Path=SelectedItem.Header}"
  Background="AliceBlue" FontSize="16" />
```

СОВЕТ

С помощью свойства TargetNullValue объекта Binding можно указать специальное значение, которое будет возвращаться, если значение реального свойства-источника равно null. Например, в показанном ниже текстовом блоке будет отображаться не пустая строка, а строка «Nothing is selected.» (Ничего не выбрано) в случае, когда значение-источник равно null:

```
<TextBlock Text="{Binding ... TargetNullValue=Nothing is selected.}" .../>
```

Свойство TargetNullValue может выручить и в более сложных ситуациях, когда какое-то свойство объекта-приемника не может принимать значение null.

КОПНЕМ ГЛУБЖЕ

Свойство RelativeSource объекта Binding

Еще один способ задать источник данных – воспользоваться свойством RelativeSource объекта Binding, которое ссылается на элемент, находящийся с элементом-приемником в определенных отношениях. Это свойство имеет тип RelativeSource, который представляет собой класс расширения разметки. Приведем несколько примеров использования RelativeSource.

Чтобы сделать элемент-источник равным элементу-приемнику:

```
{Binding RelativeSource={RelativeSource Self}}
```

Чтобы сделать элемент-источник равным свойству TemplatedParent элемента-приемника (это свойство обсуждается в следующей главе):

```
{Binding RelativeSource={RelativeSource TemplatedParent}}
```

Чтобы сделать элемент-источник равным ближайшему родителю элемента-приемника, имеющему заданный тип:

```
{Binding RelativeSource={RelativeSource FindAncestor,
  AncestorType={x:Type desiredType}}}
```

Чтобы сделать элемент-источник равным *n*-му ближайшему родителю элемента-приемника, имеющему заданный тип:

```
{Binding RelativeSource={RelativeSource FindAncestor,
  AncestorLevel=n, AncestorType={x:Type desiredType}}}
```

Чтобы сделать элемент-источник равным предыдущему объекту в коллекции, привязанной к данным:

```
{Binding RelativeSource={RelativeSource PreviousData}}
```

Особенно полезно свойство RelativeSource в контексте шаблонов элементов управления, которые мы будем обсуждать в следующей главе. Однако использование RelativeSource в режиме Self удобно для привязки одного свойства элемента к другому его же свойству без указания имени элемента. Интересным примером может служить элемент Slider, свойство ToolTip которого привязано к его же текущему значению:

```
<Slider ToolTip="{Binding RelativeSource={RelativeSource Self}, Path=Value}"/>
```

Привязка к обычным свойствам .NET

Пример элементов TreeView и TextBox работает, потому что оба свойства – источник и приемник – являются свойствами зависимости. В главе 3 упоминалось, что в механизм свойств зависимости встроена возможность получать уведомления об изменениях. Именно это и позволяет WPF синхронизировать значения свойства-источника и свойства-приемника.

Однако WPF поддерживает использование любого свойства любого объекта .NET в качестве источника привязки к данным. Допустим, к примеру, что нужно добавить в приложение Photo Gallery элемент Label, в котором будет отображаться количество фотографий в текущей папке. Вместо того чтобы вручную записывать в метку свойство Count коллекции фотографий (типа Photos), можно привязать свойство Content метки к свойству Count коллекции:

```
<Label x:Name="numItemsLabel"
  Content="{Binding Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

(Здесь предполагается, что коллекция определена как ресурс, поэтому на нее можно сослаться из XAML с помощью свойства Source. Свойство ElementName здесь не подойдет, потому что эта коллекция не является объектом класса FrameworkElement или FrameworkContentElement!) На рис. 13.1 показан результат такого добавления. Отметим, что в метке отображается только строка «54», хотя должно быть что-то вроде «54 item(s)». Это можно исправить, поместив рядом метку со статическим текстом «item(s)». Но есть и более удачные решения, которые мы рассмотрим ниже в этой главе.

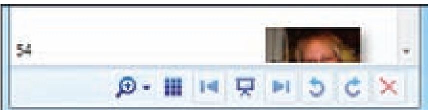

Рис. 13.1. *Отображение в левом нижнем углу главного окна значения photos.Count с помощью механизма привязки к данным*

Однако на пути использования обычного свойства .NET в качестве источника привязки нас подстерегает неприятность. Поскольку в таких свойствах не предусмотрены средства уведомления об изменениях, то приемник не будет автоматически синхронизироваться с источником – для этого нужна дополнительная работа. Таким образом, значение счетчика, показанное на рис. 13.1, не изменяется при смене папки, что, очевидно, неправильно.

Чтобы синхронизировать источник с приемником, объект-источник должен сделать одно из двух:

- Реализовать интерфейс System.ComponentModel.INotifyPropertyChanged, в котором определено единственное событие PropertyChanged.
- Реализовать событие *XXX*Changed, где *XXX* – имя свойства, значение которого изменяется.

Рекомендуется первое решение, поскольку WPF оптимизирована под него. (События вида *XXX*Changed поддерживаются в WPF только ради обратной совместимости со старыми классами.) Приложение Photo Gallery можно модифицировать, реализовав в коллекции photos интерфейс INotifyPropertyChanged. Для этого следует перехватывать все операции изменения (Add, Remove, Clear, Insert) и генерировать событие PropertyChanged. К счастью, в .NET Framework уже имеется класс, который делает эту работу за вас! Он называется ObservableCollection. Таким образом, для синхронизации привязки к photos.Count достаточно заменить в исходном коде строку

```
public class Photos : Collection<Photo>
```

на такую:

```
public class Photos : ObservableCollection<Photo>
```

КОПНЕМ ГЛУБЖЕ

Как работает привязка к обычным свойствам .NET

Чтобы получить значение свойства-источника, которое является обычным свойством .NET, WPF применяет отражение. Если объект-источник реализует интерфейс ICustomTypeDescriptor, то WPF его и использует (или, более общими словами, любой объект типа TypeDescriptionProvider, зарегистрированный конкретно для данного объекта или для его типа), чтобы узнать, какой дескриптор PropertyDescriptor применять для отражения. Реализация этого интерфейса – дело непростое, но полезное, когда нужно повысить производительность или поддержать нетривиальные ситуации (например, «на лету» изменять состав раскрываемых свойств).

ПРЕДУПРЕЖДЕНИЕ

Источники и приемники данных обрабатываются по-разному!

