ГЛАВА 8

Привязка элементов

В своей простейшей форме привязка данных — это отношение, которое сообщает WPF о необходимости извлечения некоторой информации из *исходного* объекта и использования его для установки свойства в *целевом* объекте. Целевое свойство — всегда свойство зависимости, и обычно оно находится в элементе WPF: в конце концов, конечной целью привязки данных WPF является отображение некоторой информации в пользовательском интерфейсе. Однако объектом-источником может быть почти что угодно, от другого элемента WPF до объекта ADO.NET (подобного `DataTable` и `DataRow`) или созданного вами объекта, хранящего только данные.

В этой главе рассмотрение привязки данных начинается с простейшего подхода: привязка элемента к элементу. В главе 19 мы вернемся к теме привязки данных и ознакомимся с наиболее эффективным способом передачи информации из базы данных в формы данных.

Связывание элементов вместе

Простейший сценарий привязки данных подразумевает ситуацию, когда исходным объектом является элемент WPF, а исходным свойством — свойство зависимости. Причина в том, что свойство зависимости имеет встроенную поддержку уведомлений об изменениях, как было описано в главе 4. В результате, когда значение свойства зависимости изменяется в исходном объекте, привязанное свойство целевого объекта немедленно обновляется. Это именно то, что требуется, и происходит оно без необходимости построения любой дополнительной инфраструктуры.

На заметку! Хотя привязка элемента к элементу является простейшим подходом, большинство разработчиков заинтересовано в нахождении самого общего подхода для реальных приложений. В общем, большая часть работы по привязке данных будет тратиться на привязку элементов к объектам данных. Это позволит отображать информацию, извлекаемую из внешнего источника (такого как база данных или файл). Однако привязка элемента к элементу также часто бывает полезной. Например, ее можно использовать для автоматизации способа, которым элементы взаимодействуют, так что когда пользователь модифицирует один элемент управления, другой элемент обновляется автоматически. Это ценное сокращение, которое избавляет от написания громоздкого и рутинного кода (и это прием, не доступный в предыдущем поколении приложений Windows Forns).

Чтобы понять, как привязывать один элемент к другому, рассмотрим простое окно, показанное на рис. 8.1. Оно содержит два элемента управления: `Slider` (ползунок) и `TextBlock` (текстовый блок) с единственной строкой текста. Перемещение ползунка вправо приводит к немедленному увеличению размера шрифта текста, а перемещение влево — к уменьшению размера шрифта.

Ясно, что такое поведение нетрудно реализовать и в коде. Понадобилось бы просто реагировать на событие `Slider.ValueChanged` и копировать текущее значение из ползунка в `TextBlock`. Однако привязка данных делает это еще проще.

Совет. Привязка данных обладает еще одним преимуществом: она позволяет создавать простые страницы XAML, которые можно выполнять в браузере без компиляции их в приложения. (Как известно из главы 2, если файл XAML имеет связанный файл отделенного кода, он не может быть открыт в браузере.)

Выражение привязки

При использовании привязки данных никаких изменений в исходный объект (в рассматриваемом примере это `Slider`) вносить не понадобится. Просто задайте нужный диапазон значений, как это делается обычно:

```
<Slider Name="sliderFontSize" Margin="3"
 Minimum="1" Maximum="40" Value="10"
 TickFrequency="1" TickPlacement="TopLeft">
</Slider>
```

Привязка определена в элементе `TextBlock`. Вместо установки `FontSize` с использованием литерального значения применяется выражение привязки:

```
<TextBlock Margin="10" Text="Simple Text" Name="lblSampleText"
 FontSize="{Binding ElementName=sliderFontSize, Path=Value}" >
</TextBlock>
```

Выражения привязки данных используют расширение разметки XAML (и потому помещаются в круглые скобки). В начале идет слово `Binding`, потому что создается экземпляр класса `System.Windows.Data.Binding`. Хотя объект `Binding` может быть сконфигурирован различными способами, в данной ситуации необходимо установить только два свойства: `ElementName`, которое указывает исходный элемент, и `Path`, указывающее свойство в исходном элементе.

Вместо `Property` используется `Path`, потому что `Path` может указывать на свойство свойства (например, `FontFamily.Source`) или индексатор, используемый свойством (например, `Content.Children[0]`). Путь можете включать множество фрагментов, переходящих от свойства к свойству, и т.д.

Чтобы сослаться на присоединенное свойство (свойство, которое определено в другом классе, но применяется к привязанному элементу), имя свойства должно быть указано в круглых скобках. Например, в случае привязки к элементу, помещенному в `Grid`, путь `(Grid.Row)` извлекает номер строки, в которой он находится.

Ошибки привязки

WPF не генерирует исключения для уведомления о проблемах привязки данных. Если указан несуществующий элемент или свойство, никакого сообщения об этом не будет; вместо этого данные просто не попадут в целевое свойство.

