ГЛАВА 12

Стили

В предыдущей главе шла речь о системе ресурсов WPF, которая позволяет определять объекты в одном месте и затем повторно использовать их в других частях кода разметки. Хотя ресурсы могут применяться для хранения самых различных объектов, чаще всего они служат для хранения *стилей*.

Стиль — это коллекция значений свойств, которые могут быть применены к элементу. Система стилей WPF играет ту же роль, что и стандарт каскадных таблиц стилей (Cascading Style Sheet — CSS) играет в HTML-коде разметки. Подобно CSS, стили WPF позволяют определять общий набор характеристик форматирования и применять их по всему приложению для обеспечения согласованности. Как и CSS, стили WPF могут работать автоматически, предназначаться для элементов конкретного типа и каскадироваться через дерево элементов. Однако стили WPF являются более мощными, потому что могут устанавливать *любое* свойство зависимостей. Это означает, что их можно применять для стандартизации не связанных с форматированием характеристик вроде поведения какого-то конкретного элемента управления. Кроме того, стили WPF еще также поддерживают *триггеры*, которые позволяют делать так, чтобы стиль элемента управления изменялся в случае изменения какого-то другого свойства (как будет показано далее в этой главе), и могут использовать *шаблоны* для переопределения стандартного внешнего вида того или иного элемента управления (как будет показано в главе 15). Научившись применять стили, вы обязательно начнете включать их во все свои WPF-приложения.

Основные сведения о стилях

Как рассказывалось в предыдущей главе, ресурсы предлагают несколько ключевых преимуществ, включая более простой код разметки и более удобные в сопровождении приложения. А для чего тогда нужны стили?

Чтобы понять, какие преимущества дают стили, давайте рассмотрим простой пример. Предположим, что необходимо стандартизировать используемый в окне стиль. Самым простым подходом будет установить отвечающие за шрифт свойства содержащего этот шрифт окна. К числу таких свойств, которые, кстати, определяются в классе `Control`, относятся `FontFamily`, `FontSize`, `FontWeight` (для шрифта с полужирным начертанием), `FontStyle` (для шрифта с курсивным начертанием) и `FontStretch`. Благодаря функции наследования значений свойств, при установке этих свойств на уровне окна все элементы внутри окна получат одинаковые значения, если только они явно не переопределяют их.

На заметку! Функция наследования значений свойств является лишь одной из многих дополнительных функций, которые могут предоставлять свойства зависимостей. Сами свойства зависимостей подробно рассматривались в главе 6.

Теперь давайте возьмем другую ситуацию. Представьте, что требуется заблокировать шрифт, используемый для каких-то определенных элементов в пользовательском интерфейсе. Если есть возможность изолировать эти элементы в специальный контейнер (например, если они все находятся внутри одного элемента управления `Grid` или `StackPanel`), тогда в принципе можно воспользоваться тем же самым подходом и установить отвечающие за шрифт свойства этого контейнера. Однако в реальной жизни все обычно оказывается не так просто. Например, может быть необходимо, чтобы у всех кнопок была своя гарнитура и размер шрифта, не зависящая от параметров шрифта, которые используются в других элементах. В таком случае нужен способ, позволяющий определить эти детали в одном месте и повторно использовать их везде, где они будут необходимы.

Ресурсы предоставляют решение, но оно является несколько громоздким. Поскольку никакого объекта `Font` в WPF не существует (есть только коллекция связанных со шрифтом свойств), ресурсы можно определить только так, как показано ниже:

```
<Window.Resources>
  <FontFamily x:Key="ButtonFontFamily">Times New Roman</FontFamily>
  <sys:Double x:Key="ButtonFontSize">18</s:Double>
  <FontWeight x:Key="ButtonFontWeight">Bold</FontWeight>
</Window.Resources>
```

С помощью этого фрагмента кода разметки в окно добавляются три ресурса: объект `FontFamily` с именем шрифта, который должен использоваться, объект `Double` с числом 18 и перечислимое значение `FontWeight.Bold`. Здесь предполагается, что .NET-пространство имен `System` было отображено на префикс `sys` пространства имен XML, как показано ниже:

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

Совет. При установке свойств с использованием ресурса, важно, чтобы типы данных были сопоставлены точно. WPF не будет использовать преобразование типов так же, как делает это, когда значение атрибута устанавливается напрямую. Например, в случае установки в элементе атрибута `FontFamily` можно использовать строку “Times New Roman”, потому что `FontFamilyConverter` автоматически создаст необходимый объект `FontFamily`. Однако при попытке установить свойство `FontFamily` с помощью ресурса типа строки подобное чудо не произойдет: в таком случае анализатор XAML сгенерирует исключение.

Определив необходимые ресурсы, далее следует фактически использовать их в элементе. Поскольку за время жизненного цикла приложения ресурсы никогда не изменяются, имеет смысл применять статические ресурсы, как показано ниже:

```
<Button Padding="5" Margin="5" Name="cmd"
  FontFamily="{StaticResource ButtonFontFamily}"
  FontWeight="{StaticResource ButtonFontWeight}"
  FontSize="{StaticResource ButtonFontSize}"
  >A Customized Button
</Button>
```

Этот пример работает, позволяя исключить касающиеся шрифта детали из кода разметки. Однако он также приводит к появлению двух новых проблем.

- Нет никакого четкого признака, что все три данные ресурса связаны между собой (кроме разве что похожих имен). Это делает приложение менее удобным для обслуживания. Эта проблема становится особенно серьезной, если требуется установить больше связанных со шрифтом свойств или если принимается решение поддерживать разные параметры шрифта для разных типов элементов.
- Необходимый для использования этих ресурсов код разметки подразумевает применение слишком большого количества описаний, в результате чего он оказывается даже еще менее лаконичным, чем подход, который он заменяет (т.е. длиннее определения связанных со шрифтом свойств прямо в элементе).

Чтобы избавиться от первой проблемы, можно определить специальный класс (например, `FontSettings`), объединяющий все касающиеся шрифта детали вместе, а затем создать один объект `FontSettings` как ресурс и использовать его различные свойства в коде разметки. Однако вторую проблему это не устранит — для ее решения придется приложить еще немало дополнительных усилий.

Стили предоставляют идеальное решение. Можно определить один единственный стиль, упаковывающий все свойства, которые требуется установить. Например, можно поступить так, как показано ниже.

```
<Window.Resources>
  <Style x:Key="BigFontButtonStyle">
    <Setter Property="Control.FontFamily" Value="Times New Roman" />
    <Setter Property="Control.FontSize" Value="18" />
    <Setter Property="Control.FontWeight" Value="Bold" />
  </Style>
</Window.Resources>
```

В этом коде разметки создается один единственный ресурс: объект `System.Windows.Style`. В этом объекте содержится коллекция `Setters` с тремя объектами `Setter`, по одному для каждого свойства, которое требуется установить. В каждом объекте `Setter` указывается имя свойства, на которое он влияет, и значение, которое он должен применять к этому свойству. Как и все ресурсы, объект стиля имеет ключевое имя, по которому его можно при необходимости извлекать из коллекции. В данном случае это ключевое имя выглядит как `BigFontButtonStyle`. (По общепринятому соглашению ключевые имена стилей обычно заканчиваются словом “Style”.)

Каждый элемент WPF может использовать один единственный стиль (или не использовать его вообще). Стиль встраивается в элемент через свойство `Style` (которое определено в базовом классе `FrameworkElement`). Например, сконфигурировать кнопку так, чтобы она использовала созданный выше стиль, можно, указав кнопке на ресурс стиля:

```
<Button Padding="5" Margin="5" Name="cmd"
  Style="{StaticResource BigFontButtonStyle}"
  >A Customized Button
</Button>
```

Конечно, стиль можно установить и программно. Все, что для этого требуется — это извлечь стиль из ближайшей коллекции `Resources` с помощью уже знакомого метода `FindResource()`. Ниже приведен код, который можно было бы использовать для объекта `Button` с именем `cmd`:

```
cmdButton.Style = (Style)cmd.FindResource("BigFontButtonStyle");
```

На рис. 12.1 показано окно, в котором для целых двух кнопок используется специальный стиль `BigFontButtonStyle`.