Свойством-источником действительно может быть любое свойство любого объекта .NET, однако для свойства-приемника это уже не так. Свойство-приемник *обязано* быть свойством зависимости. Отметим также, что источник должен быть настоящим (и притом открытым) свойством, а не просто полем.

Привязка ко всему объекту

Во всех приведенных до сих пор примерах использовались объекты-источники и свойства-источники, но оказывается, что свойство-источник (то есть Path в объекте Binding) необязательно! Можно привязать свойство-приемник ко всему объекту.

Но что это означает? На рис. 13.2 показано, как выглядела бы метка на рис. 13.1, если бы мы опустили задание Path:

```
<Label x:Name="numItemsLabel"
  Content="{Binding Source={StaticResource photos}}"
  DockPanel.Dock="Bottom"/>
```

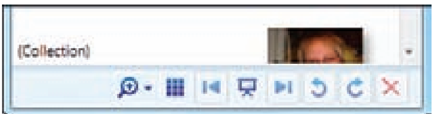

***Рис. 13.2.** Отображение в левом нижнем углу главного окна всего объекта photos с помощью механизма привязки к данным*

Поскольку класс объекта `photos` не является производным от `UIElement`, то он визуализируется в виде строки, которую возвращает метод `ToString`. В данном случае привязка ко всему объекту не очень полезна, но она крайне важна для элементов, способных распорядиться объектом более удачно, как, например, `ListBox`, который мы рассмотрим ниже.

СОВЕТ

Привязка ко всему объекту удобна, когда нужно в XAML-коде задать некое свойство, требующее объекта, который нельзя получить ни с помощью конвертера типа, ни посредством расширения разметки.

Например, в программе Photo Gallery есть элемент `Popup`, который центрируется над кнопкой `zoomButton`. Для этого мы устанавливаем свойства `Placement` и `PlacementTarget` объекта `Popup`, причем значением последнего должен быть объект типа `UIElement`. На C# это делается легко:

```
Button zoomButton = new Button();
...
Popup zoomPopup = new Popup();
zoomPopup.Placement = PlacementMode.Center;
zoomPopup.PlacementTarget = zoomButton;
```

Но в Photo Gallery для решения этой задачи применяется XAML-код:

```
<Button x:Name="zoomButton" ... >
  ...
</Button>
<Popup PlacementTarget="{Binding ElementName=zoomButton}" Placement="Center" ...>
   ...
</Popup>
```

Мы неоднократно пользовались этой техникой в предыдущих главах. Разумеется, расширение разметки `x:Reference`, появившееся в WPF 4, позволяет выполнить такое присваивание и без `Binding`.

ПРЕДУПРЕЖДЕНИЕ

Будьте осторожны с привязкой к объекту типа UIElement!

Привязывая некоторые свойства-приемники ко всему объекту UIElement, вы можете, сами того не желая, попытаться поместить один и тот же элемент в разные места визуального дерева. Например, следующий XAML-код приведет к исключению InvalidOperationException с сообщением "Specified element is already the logical child of another element" (Указанный элемент уже является логическим дочерним для другого элемента):

```
<Label x:Name="one" Content="{Binding ElementName=two}"/>
<Label x:Name="two" Content="text"/>
```

Однако исключения не будет, если заменить первый элемент Label на TextBlock (и, следовательно, свойство Content — на Text):

```
<TextBlock x:Name="one" Text="{Binding ElementName=two}"/>
<Label x:Name="two" Content="text"/>
```

Дело в том, что свойство Label.Content имеет тип Object, а свойство TextBlock.Text — строка. Поэтому, когда метке присваивается строковое значение, выполняется преобразование типа и вызывается метод ToString. В данном случае в текстовом блоке отображается строка "System.Windows.Controls.Label: *text*", по-прежнему не слишком полезная. Чтобы скопировать текст из одного элемента Label или TextBlock в другой, необходимо осуществить привязку к конкретному свойству (Label или Content).

Привязка к коллекции

Привязка метки к свойству photos.Count — вещь хорошая, но еще лучше было бы привязать список ListBox (основной элемент пользовательского интерфейса в окне Window) к коллекции photos. Эта часть программы Photo Gallery прямо-таки напрашивается на использование привязки к данным. В предыдущих версиях этого приложения связь между коллекцией фотографий, представленных в ListBox, и физическими фотографиями на диске, поддерживалась вручную. При выборе нового каталога программа очищала ListBox и создавала новые объекты ListBoxItem для каждой фотографии. Если пользователь удалял или переименовывал фотографию, то генерировалось событие от коллекции-источника (поскольку в ее реализации используется объект FileSystemWatcher), и его обработчик «вручную» обновлял содержимое списка ListBox.

К счастью, процедура замены этой логики привязкой к данным ничем не отличается от только что показанной.

Непосредственная привязка

Самым правильным было бы создать привязку Binding, задав в качестве свойства-приемника ListBox.Items, но, увы, Items не является свойством зависимости. Однако и ListBox, и все прочие многодетные элементы управления имеют свойство зависимости ItemsSource, специально предназначенное для привязки