На первый взгляд это может показаться кошмаром для отладки. К счастью, WPF выводит трассировочную информацию, которая детализирует сбои в привязке. Во время отладки приложения эта информация появляется в выходном окне Visual Studio. Например, попытка привязки к несуществующему свойству приводит к выводу в выходное окно следующего сообщения:

```
System.Windows.Data Error: 35 : BindingExpression path error:
 'Tex' property not found on 'object' ''TextBox' (Name='txtFontSize')'.
 BindingExpression:Path=Tex; DataItem='TextBox' (Name='txtFontSize');
```

```
target element is 'TextBox' (Name='');
target property is 'Text' (type 'String')
```

```
System.Windows.Data Ошибка: 35 : ошибка пути BindingExpression:
 Свойство 'Tex' не найдено в 'объекте' ''TextBox' (Name='txtFontSize')'.
 BindingExpression:Path=Tex; DataItem='TextBox' (Name='txtFontSize');
 целевой элемент - 'TextBox' (Name='');
 целевое свойство - 'Text' (типа 'String')
```

Среда WPF также игнорирует любые исключения, которые генерируются при попытке читать исходное свойство, и молча поглощает исключение, возникающее, если исходные данные не могут быть приведены к типу данных целевого свойства. Однако есть и другой способ справиться с этой проблемой — можно сообщить WPF о необходимости изменения внешнего вида исходного элемента для индикации возникшей ошибки. Например, неверный ввод можно пометить значком с восклицательным знаком или рамкой красного цвета. Более подробно проверка достоверности рассматривается в главе 19.

Режимы привязки

Одним из ценных особенностей привязки данных является то, что цель обновляется автоматически, независимо от того, как модифицируется источник. В этом примере источник может быть модифицирован только в одном направлении — через взаимодействие пользователя с ползунком. Однако рассмотрим несколько усложненную версию этого примера, в которой добавляется несколько кнопок, причем каждая из них применяет предварительно установленное значение ползунка. Новое окно показано на рис. 8.2.

В результате щелчка на кнопке Set to Large (Установить крупный) запускается следующий код:

```
private void cmd_SetLarge(object sender, RoutedEventArgs e)
{
    sliderFontSize.Value = 30;
}
```

Этот код устанавливает значение ползунка, которое, в свою очередь, изменяет размер шрифта текста через привязку данных. Это то же самое, как если бы вы двигали ползунок вручную.

Тем не менее, следующий код не работает так хорошо:

```
private void cmd_SetLarge(object sender, RoutedEventArgs e)
{
    lblSampleText.FontSize = 30;
}
```

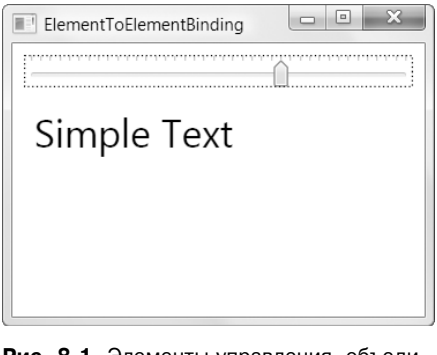

Рис. 8.1. Элементы управления, объединенные привязкой данных

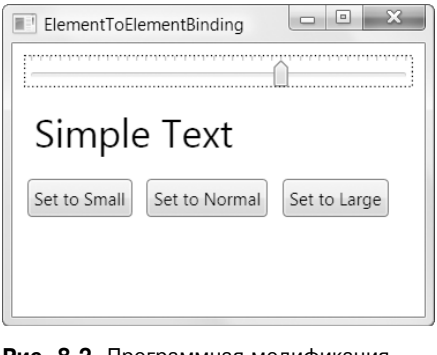

Рис. 8.2. Программная модификация источника данных

Он устанавливает шрифт текстового поля непосредственно. В результате позиция ползунка не приводится в соответствие с новым значением. Хуже того, это разрушает привязку размера шрифта и заменяет его литеральным значением. Теперь, если передвинуть ползунок, текстовое поле вообще не изменится.

Интересно, что существует способ заставить данные перемещаться в обоих направлениях: от источника к цели и от цели к источнику. Трюк заключается в установке свойства `Mode` объекта `Binding`. Ниже приведена усовершенствованная двунаправленная привязка, которая позволяет применять значения либо к источнику, либо к цели, и заставит противоположную часть привязки обновлять себя автоматически:

```
<TextBlock Margin="10" Text="Simple Text" Name="lblSampleText"
 FontSize="{Binding ElementName=sliderFontSize, Path=Value, Mode=TwoWay}" >
</TextBlock>
```

В рассматриваемом примере нет причин применять двунаправленную привязку (которая требует больше накладных расходов), поскольку эту проблему можно решить с помощью кода. Однако рассмотрим вариант этого примера, включающий текстовое поле, в котором пользователь может точно устанавливать размер шрифта. Этому текстовому полю понадобится двунаправленная привязка, чтобы оно могло как применять пользовательские изменения, так и отражать последнее значение размера, когда оно изменяется другим путем. Данный пример будет представлен в следующем разделе.

При установке свойства `Binding.Mode` можно использовать одно из пяти значений перечисления `System.Windows.Data.BindingMode`. В табл. 8.1 приведен их полный список.