На заметку! Стили задают первоначальный внешний вид элемента, но устанавливаемые ими характеристики могут переопределяться. Например, если применить стиль `BigFontButtonStyle` *и* явно установить свойство `FontSize`, параметр `FontSize` в дескрипторе кнопки будет переопределять этот стиль. В идеале полагаться на такое поведение не нужно — вместо этого лучше создавать больше стилей, так чтобы на уровне стиля можно было устанавливать как можно больше деталей. Это обеспечит высокую гибкость при настройке пользовательского интерфейса в будущем и позволит свести количество "разрушений" в нем к минимуму.

Система стилей добавляет множество преимуществ. Она не только позволяет создавать группы параметров, четко связанных между собой, но и также упрощает код разметки, облегчая применение этих параметров. Лучше всего то, что стиль можно применять, не беспокоясь о том, какие свойства он устанавливает. В предыдущем примере параметры шрифта были организованы в стиль под названием `BigFontButtonStyle`. Если позже вдруг понадобится обеспечить кнопки с крупным шрифтом областями `Padding` и `Margin`, в код можно будет также очень легко дополнительно добавить, соответственно, свойства `Padding` и `Margin`, после чего все, использующие данный стиль кнопки, автоматически получат новые параметры стиля.

Коллекция `Setters` является самым важным свойством класса `Style`. Но в принципе ключевыми считаются целых пять свойств, краткое описание которых приведено в табл. 12.1.

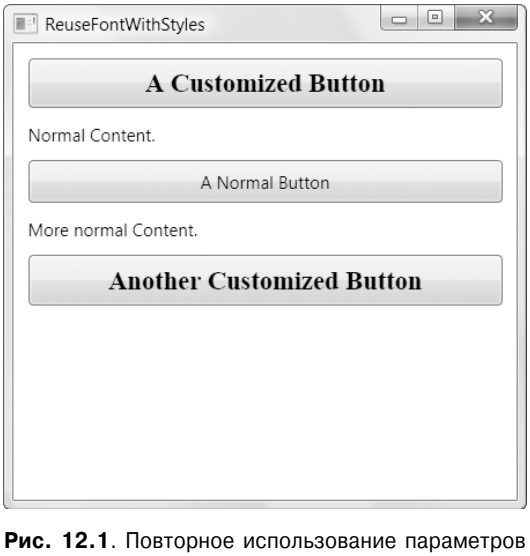

Рис. 12.1. Повторное использование параметров кнопки с помощью стиля

Таблица 12.1. Свойства класса `Style`

Свойство	Описание
`Setters`	Коллекция объектов `Setter` или `EventSetter`, которые автоматически устанавливают значения свойств и добавляют обработчики событий.
`Triggers`	Коллекция объектов, которые наследуются от `TriggerBase` и позволяют автоматически изменять параметры стиля. Параметры стиля могут изменяться, например, либо в случае изменения значения какого-то другого свойства, либо в случае возникновения какого-нибудь события.
`Resources`	Коллекция ресурсов, которые должны использоваться со стилями. Например, может возникнуть необходимость использовать один единственный объект для установки более чем одного свойства. В таком случае более эффективным вариантом будет создать объект как ресурс и затем использовать этот ресурс в объекте `Setter` (чем создавать его в виде части каждого объекта `Setter`, используя вложенные дескрипторы).
`BasedOn`	Свойство, которое позволяет создавать более специализированный стиль, наследующий (и, при желании, переопределяющий) параметры другого стиля.
`TargetType`	Свойство, идентифицирующее тип элемента, на который действует данный стиль. Это свойство позволяет создавать как объекты `Setter`, влияющие только на определенные элементы, так и объекты `Setter`, автоматически вступающие в силу для элементов конкретного типа.