к данным. Это свойство имеет тип IEnumerable, поэтому можно в качестве источника использовать весь объект photos и настроить Binding следующим образом:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
  ...
</ListBox>
```

Чтобы свойство-приемник синхронизировалось с изменениями в коллекции-источнике (то есть отражало добавление и удаление объектов), коллекция-источник должна реализовывать интерфейс INotifyCollectionChanged. Но в классе ObservableCollection реализованы оба интерфейса – INotifyPropertyChanged и InotifyCollectionChanged – так, что произведенного ранее изменения, когда мы унаследовали класс Photos от ObservableCollection<Photo>, достаточно, чтобы и эта привязка работала правильно.

На рис. 13.3 показан результат привязки.

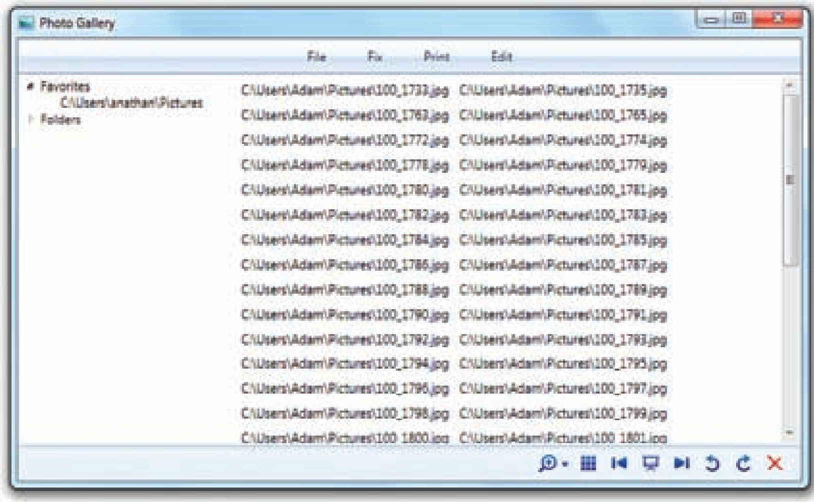

Рис. 13.3. *Если список ListBox привязан ко всему объекту photos, то данные отображаются в необработанном виде*

Улучшение отображения

Очевидно, что подразумеваемое по умолчанию отображение коллекции photos – результаты, возвращаемые методом ToString, – никуда не годится. Один из способов навести порядок – воспользоваться свойством DisplayMemberPath, которое имеется у всех многодетных элементов управления (см. главу 10). Оно работает рука об руку со свойством ItemsSource. Если записать в него путь к некоторому свойству, то для каждого объекта в списке будет отображаться значение именно этого свойства.

Коллекция в программе Photo Gallery состоит из объектов `Photo`, в которых есть такие свойства, как `Name`, `DateTime` и `Size`. Следовательно, показанная ниже XAML-разметка дает результаты, представленные на рис. 13.4, которые уже чуть лучше, чем то, что мы видим на рис. 13.3:

```
<ListBox x:Name="pictureBox" DisplayMemberPath="Name"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
  ...
</ListBox>
```

Но поскольку класс `Photo` определяли мы сами, то точно такого же результата можно было бы достичь, изменив реализацию его метода `ToString` так, чтобы он возвращал `Name` вместо полного пути.

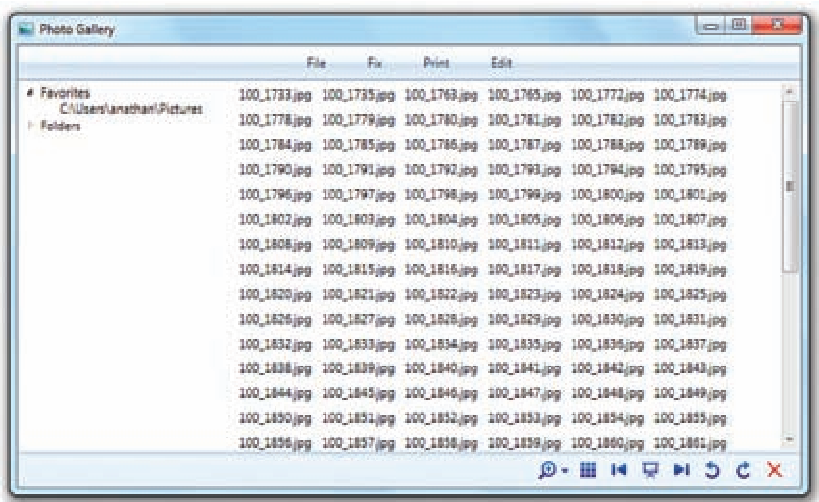

Рис. 13.4. *Свойство DisplayMemberPath дает простой механизм отображения объектов из коллекции, привязанной к данным*

Чтобы показывать в списке сами изображения, можно было бы добавить в класс `Photo` свойство `Image` и указать его в качестве `DisplayMemberPath`. Но есть и более гибкие способы контроля над представлением привязанных данных, которые не требуют вносить изменения в объект-источник. (Это существенно, потому что не всегда класс объекта-источника написан вами. Кроме того, не забывайте, что одна из основных целей WPF – отделение логики от внешнего вида!) Один такой способ (не относящийся собственно к механизму привязки к данным) – воспользоваться шаблоном данных, другой – применить конвертер значений. Ниже в разделе «Управление визуализацией» мы рассмотрим оба способа.

ПРЕДУПРЕЖДЕНИЕ

Свойства Items и ItemsSource объекта ItemsControl нельзя модифицировать одновременно!

Вы должны заранее решить, как будете заполнять многодетный элемент управления: вручную с помощью свойства Items или путем привязки к данным с помощью свойства ItemsSource, потому что смешивать эти два приема нельзя. Свойству ItemsSource можно присвоить значение только в тот момент, когда коллекция Items пуста, а свойство Items можно модифицировать, лишь если ItemsSource равно null (иначе вы получите исключение InvalidOperationException). Таким образом, если вы собираетесь добавлять объекты в привязанный к данным список ListBox или удалять из него объекты, то делать это следует только с помощью коллекции-источника (ItemsSource), а не на уровне пользовательского интерфейса (через свойство Items). Отметим, что вне зависимости от того, каким способом объекты *помещаются* в многодетный элемент управления, *обращаться* к ним для чтения всегда разрешается с помощью коллекции Items.

Управление выбранным объектом

В главе 10 было сказано, что для селекторов Selector, к числу которых относится и ListBox, определено понятие выбранного объекта или объектов. Когда селектор привязывается к коллекции (любому объекту, реализующему интерфейс IEnumerable), WPF отслеживает выбранные элементы, поэтому любой объект-приемник, привязанный к тому же источнику, может воспользоваться этой информацией, не требуя дополнительного кода. Эту возможность можно использовать для создания пользовательских интерфейсов вида главный/подчиненный (мы покажем, как это делается, в конце главы) или для синхронизации нескольких селекторов, чем мы сейчас и займемся.

Чтобы включить рассматриваемую поддержку, присвойте значение true свойству IsSynchronizedWithCurrentItem (которое наследуется всеми селекторами). В следующем XAML-коде это свойство установлено для трех списков ListBox, в каждом из которых отображается по одному свойству объекта из коллекции photos:

```
<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="Name"
  ItemsSource="{Binding Source={StaticResource photos}}"></ListBox>
<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="DateTime"
  ItemsSource="{Binding Source={StaticResource photos}}"></ListBox>
<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="Size"
  ItemsSource="{Binding Source={StaticResource photos}}"></ListBox>
```

Поскольку для каждого списка IsSynchronizedWithCurrentItem="True" и все они указывают на одну и ту же коллекцию-источник, то изменение выбранного объекта в любом списке приведет к аналогичному изменению в двух других. (Отметим, однако, что прокрутка списков автоматически не синхронизируется!) На рис. 13.5 показано, как это выглядит. Если в каком-то списке не задать свойство IsSynchronizedWithCurrentItem вообще или задать его равным false, то

изменение в нем выбранного объекта никак не отразится на двух других списках. Верно и обратное – изменение выбранного объекта в любом из двух других списков не влияет на выбранный объект в этом списке.

> **ПРЕДУПРЕЖДЕНИЕ**
>
> **Свойство IsSynchronizedWithCurrentItem не поддерживает множественный выбор!**
>
> Если в селекторе Selector выбрано несколько объектов (как в случае, когда свойство SelectionMode элемента ListBox равно Multiple или Extended), то все остальные синхронизированные с ним элементы увидят только первый выбранный объект, даже если сами поддерживают множественный выбор!

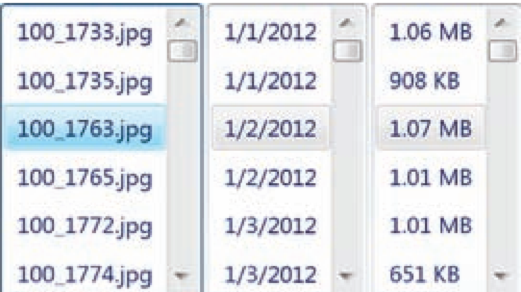

Рис. 13.5. *Благодаря привязке к данным все три списка синхронизированы*

Обобществление источника с помощью DataContext

На данный момент мы применили привязку к данным к нескольким свойствам-приемникам и для всех, кроме одного, указали один и тот же объект-источник (коллекцию photos). Вообще практика привязки нескольких элементов в одном пользовательском интерфейсе к общему объекту-источнику (*свойства*-источники могут быть разными, но *объект*-источник один) весьма распространена. Поэтому WPF поддерживает задание неявного источника данных вместо того, чтобы явно указывать в каждом элементе Binding свойства Source, RelativeSource или ElementName. Такой неявный источник данных называют также *контекстом данных*.

Чтобы назначить некоторый объект-источник, например коллекцию photos, контекстом данных, достаточно найти какой-нибудь общий родительский элемент и записать в его свойство DataContext ссылку на объект-источник. (Во всех классах, производных от FrameworkElement или FrameworkContentElement, имеется свойство DataContext типа Object.) Обнаружив элемент Binding, в котором объект-источник явно не задан, WPF поднимается вверх по логическому дереву, пока не найдет элемент с отличным от null свойством DataContext.

Поэтому, чтобы привязать элементы Label и ListBox к одному и тому же объекту-источнику, можно было бы установить DataContext следующим образом:

```
<StackPanel DataContext="{StaticResource photos}">
  <Label x:Name="numItemsLabel"
    Content="{Binding Path=Count}" .../>
  ...
  <ListBox x:Name="pictureBox" DisplayMemberPath="Name"
    ItemsSource="{Binding}" ...>
    ...
  </ListBox>
  ...
</StackPanel>
```

Поскольку DataContext – самое обычное свойство, его можно без труда установить и в процедурном коде, избежав тем самым необходимости сохранять объект-источник в качестве ресурса:

```
parent.DataContext = photos;
```

СОВЕТ

Увидев в XAML свойство, значением которого является просто строка {Binding}, легко прийти в недоумение, но это всего лишь означает, что объект-источник задан где-то выше в дереве в виде контекста данных и что привязка производится ко всему этому объекту, а не к отдельному его свойству.

FAQ

Когда следует задавать объект-источник в виде контекста данных, а когда явно в привязке Binding?

Вообще говоря, это дело вкуса. Если объект-источник используется только в одном свойстве-приемнике, то определение контекста данных – это перебор, да и разметка становится менее понятной. Если же объект-источник обобществляется, то контекст данных позволяет описать его в одном месте и, значит, уменьшить вероятность ошибки в случае смены источника.

Одна из ситуаций, когда контекст данных оказывается по-настоящему полезным, – подключение ресурсов, определенных где-то в другом месте. В таких ресурсах привязки Binding можно задавать без явного указания источника или контекста данных. Тем самым предполагается, что разрешение привязки произойдет в *контексте использования*, а не в *контексте объявления*. Контекстом использования будет то место в логическом дереве, куда помещен ресурс; и для каждого такого места может быть определен свой контекст данных. (Впрочем, такой же гибкости можно добиться за счет использования свойства RelativeSource для явного задания источника, определяемого относительным путем.)

Управление визуализацией

Привязка к данным не составляет труда, когда свойство-источник и свойство-приемник имеют совместимые типы и все, что нужно, – это получить визуализацию источника, подразумеваемую по умолчанию. Но очень часто требуется та или иная настройка. В предыдущих разделах необходимость такой настройки была очевидна – ведь нам нужно показывать в списке сами фотографии, а не текстовые строки!

Без привязки к данным такая настройка была бы простым делом, потому что весь код извлечения данных вы пишете сами (как в первоначальной версии Photo Gallery). Однако WPF предоставляет три разных способа получить и отобразить значение источника, поэтому не стоит отказываться от преимуществ привязки к данным, когда с первого взгляда неясно, как добиться желаемого результата в нестандартной ситуации. Вот эти способы: форматирование строк, шаблоны данных и конвертеры значений.

Форматирование строк

Когда результатом привязки к данным должно стать отображение строки, может оказаться полезным свойство `StringFormat` объекта `Binding`. Если оно задано, то WPF вызывает метод `String.Format`, передавая ему значение свойства `StringFormat` в первом аргументе (`format`) и неформатированный объект-приемник – во втором (`args[0]`). Таким образом, `{0}` в форматной строке будет относиться к объекту-приемнику, и вы можете применить все поддерживаемые спецификаторы формата, например `{0:C}` – для форматирования денежных величин, `{0:P}` – для представления в виде процента, `{0:X}` – для представления в шестнадцатеричном виде.

Таким образом, чтобы показать в метке на рис. 13.1 строку «54 item(s)», а не просто «54», нужно заменить ее элементом `TextBlock` и указать в элементе `Binding` простое свойство `StringFormat`:

```
<TextBlock x:Name="numItemsLabel"
  Text="{Binding StringFormat={}{0} item(s),
         Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

Неуместно выглядящие скобки `{}` в начале значения нужны для того, чтобы экранировать знак `{` в начале строки. Напомним (см. главу 2 «Все тайны XAML»), что без этого строка неправильно интерпретировалась бы как расширение разметки. Вставлять `{}` необязательно, если `Binding` используется в виде элемента (а не атрибута) свойства:

```
<TextBlock x:Name="numItemsLabel" DockPanel.Dock="Bottom">
  <TextBlock.Text>
    <Binding Source="{StaticResource photos}" Path="Count">
      <Binding.StringFormat>{0} item(s)</Binding.StringFormat>
    </Binding>
  </TextBlock.Text>
</TextBlock>
```

ПРЕДУПРЕЖДЕНИЕ

Свойство StringFormat работает, только если свойство-приемник — строка!