Таблица 8.1. Значения из перечисления `BindingMode`

Имя	Описание
`OneWay`	Целевое свойство обновляется при изменениях исходного свойства
`TwoWay`	Целевое свойство обновляется при изменениях исходного свойства, а исходное свойство обновляется при изменении целевого свойства
`OneTime`	Целевое свойство устанавливается изначально на основе значения исходного свойства. Однако с этого момента изменения игнорируются (если только привязка не устанавливается на совершенно другой объект или не вызывается `BondingExpression.UpdateTarget()`, как описано далее в этой главе). Обычно этот режим используется для сокращения накладных расходов, если известно, что целевое свойство не изменится
`OneWayToSource`	Подобно `OneWay`, но действует в обратном направлении. Исходное свойство обновляется, когда изменяется целевое свойство (что может показаться несколько странным), но целевое свойство никогда не обновляется
`Default`	Этот тип привязки зависит от целевого свойства. Это либо `TwoWay` (для устанавливаемых пользователем свойств, таких как `TextBox.Text`), либо `OneWay` (для всего остального). Все привязки используют данный подход, если только не указано иное

На рис. 8.3 демонстрируется разница. Вы уже видели `OneWay` и `TwoWay`. Значение `OneTime` достаточно очевидно. Оставшиеся два варианта требуют ряда дополнительных исследований.

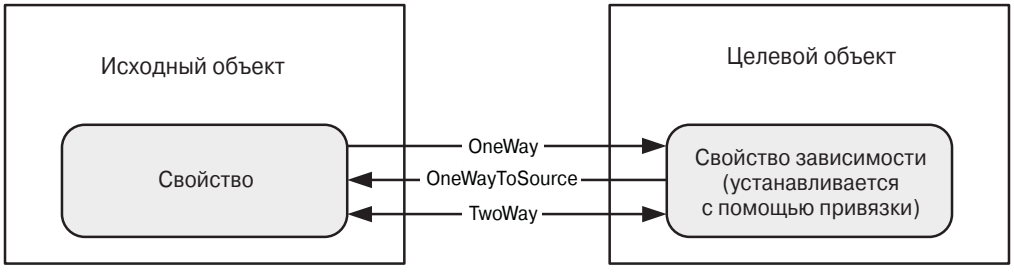

Рис. 8.3. Различные направления привязки свойств

Привязка OneWayToSource

Может возникнуть вопрос: зачем нужны две опции — и `OneWay`, и `OneWayToSource`? В конце концов, оба значения создают однонаправленную привязку, которая работает одинаковым образом. Единственное отличие в том, куда помещено выражение привязки. По сути, `OneWayToSource` позволяет поменять местами источник и цель, поместив выражение в то, что обычно считается источником привязки.

Наиболее общая причина использования этого трюка состоит в установке свойства, которое не является свойством зависимости. Как упоминалось в начале этой главы, выражения привязки могут применяться только для установки свойств зависимости. Однако с помощью `OneWayToSource` это ограничение можно обойти, предоставляя в качестве свойства, поставляющего значение, свойство зависимости.

Привязка Defaut

Изначально кажется логичным предположить, что все привязки однонаправленные, если только явно не указано иное. (В конце концов, именно так работает простой пример с ползунком.) Но на самом деле это не так. Чтобы убедиться в этом, вернемся к примеру с привязанным текстовым полем и позволим редактировать текущий размер шрифта. Если убрать установку `Mode=TwoWay`, этот пример все равно будет работать точно также. Причина в том, что WPF использует разные значения `Mode` по умолчанию, в зависимости от привязываемого свойства. (Формально в каждом свойстве зависимости присутствует фрагмент метаданных — флаг `FrameworkPropertyMetadata.BindsTwoWayByDefault`, который указывает, какую привязку должно использовать свойство: однонаправленную или двунаправленную).

Часто значение по умолчанию — именно то, что и нужно. Тем не менее, можно представить пример с текстовым полем только для чтения, которое пользователь не может изменять. В этом случае удается слегка сократить накладные расходы, установив режим однонаправленной привязки.

В качестве общего эмпирического правила: всегда неплохо явно устанавливать режим. Даже в случае текстового поля стоит подчеркнуть, что нужна двунаправленная привязка, включив свойство `Mode`.

Создание привязки в коде

При построении окна обычно наиболее эффективно объявлять выражение привязки в разметке XAML с помощью расширения разметки `Binding`. Тем не менее, допускается также создавать привязку и в коде.

Вот как можно создать привязку для элемента `TextBlock`, показанного в предыдущем примере:

```
Binding binding = new Binding();
binding.Source = sliderFontSize;
binding.Path = new PropertyPath("Value");
binding.Mode = BindingMode.TwoWay;
lblSampleText.SetBinding(TextBlock.FontSizeProperty, binding);
```

Для удаления привязки в коде предусмотрены два статических метода класса `BindingOperations`. Метод `ClearBinding()` принимает ссылку на свойство зависимости, которое имеет привязку, подлежащую удалению, а метод `ClearAllBindings()` удаляет все привязки данных элемента:

```
BindingOperations.ClearAllBindings(lblSampleText);
```

И `ClearBinding()`, и `ClearAllBindings()` используют метод `ClearValue()`, который каждый элемент наследует от базового класса `DependencuObject`. Метод `ClearValue()` просто удаляет локальное значение свойства (которым в данном случае является выражение привязки).