Теперь, рассмотрев базовый пример того, как работает стиль, можно переходить к более глубокому анализу модели стилей.

Создание объекта стиля

В предыдущем примере объект стиля определялся на уровне окна и затем повторно использовался в двух находящихся внутри этого окна кнопках. Хотя такой дизайн и является распространенным, применять именно его, конечно же, не обязательно.

Если требуется создать стили с более точным предназначением, их можно определить с помощью коллекции `Resources` содержащего их контейнера, в роли которого может выступать элемент `StackPanel` или `Grid`. При желании иметь возможность повторно использовать стили во всем приложении их можно определить с помощью коллекции `Resources` приложения. Эта два подхода также являются довольно распространенными. Собственно говоря, применять стили и ресурсы вместе вовсе необязательно. Например, стиль определенной кнопки можно определить, и просто напрямую заполнив ее коллекцию `Style`, как показано ниже:

```
<Button Padding="5" Margin="5">
  <Button.Style>
    <Style>
      <Setter Property="Control.FontFamily" Value="Times New Roman" />
      <Setter Property="Control.FontSize" Value="18" />
      <Setter Property="Control.FontWeight" Value="Bold" />
    </Style>
  </Button.Style>
  <Button.Content>A Customized Button</Button.Content>
</Button>
```

Такой подход работает, но очевидно является гораздо менее полезным, потому что исключает возможность разделения (совместного использования) данного стиля вместе с другими элементами.

Такой подход не стоит затрачиваемых усилий, если стиль применяется просто для установки определенных свойств (как в этом примере), поскольку установить эти свойства напрямую гораздо проще. Однако иногда в нем есть смысл, если используется еще одна функция стилей и ее требуется применить всего лишь к одному единственному элементу. Например, такой подход очень удобен для присоединения триггеров к элементу. Он также позволяет изменять часть шаблона элемента управления. (В таких случаях используется свойство `Setter.TargetName` для применения объекта `Setter` к какому-то конкретному компоненту внутри элемента, например, к кнопками полосы прокрутки в окне списка. Более подробно об этом приеме будет рассказываться в главе 15.)

Установка свойств

Как было показано, каждый объект `Style` упаковывает коллекцию объектов `Setter`. Каждый объект `Setter` устанавливает в элементе одно свойство. Единственное ограничение состоит в том, что объект `Setter` может изменять только свойство зависимости — другие свойства изменяться им не могут.

В некоторых случаях установка значения свойства с помощью простой строки атрибута оказывается невозможной. Например, объект `ImageBrush` (подобный тому, который в предыдущей главе использовался для отображения плиточного узора) установить с помощью простой строки нельзя. В такой ситуации можно воспользоваться уже знакомым XAML-приемом замещения атрибута вложенным элементом, как показано ниже:

```
<Style x:Key="HappyTiledElementStyle">
  <Setter Property="Control.Background">
    <Setter.Value>
      <ImageBrush TileMode="Tile"
        ViewportUnits="Absolute" Viewport="0 0 32 32"
        ImageSource="happyface.jpg" Opacity="0.3">
      </ImageBrush>
    </Setter.Value>
  </Setter>
</Style>
```

Совет. Если необходимо иметь возможность повторно использовать один и тот же объект `ImageBrush` в более чем одном стиле (или в более чем одном объекте `Setter` в пределах одного и того же стиля), можно определить его как ресурс и затем просто применять в стиле этот ресурс.