Основной недостаток подхода на основе свойства StringFormat заключается в том, что объект Binding попросту игнорирует это свойство, если тип свойства-приемника отличен от string. Попытка воспользоваться им для свойства Content элемента Label не даст никакого эффекта, потому что тип этого свойства – Object:

```
<Label x:Name="numItemsLabel"
  Content="{Binding StringFormat={}{0} item(s),
         Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

А вот свойство Text элемента TextBlock имеет тип string, поэтому та же привязка работает для него отлично. Именно поэтому в примерах данного раздела мы заменили Label на TextBlock. Другое обходное решение – воспользоваться свойством ContentStringFormat метки (обсуждается ниже).

Не нужны скобки и тогда, когда строка начинается с любого символа, кроме {:

```
<TextBlock x:Name="numItemsLabel"
  Text="{Binding StringFormat=Number of items: {0},
  Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

Можно также улучшить форматирование, добавив спецификатор N0, который расставляет разделители между группами по три цифры. Например, если Count равно 54, то будет показана строка «54 item(s)», а если Count равно 1001, то строка «1,001 item(s)» (по крайней мере, в культуре en-US):

```
<TextBlock x:Name="numItemsLabel"
  Text="{Binding StringFormat={}{0:N0} item(s),
         Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

ПРЕДУПРЕЖДЕНИЕ

System.Xaml обрабатывает управляющую последовательность {} некорректно!

В библиотеке *System.Xaml*, добавленной в WPF 4, есть ошибка, из-за которой управляющая последовательность {} внутри расширения разметки обрабатывается некорректно. При работе с *System.Xaml* последовательность {} можно использовать для экранирования *всего* строкового значения свойства (предотвращая ее интерпретацию как расширения разметки), но не *внутри* расширения разметки. Например, такой фрагмент XAML-кода разбирается библиотекой *System.Xaml* неправильно:

```
<TextBlock Text="{Binding StringFormat={}{0:C}}" />
```

К счастью, *System.Xaml* еще не применяется в наиболее распространенных ситуациях (например, при компиляции XAML-кода), так что влияние этой ошибки пока ощущается не очень сильно. Обходной путь состоит в том, чтобы в расширениях разметки использовать альтернативную управляющую последовательность, а именно экранировать отдельные символы, например:

```
<TextBlock Text="{Binding StringFormat=\{0:C\}}" />
```

У многих элементов управления имеется также свойство *XXX*StringFormat, где *XXX* представляет форматируемую часть. Например, у однодетных элементов есть свойство ContentStringFormat, которое применяется к свойству Content, а у многодетных — свойство ItemStringFormat, применяемое к каждому объекту в коллекции. В табл. 13.1 перечислены все свойства форматирования, допускающие чтение и запись.

Таблица 13.1. Свойства форматирования строк, имеющиеся в WPF

Свойство	Классы
StringFormat	BindingBase
ContentStringFormat	ContentControl, ContentPresenter, TabControl
ItemStringFormat	ItemsControl, HierarchicalDataTemplate
HeaderStringFormat	HeaderedContentControl, HeaderedItemsControl, DataGridColumn, GridViewColumn, GroupStyle
ColumnHeaderStringFormat	GridView, GridViewHeaderRowPresenter

Вместо замены Label на TextBlock, чтобы иметь возможность воспользоваться свойством StringFormat объекта Binding, можно было бы прибегнуть к собственному свойству элемента Label — ContentStringFormat, поскольку Label — однодетный элемент управления:

```
<Label x:Name="numItemsLabel" ContentStringFormat="{}{0} item(s)"
  Content="{Binding Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

Этот механизм работает и безотносительно к привязке к данным. На рис. 13.6 показан результат визуализации следующего списка ListBox для двух языков: американского диалекта английского и корейского:

```
<ListBox ItemStringFormat="{}{0:C}"
  xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <sys:Int32>-9</sys:Int32>
  <sys:Int32>9</sys:Int32>
  <sys:Int32>1234</sys:Int32>
  <sys:Int32>1234567</sys:Int32>
</ListBox>
```

Английский (USA)

($9.00)
$9.00
$1,234.00
$1,234,567.00

Корейский

-₩9
₩9
₩1,234
₩1,234,567

Рис. 13.6. *Числа в списке ListBox представлены в соответствии с декларативно заданным форматированием строки*

Шаблоны данных

Шаблон данных – это часть пользовательского интерфейса, которую можно применять к произвольному объекту .NET на этапе его визуализации. У многих элементов управления WPF имеются специальные свойства (типа DataTemplate) для присоединения шаблона данных. Например, в классе ContentControl есть свойство ContentTemplate для управления визуализацией объекта содержимого Content, а в классе ItemsControl – свойство ItemTemplate, применяемое к каждому объекту хранимой коллекции. Все подобные свойства перечислены в табл. 13.2. Как видите, свойств вида *XXX*Template больше, чем *XXX*StringFormat.

Таблица 13.2. Свойства типа DataTemplate

Свойство	Классы
ContentTemplate	ContentControl, ContentPresenter, TabControl
ItemTemplate	ItemsControl, HierarchicalDataTemplate
HeaderTemplate	HeaderedContentControl, HeaderedItemsControl, DataGridRow, DataGridColumn, GridViewColumn, GroupStyle
SelectedContentTemplate	TabControl
DetailsTemplate	DataGridRow
RowDetailsTemplate	DataGrid
RowHeaderTemplate	DataGrid
ColumnHeaderTemplate	GridView, GridViewHeaderRowPresenter
CellTemplate	DataGridTemplateColumn, GridViewColumn
CellEditingTemplate	DataGridTemplateColumn

Поместив в любое из этих свойств ссылку на объект типа DataTemplate, можно подключить совершенно новое визуальное дерево. Класс DataTemplate, как и класс ItemsPanelTemplate, описанный в главе 10, является производным от FrameworkTemplate. Поэтому в нем есть свойство содержимого VisualTree, в которое можно записать произвольное дерево элементов FrameworkElement. Это легко делается в XAML, но довольно неуклюже в процедурном коде.