Привязка на основе разметки применяется намного чаще, чем программная привязка, потому что она яснее и требует меньше работы. В этой главе во всех примерах для создания привязок используется разметка. Однако код будет применяться для создания привязки в некоторых специализированных сценариях.

- *Создание динамических привязок.* Если необходимо тонко настраивать привязку на основе другой информации времени выполнения или создавать разные привязки в зависимости от обстоятельств, имеет смысл делать это в коде. (В качестве альтернативы можно было бы определить все необходимые привязки в коллекции `Resources` окна и просто добавить код, который вызывает `SetBinding()` с соответствующим объектом привязки.)
- *Удаление привязки.* Чтобы удалить привязку и получить возможность установки свойства обычным образом, понадобится помощь метода `ClearBinding()` или `ClearAllBindings()`. Недостаточно просто присвоить новое значение свойству. В случае использования двунаправленной привязки установленное значение распространится на привязанный объект, и оба свойства останутся синхронизированными.

На заметку! С помощью методов `ClearBinding()` и `ClearAllBindings()` можно удалить любую привязку. Не имеет значения, применялась привязка программно или в коде XAML.

- *Создание специальных элементов управления.* Чтобы облегчить для других модификацию визуального представления специального элемента управления, который разрабатывается, определенные детали (такие как обработчики событий и выражения привязки данных) понадобится перенести в код разметки. В главе 18 представлен пользовательский элемент управления для выбора цвета, который использует код для создания своих привязок.

Множественные привязки

Хотя в предыдущий пример включена только одиночная привязка, останавливаться на этом не обязательно. При желании можно заставить `TextBlock` отображать текст из текстового поля, брать текущие цвета фона и переднего плана из отдельного списка цветов и т.д. Например:

```
<TextBlock Margin="3" Name="lblSampleText"
  FontSize="{Binding ElementName=sliderFontSize, Path=Value}"
  Text="{Binding ElementName=txtContent, Path=Text}"
  Foreground="{Binding ElementName=lstColors, Path=SelectedItem.Tag}" >
</TextBlock>
```

На рис. 8.4 показан элемент TextBlock с тремя привязками.

Допускается также реализовать привязку данных. Например, можно создать выражение привязки для свойства TextBox.Text, связывающее его со свойством TextBlock.FontSize, которое содержит выражение привязки, связывающее со свойством Slider.Value. В этом случае, когда пользователь перетаскивает ползунок в новую позицию, значение передается от Slider в TextBlock и затем из TextBlock в TextBox. Хотя все работает прозрачно, более ясный подход состоит в том, чтобы привязать элементы как можно ближе к данным, которые они используют. В описанном здесь примере необходимо предусмотреть привязку и TextBlock, и TextBox непосредственно к свойству Slider.Value.

Все становится немного более интересно, когда на целевое свойство должны оказывать влияние более одного источника, например, если нужно иметь две равноправные привязки, устанавливающие одно и то же свойство. На первый взгляд это кажется невозможным. Однако существует несколько способов решения.

Простейший подход состоит в изменении режима привязки данных. Как уже известно, свойство Mode позволяет модифицировать способ работы привязки так, что данные передаются не только от источника к цели, но и от цели к источнику. С помощью такого приема можно создать несколько выражений привязки, устанавливающих одно и то же свойство. Последнее из них будет иметь эффект.

Чтобы понять, как это работает, рассмотрим вариацию примера элемента — панели с ползунком, который включает текстовое поле, куда можно поместить точное значение размера шрифта. В этом примере (рис. 8.5) свойство TextBlock.FontSize может быть установлено двумя путями: перетаскиванием ползунка или вводом в текстовом поле размера шрифта. Все элементы управления синхронизированы так, что если вводится новое число в текстовом поле, размер шрифта текста примера изменяется и ползунок перемещается в соответствующую позицию.

Как уже упоминалось, к свойству TextBlock.FontSize можно применять только одну привязку данных. Поэтому имеет смысл оставить свойство TextBlock.FontSize в том виде, как оно есть, чтобы оно привязывалось прямо к ползунку:

```
<TextBlock Margin="10" Text="Simple Text" Name="lblSampleText"
 FontSize="{Binding ElementName=sliderFontSize, Path=Value, Mode=TwoWay}" >
</TextBlock>
```