Чтобы идентифицировать устанавливаемое свойство, нужно предоставить как имя свойства, так и имя класса. Однако указываемое имя класса вовсе не обязательно должно представлять тот класс, в котором определено данное свойство. Это может быть и имя производного класса, который наследуется от данного свойства. Например, взгляните на следующую версию стиля `BigFontButton`, в которой ссылки на класс `Control` заменяются ссылками на класс `Button`:

```
<Style x:Key="BigFontButtonStyle">
  <Setter Property="Button.FontFamily" Value="Times New Roman" />
  <Setter Property="Button.FontSize" Value="18" />
  <Setter Property="Button.FontWeight" Value="Bold" />
</Style>
```

Если в примере, который был показан на рис. 12.1, использовать вместо предыдущего этот стиль, результат будет выглядеть точно так же. Так в чем тогда разница? В данном случае разница состоит в способе, которым WPF будет обрабатывать другие классы, которые могут включать те же самые свойства `FontFamily`, `FontSize` и `FontWeight`, но не наследоваться от `Button`. Например, если применить эту версию стиля `BigFontButton` к элементу управления `Label`, ничего не произойдет. WPF просто проигнорирует эти три свойства, поскольку здесь они недействительны. Но в случае применения исходного стиля связанные со шрифтом свойства обязательно повлияют на элемент управления `Label`, потому что класс `Label` наследуется от класса `Control`.

Совет. Тот факт, что WPF игнорирует недействительные свойства, означает, что можно также устанавливать и свойства, которые вовсе необязательно будут доступны в элементе, к которому применяется данный стиль. Например, если установить свойство `ButtonBase.IsCancle`, оно будет иметь эффект только в случае установки стиля на кнопке.

В WPF имеются случаи, когда одни и те же свойства определяются в более чем одном месте в иерархии элементов. Например, полный набор связанных со шрифтом свойств (таких как свойство `FontFamily`) определяется как в классе `Control`, так и в классе `TextBlock`. Из-за этого в случае создания стиля, который применяется к объектам `TextBox` и элементам, унаследованным от класса `Control`, может возникнуть идея написать код разметки следующим образом:

```
<Style x:Key="BigFontStyle">
  <Setter Property="Button.FontFamily" Value="Times New Roman" />
  <Setter Property="Button.FontSize" Value="18" />

  <Setter Property="TextBlock.FontFamily" Value="Arial" />
  <Setter Property="TextBlock.FontSize" Value="10" />
</Style>
```

Однако такой подход не даст желаемого эффекта. Проблема заключается в том, что хотя свойства `Button.FontFamily` и `TextBlock.FontFamily` и объявляются отдельно в соответствующих им базовых классах, оба они являются ссылками на одно и то же свойство зависимостей. (Другими словами, свойства `TextBlock.FontSizeProperty` и `Control.FontSizeProperty` являются ссылками, которые указывают на один и тот же объект `DependencyProperty`. Впервые о вероятности такой проблемы говорилось в главе 6.) В результате из-за этого при использовании данного стиля WPF устанавливает свойство `FontFamily` и `FontSize` дважды. Параметры, примененные последними (каковыми в данном случае являются шрифт Arial и размер 10), имеют преимущественное значение и применяются к обоим объектам — объекту `Button` и объекту `TextBlock`. Хотя эта проблема является весьма специфической и не возникает со многими свойствами, важно не забывать о ней, если часто приходится создавать стили, применяющие разное форматирование к разным типам элементов.

Существует еще один прием, которым можно пользоваться для упрощения объявлений стилей. Если все свойства предназначаются для одного и того же типа элементов, можно установить свойство TargetType объекта Style, указав в нем класс, для которого данные свойства должны быть действительными. Например, в случае создания стиля, предназначенного только для кнопок, можно написать такой код:

```
<Style x:Key="BigFontButtonStyle" TargetType="Button">
  <Setter Property="FontFamily" Value="Times New Roman" />
  <Setter Property="FontSize" Value="18" />
  <Setter Property="FontWeight" Value="Bold" />
</Style>
```

Это относительно небольшое удобство. Как будет рассказываться далее в этой главе, свойство TargetType также может дублироваться в качестве сокращения, что при условии пропускания ключевого имени стиля позволяет применять стили автоматически.