Давайте применим шаблон DataTemplate к списку ListBox в программе Photo Gallery. На рис. 13.4 в этом списке отображались строки вместо изображений. В следующем фрагменте мы добавляем простой шаблон данных путем задания свойства списка ItemTemplate:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
<ListBox.ItemTemplate>
  <DataTemplate>
    <Image Source="placeholder.jpg" Height="35"/>
  </DataTemplate>
</ListBox.ItemTemplate>
  ...
</ListBox>
```

На рис. 13.7 видно, что начало положено. Правда, пока вместо наших фотографий показывается иллюстрация-заглушка *placeholder.jpg*, но все-таки это уже изображения!

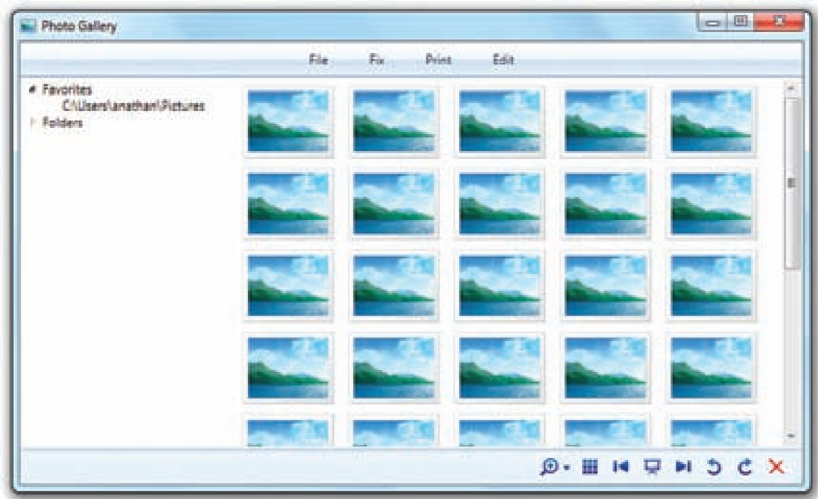

Рис. 13.7. *Простой шаблон данных приводит к показу иллюстрации-заглушки во всех позициях списка*

Итак, мы поместили шаблон Image в нужное место, но как записать в его свойство Source значение свойства FullPath текущего объекта Photo? Разумеется, с помощью привязки к данным! С каждым шаблоном данных неявно ассоциируется контекст данных (то есть объект-источник). Когда шаблон применяется в качестве ItemTemplate, контекстом данных будет текущий объект в источнике ItemsSource. Поэтому для получения результата, показанного на рис. 13.8, достаточно изменить шаблон данных следующим образом:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
```

```
<ListBox.ItemTemplate>
  <DataTemplate>
    <Image Source="{Binding Path=FullPath}" Height="35"/>
  </DataTemplate>
</ListBox.ItemTemplate>
  ...
</ListBox>
```

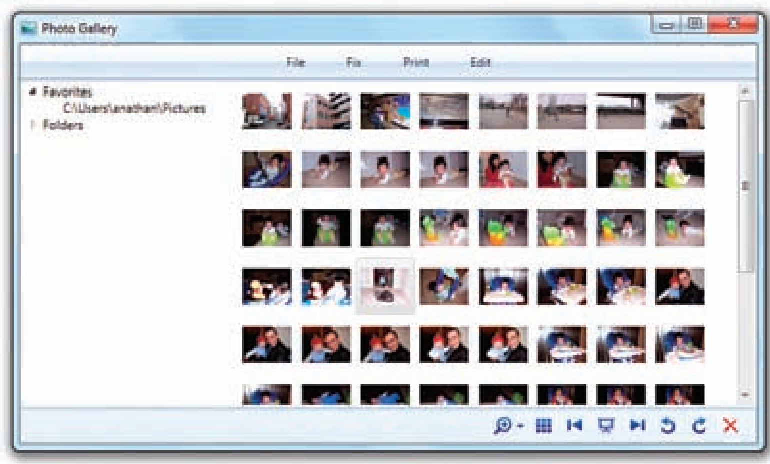

Рис. 13.8. *После модификации шаблон данных дает желаемый результат – в каждой позиции списка отображается нужная фотография*

Разумеется, шаблон данных необязательно определять в том месте, где он используется. Чаще всего шаблоны данных хранятся в виде ресурсов, разделяемых несколькими элементами. Можно даже сделать так, что шаблон данных будет автоматически применяться к любому экземпляру конкретного типа – для этого нужно лишь записать имя этого типа в свойство `DataType` шаблона. Если, к примеру, поместить такой объект `DataTemplate` в коллекцию `Resources` окна `Window`, то он автоматически будет применяться к любому элементу указанного типа, визуализируемому внутри окна, независимо от того, где он находится: в многодетном элементе управления, в однодетном или где-то еще. Если же поместить такой объект `DataTemplate` в коллекцию `Resources` объекта `Application`, то он будет относиться ко всему приложению в целом.

СОВЕТ

Хотя шаблоны данных можно использовать и с объектами, не привязанными к данным (например, со списком `ListBox`, который заполнялся вручную), почти всегда желательно, чтобы *внутри* шаблона привязка к данным была – именно это позволяет изменять визуальное дерево в соответствии с отображаемыми объектами.

Существует специальный подкласс DataTemplate для работы с иерархически организованными данными, например с XML-документами или с файловой системой, — HierarchicalDataTemplate. Он позволяет не только изменять представление таких данных, но и напрямую привязывать иерархию объектов к элементу, который умеет работать с иерархиями, например TreeView или Menu. Ниже в разделе «Класс XmlDataProvider» мы приведем пример использования HierarchicalDataTemplate с XML-данными.

КОПНЕМ ГЛУБЖЕ

Селекторы шаблонов

Иногда желательно выполнить глубокую настройку шаблона данных в зависимости от входных данных. Хотя многое можно сделать в пределах одного лишь шаблона данных, WPF предоставляет также механизм, позволяющий подключить процедурный код, который может выбрать любой шаблон (или создать новый «на лету») во время выполнения программы, когда возникает необходимость визуализировать данные. Для этого нужно создать класс, производный от DataTemplateSelector, и переопределить в нем виртуальный метод SelectTemplate. Затем следует ассоциировать экземпляр этого класса с интересующим вас элементом, установив в этом элементе свойство *XXX*TemplateSelector. Для каждого свойства *XXX*Template, показанного в табл. 13.2, существует соответствующее ему свойство *XXX*TemplateSelector (табл. 13.3).