Рис. 8.4. Элемент TextBlock с тремя привязками

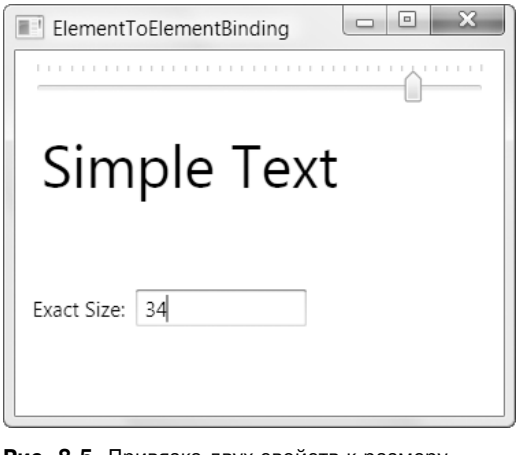

Рис. 8.5. Привязка двух свойств к размеру шрифта

Хотя добавить другую привязку к свойству FontSize нельзя, можно привязать новый элемент управления TextBox к свойству TextBlock.FontSize. Ниже показана необходимая для этого разметка:

```
<TextBox Text="{Binding ElementName=lblSampleText, Path=FontSize, Mode=TwoWay}">
</TextBox>
```

Теперь при каждом изменении свойства TextBlock.FontSize текущее значение будет вставляться в текстовое поле. Более того, значение в текстовом поле можно редактировать, применяя указанный размер шрифта. Обратите внимание, что для того, чтобы пример работал, свойство TextBlock.Text должно использовать двунаправленную привязку, передающую данные в обоих направлениях. В противном случае текстовое поле сможет отображать значение TextBlock.FontSize, но не позволит изменять его.

С этим примером связано несколько нюансов.

- Поскольку значение свойства Slider.Value имеет тип double, при перетаскивании ползунка получается дробное значение размера. Установив свойство TickFrequency в 1 (или в некоторый целочисленный интервал), В свойство InSnalToEnabled в true, можно ограничить значение ползунка только целыми величинами.
- Текстовое поле позволяет вводить буквы и другие нечисловые символы. В таком случае значение текстового поля не сможет быть интерпретировано как число. В результате привязка данных молча потерпит неудачу, а значение размера шрифта станет равно 0. Другой подход мог бы состоять в обработке нажатий клавиш в текстовом поле, чтобы вообще предотвратить неправильный ввод, либо в использовании проверки достоверности, как описано в главе 19.
- Изменения, которые вносятся в текстовое поле, не будут применены до тех пор, пока текстовое поле не потеряет фокус (например, когда с помощью клавиши <Tab> происходит переход на другой элемент управления). Если такое поведение не подходит, можно обеспечить непрерывное обновление, используя свойство UpdateSourceTrigger объекта Binding, как вы узнаете далее в разделе “Обновления привязок”.

Интересно, что показанное здесь решение — не единственный способ привязки текстового поля. Столь же разумно конфигурировать текстовое поле таким образом, чтобы оно изменяло значение свойства Slider.Value вместо свойства TextBlock.FontSize:

```
<TextBox Text="{Binding ElementName=sliderFontSize, Path=Value, Mode=TwoWay}">
</TextBox>
```

Теперь изменение текстового поля инициирует изменение положения ползунка, которое затем установит новый размер шрифта текста. Опять-таки, данный подход работает только с двунаправленной привязкой данных.

И, наконец, можно поменять местами роли ползунка и текстового поля, чтобы ползунок привязывался к текстовому полю. Для этого понадобится создать непривязанный элемент TextBox и назначить ему имя:

```
<TextBox Name="txtFontSize" Text="10">
</TextBox>
```

Затем можно привязать свойство Slider.Value, как показано ниже:

```
<Slider Name="sliderFontSize" Margin="3"
 Minimum="1" Maximum="40"
 Value="{Binding ElementName=txtFontSize, Path=Text, Mode=TwoWay}"
 TickFrequency="1" TickPlacement="TopLeft">
</Slider>
```

Теперь ползунок под контролем. При отображении окна в первый раз значение свойства TextBox.Text извлекается и применяется для установки его значения Value. Когда пользователь перетаскивает ползунок в новую позицию, он использует привязку для обновления текстового поля. В качестве альтернативы пользователь может обновить значение ползунка (и размер шрифта текста примера), вводя значение в текстовое поле.

На заметку! В случае привязки Slider.Value текстовое поле ведет себя несколько иначе, чем в предыдущих двух примерах. Любые изменения, которые вносятся в текстовое поле, применяются немедленно, вместо того, чтобы ожидать момента утери фокуса. Более подробно об этом речь пойдет в следующем разделе.

Как видно из примера, двунаправленные привязки обеспечивает значительную гибкость. Их можно использовать для применения изменений от источника к цели и от цели к источнику. Допускается их применение в комбинации, что позволяет создать неожиданно сложные окна без какого-либо кода.