Добавление обработчиков событий

Отвечающие за установку свойств объекты Setters являются наиболее распространенными элементами в любом стиле, но существует также и возможность создания коллекции объектов EventSetters, привязывающих события к определенным обработчикам. Ниже показан пример добавления обработчиков событий для событий MouseEnter и MouseLeave.

```
<Style x:Key="MouseOverHighlightStyle">
 <EventSetter Event="TextBlock.MouseEnter" Handler="element_MouseEnter" />
 <EventSetter Event="TextBlock.MouseLeave" Handler="element_MouseLeave" />
 <Setter Property="TextBlock.Padding" Value="5"/>
</Style>
```

Ниже приведен код обработки этих событий.

```
private void element_MouseEnter(object sender, MouseEventArgs e)
{
    ((TextBlock)sender).Background = new
    SolidColorBrush(Colors.LightGoldenrodYellow);
}
private void element_MouseLeave(object sender, MouseEventArgs e)
{
    ((TextBlock)sender).Background = null;
}
```

События MouseEnter и MouseLeave маршрутизируются напрямую, что означает, что они и не поднимаются вверх, и не тунеллируются вниз по дереву элементов. При желании применить эффект наведения курсора мыши к большому количеству элементов (например, сделать так, что при наведении курсора мыши на каждый элемент изменялся цвет его фона), обработчики событий MouseEnter и MouseLeave необходимо добавить в каждый элемент. Обработчики событий, основанные на использовании стиля, упрощают эту задачу. Далее остается только применить один стиль, который может включать как отвечающие за свойства, так и отвечающие за события объекты Setter:

```
<TextBlock Style="{StaticResource MouseOverHighlightStyle}">
  Hover over me.
</TextBlock>
```

На рис. 12.2 показан простой демонстрационный пример применения этого приема с тремя элементами, два из которых используют стиль `MouseOverHighlightStyle`.

Рис. 12.2. Обработки событий `MouseEnter` и `MouseLeave` с помощью стиля

Отвечающие за события объекты `Setter` редко применяются в WPF. При необходимости получить показанные здесь функциональные возможности чаще используются триггеры событий, которые определяют требующееся действие декларативно (и потому не требуют написания никакого кода). Триггеры событий предназначены для реализации анимационных эффектов, что делает их более удобными для создания эффектов наведения курсора мыши.

Объекты `Setter` прекрасно подходят для обработки событий, использующих поднятие. В такой ситуации обрабатывать требуемые события обычно легче в элементе более высокого уровня. Например, при необходимости соединить все кнопки в панели инструментов с одним и тем же обработчиком события `Click`, наилучшим подходом будет присоединить один такой обработчик к содержащему все эти кнопки элементу `Toolbar`. В такой ситуации применение для события объекта `Setter` лишь излишне усложнит код.

Совет. Во многих случаях понятнее явно определить все события и вообще не использовать для них объектов `Setter`. Если необходимо связать несколько событий с одним и тем же обработчиком, лучше делать это вручную. Также еще можно применять приемы вроде присоединения обработчика на уровне контейнера и централизации логики с помощью команд (о которых более подробно говорилось в главе 10).

Несколько уровней стилей

Хотя допускается определять неограниченное количество стилей на множестве различных уровней, каждый элемент WPF может одновременно использовать только один единственный объект стиля. Несмотря на то что на первый взгляд это может показаться ограничением, с наследованием значений свойств и наследованием стилей оно обычно не связано.

Например, предположим, что требуется назначить группе элементов управления один и тот же шрифт без применения к каждому из них одного и того же стиля. В таком случае можно разместить все эти элементы управления в одной панели (или каком-нибудь еще типе контейнера) и установить стиль контейнера. При условии установки таких свойств, которые подразумевают использование функции наследования значений, их значения будут передаваться потомкам. К числу свойств,

которые используют такую модель, относятся IsEnabled, IsVisible, Foreground и все свойства, связанные со шрифтом.