Таблица 13.3. Свойства, относящиеся к селекторам шаблона данных

Свойство	Классы
ContentTemplateSelector	ContentControl, ContentPresenter, TabControl
ItemTemplateSelector	ItemsControl, HierarchicalDataTemplate
HeaderTemplateSelector	HeaderedContentControl, HeaderedItemsControl, DataGridRow, DataGridColumn, GridViewColumn, GroupStyle
SelectedContentTemplateSelector	TabControl
DetailsTemplateSelector	DataGridRow
RowDetailsTemplateSelector	DataGrid
RowHeaderTemplateSelector	DataGrid
ColumnHeaderTemplateSelector	GridView, GridViewHeaderRowPresenter
CellTemplateSelector	DataGridTemplateColumn, GridViewColumn
CellEditingTemplateSelector	DataGridTemplateColumn

Конвертеры значений

Если шаблоны данных позволяют изменить способ визуализации значений в свойстве-приемнике, то конвертеры значений предназначены для преобра-

зования значения, полученного из источника, в нечто совершенно иное перед доставкой приемнику. С их помощью можно подключить собственный код, не отказываясь от преимуществ привязки к данным.

Конвертеры значений часто применяются для того, чтобы согласовать различные типы данных в источнике и приемнике. Например, можно изменить цвет текста или фона элемента в зависимости от значения источника, тип данных которого отличен от Brush, – примерно так же, как это делается в функции условного форматирования в Microsoft Excel. Или просто расширить отображаемую информацию, не вводя отдельных элементов. В следующих двух разделах мы приведем примеры и того и другого.

Согласование несовместимых типов данных

Допустим, вы задумали менять цвет фона метки (свойство Background) в зависимости от количества фотографий в коллекции photos (значение свойства Count). Следующая привязка не имеет смысла, так как пытается присвоить свойству Background число, а не ожидаемый объект Brush:

```
<Label Background="{Binding Path=Count, Source={StaticResource photos}}" .../>
```

Дело можно поправить, подключив конвертер значений с помощью свойства Converter:

```
<Label Background="{Binding Path=Count, Converter={StaticResource myConverter},
  Source={StaticResource photos}}" .../>
```

Здесь предполагается, что вы написали класс, умеющий преобразовывать целое число в кисть Brush, и включили его в состав ресурсов:

```
<Window.Resources>
  <local:CountToBackgroundConverter x:Key="myConverter"/>
</Window.Resources>
```

Чтобы создать такой класс CountToBackgroundConverter, необходимо реализовать простой интерфейс IValueConverter (определен в пространстве имен System.Windows.Data). В этом интерфейсе есть всего два метода: Convert, принимающий объект-источник, который нужно преобразовать в объект-приемник, и ConvertBack, выполняющий обратную операцию.

Таким образом, на языке C# класс CountToBackgroundConverter можно написать так:

```
public class CountToBackgroundConverter : IValueConverter
{
   public object Convert(object value, Type targetType,
     object parameter, CultureInfo culture)
  {
    if (targetType != typeof(Brush))
      throw new InvalidOperationException("The target must be a Brush!");

    // Если формат входных данных недопустим, то Parse возбудит исключение
    int num = int.Parse(value.ToString());
```

```
        return (num == 0 ? Brushes.Yellow : Brushes.Transparent);
      }

      public object ConvertBack(object value, Type targetType,
        object parameter, CultureInfo culture)
      {
        return DependencyProperty.UnsetValue;
      }
    }
```

Метод Convert вызывается при каждом изменении значения источника. Ему передается целочисленное значение, а в ответ он возвращает объект Brushes.Yellow, если это значение равно нулю, и Brushes.Transparent — в противном случае. (Идея в том, чтобы выделить метку цветом, когда отображаемая папка пуста.) Метод ConvertBack нам не нужен, поэтому мы просто возвращаем фиктивное значение. В части VI «Дополнительные вопросы» мы рассмотрим ситуации, когда возникает необходимость в методе ConvertBack. На рис. 13.9 показан результат работы конвертера значений.

Рис. 13.9. *Конвертер значений делает желтым цвет фона метки в левом нижнем углу главного окна, когда в коллекции photos нет ни одной фотографии*

СОВЕТ

Во избежание недоразумений рекомендуется отражать семантику конвертера значений в его названии. Я мог бы назвать описанный выше класс не CountToBackgroundConverter, а, скажем, IntegerToBrushConverter, потому что технически его можно использовать всюду, где тип данных источника – целое число, а тип данных приемника – Brush. Однако это имеет смысл, только если исходное целое число представляет количество объектов, а кисть – цвет фона. (Например, маловероятно, что кто-нибудь захочет присвоить свойству элемента Foreground (цвет текста) значение Transparent (прозрачный)!) Кроме того, возможно, вам понадобится определить дополнительные конвертеры целого в кисть с иной семантикой.

Методам интерфейса IValueConverter передаются аргументы parameter и culture. По умолчанию parameter равен null, а culture — значению свойства Language элемента-приемника. Это свойство (определенное в классах FrameworkElement и FrameworkContentElement, где часто наследуется от корневого элемента, если вообще задается) по умолчанию равно "en-US" (американский диалект английского языка). Однако пользователь привязки может управлять обоими значениями с помощью свойств Binding.ConverterParameter и Binding.Converter-

Culture. Например, вместо того чтобы жестко зашивать цвет Brushes.Yellow в код метода CountToBackgroundConverter.Convert, можно было бы передавать его в параметре:

```
return (num == 0 ? parameter : Brushes.Transparent);
```

Здесь предполагается, что parameter всегда задается следующим образом:

```
<Label Background="{Binding Path=Count, Converter={StaticResource myConverter},
  ConverterParameter=Yellow, Source={StaticResource photos}}" Content="..." />
```

Присваивание свойству ConverterParameter простой строки "Yellow" работает, а вот почему работает – это тонкий вопрос. Как и все параметры расширения разметки, строка "Yellow" подвергается преобразованию типа, но только в тип свойства ConverterParameter (Object). Следовательно, метод Convert получает параметр в виде строки "Yellow", а не Brush. Поскольку Convert не делает со своим аргументом parameter ничего, кроме его возврата, когда num отлично от нуля, то в конечном итоге он возвращает строку. И вот в этот момент механизм привязки выполняет преобразование типа, чтобы присваивание свойству Background элемента Label завершилось нормально.