Обычно решение относительно того, куда применять выражение привязки, диктуется логикой модели кодирования. В предыдущем примере, возможно, было бы больше смысла поместить привязку в свойство TextBox.Text вместо свойства Slider.Value, потому что текстовое поле — это необязательное дополнение к вполне готовому примеру, а не основной ингредиент, на который полагается ползунок. Также имело бы больше смысла привязать текстовое поле непосредственно к свойству TextBlock.FontSize вместо свойства Slider.Value. (Концептуально вы заинтересованы в том, чтобы видеть текущий размер шрифта, и ползунок — только один из способов его установки. Даже несмотря на то, что положение ползунка совпадает с размером шрифта, это — необязательная дополнительная деталь, если вы пытаетесь написать максимально ясную разметку.) Конечно, эти решения субъективны и определяются стилем кодирования. Наиболее важный урок состоит в том, что три подхода могут обеспечить одинаковое поведение.

В следующих разделах мы рассмотрим две детали, касающиеся этого примера. Во-первых, речь пойдет о возможных выборах при установке направления привязки. Во-вторых, будет показано, каким образом точно указать WPF, когда имеет смысл обновлять исходное свойство при двунаправленной привязке.

Обновления привязок

В примере, который показан на рис. 8.5 (где TextBox.Text привязывается к TextBlock.FontSize) имеется еще один нюанс. При попытке изменить отображаемый размер шрифта, вводя значение в текстовое поле, ничего не происходит. Изменение не применяется до тех пор, пока не будет совершен переход на другой элемент. Это поведение отличается от поведения, которое демонстрировалось в примере с ползунком. Там новый размер шрифта применялся после перетаскивания ползунка в другую позицию, т.е. в переходе на другой элемент вообще не было необходимости.

Чтобы понять это различие, следует повнимательнее присмотреться к выражениями привязки, которые используются этими двумя элементами управления. Когда применяется привязка OneWay или TwoWay, измененное значение распространяется от источника к цели немедленно. В случае с ползунком есть однонаправленное выражение привязки в TextBlock. Таким образом, изменения в свойстве Slider.Value немедленно отражаются в свойстве TextBlock.FontSize. То же поведение имеет место в примере с текстовым полем: изменения в источнике (которым является TextBlock.FontSize) немедленно влияют на цель (TextBox.Text).

Однако изменения, протекающие в обратном направлении — от цели к источнику — не обязательно происходят немедленно. Вместо этого их поведение управляется свойством Binding.UpdateSourceTrigger, которое принимает одно из значений, описанных в табл. 8.2. Когда текст берется из текстового поля и используется для обновления свойства TextBlock.FontSize, это пример обновления цель-источник, которое использует поведение UpdateSourceTrigger.LostFocus.

Таблица 8.2. Значения перечисления UpdateSourceTrigger

Имя	Описание
PropertyChanged	Источник обновляется немедленно, когда изменяется целевое свойство
LostFocus	Источник обновляется немедленно, когда изменяется целевое свойство и цель теряет фокус
Explicit	Источник не обновляется, пока не будет вызван метод BindingExpression.UpdateSource()
Default	Поведение обновления определяется метаданными целевого свойства (формально — его свойства FrameworkPropertyMetadata.DefaultUpdateSourceTrigger). Для большинства свойств поведением по умолчанию будет PropertyChanged, хотя свойство TextBox.Text обладает поведением по умолчанию LostFocus

Помните, что значения в табл. 8.2 не оказывают эффекта на обновление цели. Они просто управляют тем, как обновляется *источник* в привязках TwoWay и OneWayToSource.

Вооружившись этим знанием, можно усовершенствовать пример с текстовым полем, чтобы изменения применялись к размеру шрифта по мере их ввода в текстовое поле:

```
<TextBox Text="{Binding ElementName=txtSampleText, Path=FontSize, Mode=TwoWay,
 UpdateSourceTrigger=PropertyChanged}" Name="txtFontSize"></TextBox>
```

Совет. Поведением по умолчанию свойства TextBox.Text является LostFocus просто потому, что текст в текстовом поле будет изменяться непрерывно в процессе пользовательского ввода, вызывая множественные обновления. В зависимости от того, как исходный элемент управления обновляет себя, режим обновления PropertyChanged может сделать приложение более медлительным. Вдобавок это может заставить исходный объект обновлять себя до завершения редактирования, что создаст проблемы при проверке достоверности.

Для полного контроля над моментом обновления исходного объекта можно выбрать режим UpdateSourceTrigger.Explicit. Если воспользоваться этим подходом в примере с текстовым полем, то когда текстовое поле утратит фокус, ничего не произойдет. Вместо этого код должен будет вручную инициировать обновление. Например, можно было бы добавить кнопку Add (Добавить), которая вызовет метод BindingExpression.UpdateSource(), инициируя немедленное обновление размера шрифта.

Разумеется, прежде чем можно будет вызвать метод BindingExpression.UpdateSource(), нужен способ получения объекта BindingExpression. Объект BindingExpression — это тонкая упаковка, которая содержит в себе две вещи: уже известный объект Binding (предоставленный через свойство BindingExpression.ParentBinding) и объект, привязанный от источника (BindingExpression.DataItem). Вдобавок объект BindingExpression предоставляет два метода для запуска немедленного обновления одной части привязки: UpdateSource() и UpdateTarget().

Для получения объекта BindingExpression используется метод GetBindingExpression(), унаследованный каждым элементом от базового класса FrameworkElement, которому передается целевое свойство, имеющее привязку. Ниже приведен пример, в котором изменяется размер шрифта в TextBlock на основе текущего текста в текстовом поле:

```
// Получить привязку, примененную к текстовому полю.
BindingExpression binding = txtFontSize.GetBindingExpression(TextBox.TextProperty);

// Обновить привязанный источник (the TextBlock).
binding.UpdateSource();
```

Привязка к объектам, не являющимся элементами

До сих пор добавлялись привязки, которые устанавливали связь между двумя элементами. Однако в приложениях, управляемых данными, чаще создаются выражения привязки, которые извлекают данные из невизуальных объектов. Единственное требование, которое должно при этом соблюдаться — информация, которую необходимо отобразить, должны храниться в *общедоступных свойствах*. Инфраструктура привязки данных WPF не может извлекать приватную информацию или читать общедоступные поля.

При привязке к объекту, не являющемуся элементом, следует отказаться от свойства Binding.ElementName и применять вместо него одно из следующих свойств.

1. Source. Ссылка, указывающая на исходный объект; другими словами, это объект, поставляющий данные.
2. RelativeSource. Указывает на исходный объект, использующий объект RelativeSource, который позволяет базировать ссылку на текущем элементе. Это специализированный инструмент, который удобен при написании шаблонов элементов управления и шаблонов данных.
3. DataContext. Если источник не был указан с помощью свойства Source или RelativeSource, то среда WPF производит поиск в дереве элементов, начиная с текущего элемента. Она проверяет свойство DataContext каждого элемента и использует первый из них, который не равен null. Свойство DataContext исключительно полезно, когда нужно привязать несколько свойств одного объекта к разным элементам, потому что можно установить свойство DataContext высокоуровневого объекта контейнера, вместо его установки непосредственно на целевой элемент.

В следующем разделе эти варианты описаны более подробно.

Свойство Source

Свойство Source достаточно прямолинейно. Единственный момент, который следует учитывать — объект данных должен быть сделан удобным для привязки. Как будет показано, для получения объекта данных существует несколько подходов: извлечь его из ресурса, генерировать программно или получить от поставщика данных.

Простейший вариант — установить Source в некоторый готовый и доступный статический объект. Например, можно создать статический объект в коде и использовать его. Или же можно применить ингредиент из библиотеки классов .NET, как показано ниже:

```
<TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
 Path=Source}"></TextBlock>
```

Это выражение привязки получает объект FontFamily, который предоставлен свойством SystemFonts.IconFontFamily. (Обратите внимание, что для установки свойства Binding.Source понадобится помощь расширения разметки Static.) Затем свойство

Binding.Path устанавливается в свойство FontFamily.Source, которое выдает имя семейства шрифтов. Результатом будет единственная строка текста. В Windows Vista или Windows 7 имя шрифта выглядит как Segoe UI.

Другой вариант состоит в привязке к объекту, который ранее создавался в виде ресурса. Например, следующая разметка создает объект FontFamily, указывающий на шрифт Calibri:

```
<Window.Resources>
  <FontFamily x:Key="CustomFont">Calibri</FontFamily>
</Window.Resources>
```

Ниже показан элемент TextBlock, привязанный к ресурсу:

```
<TextBlock Text="{Binding Source={StaticResource CustomFont}, Path=Source}">
</TextBlock>
```

После этого появляется текст “Calibri”.

Свойство RelativeSource

Свойство RelativeSource позволяет установить его в исходный объект на основе его отношения к целевому объекту. Например, свойством RelativeSource можно воспользоваться для привязки элемента к самому себе или для привязки к родительскому элементу, который находится в неизвестном количестве уровней выше в дереве элементов.

Для установки свойства Binding.RelativeSource применяется объект RelativeSource. Это несколько усложняет синтаксис, поскольку нужно создать объект Binding и внутри него — вложенный объект RelativeSource. Один вариант состоит в использовании синтаксиса установки свойства вместо расширения разметки Binding.

Например, в следующем коде создается объект Binding для свойства TextBlock.Text. Объект Binding использует RelativeSource, которое ищет родительское окно и отображает заголовок окна.

```
<TextBlock>
  <TextBlock.Text>
    <Binding Path="Title">
      <Binding.RelativeSource>
        <RelativeSource Mode="FindAncestor" AncestorType="{x:Type Window}" />
      </Binding.RelativeSource>
    </Binding>
  </TextBlock.Text>
</TextBlock>
```

Для объекта RelativeSource выбран режим FindAncestor, который заставляет его осуществлять поиск вверх по дереву элементов до тех пор, пока не будет найден тип элемента, определенный свойством AncestorType.

Наиболее общий способ записи этой привязки состоит в комбинировании ее в одну строку, используя расширения разметки Binding и RelativeSource, как показано ниже:

```
<TextBlock Text="{Binding Path=Title,
  RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type Window}} }">
</TextBlock>
```

Режим FindAncestor — один из четырех возможных вариантов при создании объекта RelativeSource. Все варианты кратко описаны в табл. 8.3.