В свойство ConverterCulture можно записать любое обозначение языка, стандартизованное Инженерной группой по развитию Интернета (IETF), например "ko-KR", тогда конвертер получит соответствующий объект типа CultureInfo.

СОВЕТ

В состав WPF входит ряд конвертеров значений для нескольких очень распространенных сценариев привязки к данным. Один из них называется BooleanToVisibilityConverter и преобразует трехпозиционное перечисление Visibility (в нем определены значения Visible, Hidden, Collapsed) в тип Boolean, в том числе допускающий значение null (и обратно). В одном направлении true отображается на Visible, а false и null – на Collapsed. В другом направлении Visible отображается на true, а Hidden и Collapsed – на false.

Это бывает полезно для переключения видимости одного элемента в зависимости от состояния другого. Например, в следующей XAML-разметке реализован флажок Show Status Bar (Показывать строку состояния) вообще без процедурного кода:

```
<Window.Resources>
  <BooleanToVisibilityConverter x:Key="booltoVis"/>
</Window.Resources>
...
<CheckBox x:Name="checkBox">Show Status Bar</CheckBox>
...
<StatusBar Visibility="{Binding ElementName=checkBox, Path=IsChecked,
  Converter={StaticResource booltoVis}}">...</StatusBar>
```

В данном случае элемент StatusBar виден тогда и только тогда, когда свойство IsChecked флажка CheckBox равно true.

ПРЕДУПРЕЖДЕНИЕ

Ошибки привязки к данным не возбуждают исключений!

В случае когда во время привязки к данным обнаруживается ошибка, WPF не возбуждает исключение, а выводит пояснительный текст в трассировку отладки, которую можно видеть, только если к программе присоединен отладчик (или другой прослушиватель трассировки). Поэтому если привязка к данным работает не так, как вы ожидали, попробуйте запустить программу под отладчиком и не забудьте посмотреть на трассировку. В Visual Studio трассировка выводится в окно Output (Вывод). В Visual Studio 2010 Ultimate трассировку отладки можно также интегрировать в удобное окно IntelliTrace.

В предыдущем примере бессмысленной привязки (попытки привязать свойство Background непосредственно к photos.Count) выводится такая отладочная трассировка:

```
System.Windows.Data Error: 5 : Value produced by BindingExpression is not valid
for target property.; Value='39' BindingExpression:Path=Count; DataItem='Photos'
(HashCode=58961324); target element is 'Label' (Name='numItemsLabel'); target
property is 'Background' (type 'Brush')
```

Даже исключения, возбуждаемые объектом-источником (или конвертером значений) по умолчанию, «проглатываются» и отображаются в отладочной трассировке!

Поскольку трассировка реализована с помощью объектов System.Diagnostics.TraceSource, то существует несколько стандартных способов получить ту же трассу вне отладчика. Майк Хиллберг (Mike Hillberg), архитектор WPF, поделился подробностями в своем блоге по адресу *http://blogs.msdn.com/mikehillberg/archive/2006/09/14/WpfTraceSources.aspx*. Можно перехватывать трассировки, которые WPF генерирует в разных местах (некоторые из них по умолчанию не видны даже под отладчиком), например информацию о маршрутизации событий, о регистрации свойств зависимости, о поиске ресурсов и многом другом.

Можно также воспользоваться свойством PresentationTraceSources.TraceLevel (определено в пространстве имен System.Diagnostics в сборке WindowsBase), которое присоединяется к любому объекту Binding и позволяет увеличить или уменьшить объем информации, помещаемой им в трассировку. Это свойство может принимать значения из перечисления PresentationTraceLevel: None, Low, Medium, High.

Настройка отображения данных

Конвертеры значений иногда бывают полезны даже в случае, когда типы данных источника и приемника совместимы. Когда мы устанавливали свойство Content метки numItemsLabel равным свойству Count коллекции photos (см. рис. 13.1), оно отображалось нормально, но потребовался дополнительный текст, чтобы пользователю было понятно, что означает число. Мы решили эту проблему с помощью свойства StringFormat, но можно было бы добиться лучшего результата, чем отображение суффикса item(s). (Не знаю, как вам, а мне строки вида «1 item(s)» всегда кажутся свидетельством лени разработчика.)

Конвертер значений позволяет настроить текст в зависимости от значения, то есть выводить «1 item» (в единственном числе), но «2 items» (во множественном числе). Эту задачу решает класс `RawCountToDescriptionConverter`:

```
public class RawCountToDescriptionConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
    object parameter, CultureInfo culture)
  {
    // Если формат входных данных недопустим, то Parse возбудит исключение
    int num = int.Parse(value.ToString());
    return num + (num == 1 ? " item" : " items");
  }

  public object ConvertBack(object value, Type targetType,
    object parameter, CultureInfo culture)
  {
    return DependencyProperty.UnsetValue;
  }
}
```

Отметим, что здесь жестко зашиты англоязычные строки, тогда как в программе промышленного качества конвертер следовало бы сделать локализуемым ресурсом (или, по крайней мере, воспользоваться переданным параметром `culture`).

СОВЕТ

Конвертеры значений дают возможность подключить к процессу привязки к данным любую логику, выходящую за рамки простого форматирования. Нужно ли вам применить к значению-источнику некоторое преобразование перед отображением или изменить способ обновления приемника при изменении значения-источника – все это можно легко сделать с помощью класса, реализующего интерфейс `IValueConverter`.

СОВЕТ

Можно сделать так, что конвертер значений будет временно отменять привязку к данным, возвращая специальное значение `Binding.DoNothing`. Это не то же самое, что возврат `null`, поскольку `null` может быть вполне допустимым значением свойства-приемника.

`Binding.DoNothing` означает следующее: «Я не хочу сейчас ничего привязывать, сделай вид, что объекта `Binding` не существует». В таком случае значение свойства-приемника не изменяется, если только не найдется другого влияющего на него компонента (анимации, триггера и т. д.). Возврат `Binding.DoNothing` распространяется только на данный вызов `Convert` или `ConvertBack`, так что если объект `Binding` не уничтожен (например, обращением к методу `ClearBinding`), то конвертер будет и дальше вызываться при каждом изменении значения-источника.