Таблица 8.3. Значения перечисления `RelativeSourceMode`

Имя	Описание
`Self`	Выражение привязывается к другому свойству того же элемента
`FindAncestor`	Выражение привязывается к родительскому элементу. WPF будет проводить поиск вверх по дереву элементов, пока не найдет нужный родительский элемент. Чтобы указать родителя, необходимо также установить свойство `AncestorType` для индикации типа родительского элемента, который должен быть найден. Дополнительно с помощью свойства `AncestorLevel` можно пропустить определенное количество совпадений указанного элемента. Например, если требуется привязка к третьему элементу типа `ListBoxItem` при восхождении вверх по дереву, то следует установить `AncestorType={x:Type ListBoxItem}` и `AncestorLevel=3`, тем самым пропуская первые два `ListBoxItem`. По умолчанию `AncestorLevel` равен 1, и поиск прекращается на первом найденном элементе
`PreviousData`	Выражение осуществляет привязку к предыдущему элементу данных в списке, привязанном к данным. Это можно использовать в элементе списка
`TemplatedParent`	Выражение осуществляет привязку к элементу, к которому применен шаблон. Этот режим работает, только если привязка находится внутри шаблона элемента управления или шаблона данных

На первый взгляд свойство `RelativeSource` может показаться излишним усложнением разметки. В конце концов, почему бы просто не привязаться непосредственно к необходимому источнику, используя свойство `Source` или `ElementName`? Однако, это не всегда возможно, и обычно потому, что объект-источник и целевой объект находятся в разных частях разметки. Так получается при создании шаблонов элементов управления и шаблонов данных. Например, при построении шаблона данных, который изменяет способ представления элементов в списке, может понадобиться доступ к объекту `ListBox` верхнего уровня, чтобы прочитать какое-то его свойство.

Свойство `DataContext`

В некоторых случаях имеется множество элементов, привязанных к одному объекту. Например, рассмотрим следующую группу элементов `TextBlock`, каждый из которых использует сходное выражение привязки для извлечения различных деталей о шрифте значков по умолчанию, включая промежутки между строками, стиль и вес первой гарнитуры (то и другое — просто Regular). Можете воспользоваться свойством `Source` для каждого из них, но это приводит к довольно длинной разметке:

```
<StackPanel>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
   Path=Source}"></TextBlock>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
   Path=LineSpacing}"></TextBlock>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
   Path=FamilyTypefaces[0].Style}"></TextBlock>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
   Path=FamilyTypefaces[0].Weight}"></TextBlock>
</StackPanel>
```

В такой ситуации было бы яснее и удобнее определить источник привязки один раз с помощью свойства `FrameworkElement.DataContext`. В данном примере имеет смысл установить свойство `DataContext` элемента `StackPanel`, содержащего в себе все эле-

менты TextBlock. (Можно было бы также установить свойство DataContext на еще более высоком уровне, например, на уровне всего окна, но лучше определить его насколько возможно *уже*, чтобы яснее выразить намерения.)

Установить свойство DataContext элемента можно таким же образом, как устанавливается свойство Binding.Source. Другими словами, можно встроить объект, извлечь его из статического свойства либо получить из ресурса, как показано ниже:

```
<StackPanel DataContext="{x:Static SystemFonts.IconFontFamily}">
```

После этого выражения привязки упрощаются за счет исключения некоторой информации об источнике:

```
<TextBlock Margin="5" Text="{Binding Path=Source}"></TextBlock>
```

Когда информация об источнике отсутствует в выражении привязки, WPF проверяет свойство DataContext элемента. Если оно равно null, WPF ищет в дереве элементов первый контекст данных, отличный от null. (Изначально свойства DataContext всех элементов равны null.) Если подходящий контекст данных обнаружен, то он используется для привязки. Если же нет, то выражение привязки не передает никакого значения целевому свойству.

На заметку! При создании привязки с явно указанным источником в свойстве Source элемент использует этот источник вместо любого другого доступного контекста данных.

Этот пример продемонстрировал, как создавать базовую привязку к объекту, который не является элементом. Однако чтобы использовать такой прием в реальном приложении, понадобятся некоторые дополнительные знания. В главе 19 будет показано, как отобразить информацию, извлеченную из базы данных, основываясь на описанных приемах привязки данных.

Резюме

В этой главе был представлен краткий обзор основ привязки данных. Вы узнали, как извлечь информацию из одного элемента и отобразить в другом, не написав ни единой строки кода. И хотя эта техника выглядит сейчас довольно скромно, ее знание позволит решать более сложные задачи, такие как изменение стиля элементов управления посредством специальных шаблонов (см. главу 17).

В главах 19 и 20 навыки привязки данных будут существенно расширены. Вы узнаете, как отображать целые коллекции объектов данных в списке, обрабатывать редактирование с помощью проверки достоверности и превращать обычный текст в изощренно сформатированное отображение данных. Полученных до сих пор знаний о привязке данных будет достаточно, чтобы приступить к изучению последующих глав.