В других случаях может потребоваться создать стиль, основанный на каком-то другом стиле. Использовать такой вид наследования стилей можно путем установки атрибута BasedOn соответствующего стиля. Например, возьмем два следующих стиля:

```
<Window.Resources>
  <Style x:Key="BigFontButtonStyle">
    <Setter Property="Control.FontFamily" Value="Times New Roman" />
    <Setter Property="Control.FontSize" Value="18" />
    <Setter Property="Control.FontWeight" Value="Bold" />
  </Style>
  <Style x:Key="EmphasizedBigFontButtonStyle"
    BasedOn="{StaticResource BigFontButtonStyle}">
    <Setter Property="Control.Foreground" Value="White" />
    <Setter Property="Control.Background" Value="DarkBlue" />
  </Style>
</Window.Resources>
```

Первый стиль (BigFontButtonStyle) определяет три свойства, касающиеся шрифта. Второй стиль (EmphasizedBigFontButtonStyle) получает эти свойства от стиля BigFontButtonStyle и затем дополняет их еще двумя свойствами, которые изменяют цвета переднего плана и фона. Этот состоящий из двух частей дизайн дает возможность применять как одни лишь параметры шрифта, так и параметры шрифта и цвета вместе, а также позволяет создать больше стилей, включающих уже определенные детали шрифта или цвета (и не обязательно и те и другие одновременно).

На заметку! Свойство BasedOn можно использовать для создания целой цепочки наследуемых стилей. Главное помнить о том, что если одно и то же свойство устанавливается дважды, последний установщик (Setter) свойства (т.е. тот, что в производном классе расположен дальше всех в цепочке наследования) перекрывает любые более ранние определения.

На рис. 12.3 показано, как работает наследование стилей в простом окне, где используются оба стиля.

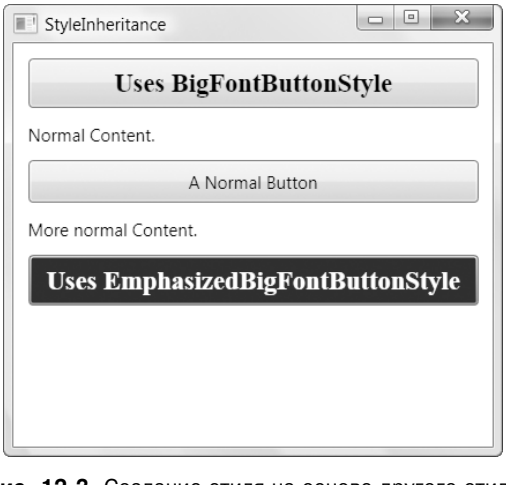

Рис. 12.3. Создание стиля на основе другого стиля

Наследование стилей увеличивает сложность

Хотя наследование стилей на первый взгляд может показаться очень удобным приемом, обычно оно не стоит затрачиваемых усилий. А все дело в том, что наследование стилей чревато теми же самыми проблемами, что и наследование кода, а именно — зависимостями, которые делают приложение более хрупким. Например, в случае использования приведенного выше кода разметки, одни и те же характеристики шрифта обязательно должны сохраняться для обоих стилей. В случае изменения стиля `BigFontButtonStyle` стиль `EmphasizedBigFontButtonStyle` тоже изменяется — если только явно не добавить дополнительных установщиков (`Setters`), переопределяющих наследуемые значения.

Эта проблема является довольно-таки тривиальной в данном примере с наследованием двух стилей, но при наследовании стилей в реальных приложениях степень ее серьезности существенно возрастает. Обычно стили разбиваются на категории на основании различных типов содержимого и роли, которую играет это содержимое. Например, приложение, предназначенное для проведения операций по сбыту продуктов, может включать стили вроде `ProductTitleStyle`, `ProductTextStyle`, `HighlightQuoteStyle`, `NavigationButtonStyle` и т.д. Если основать стиль `ProductTitleStyle` на стиле `ProductTextStyle` (например, потому, что они оба должны применять один и тот же шрифт), то в случае возникновения необходимости позже применить к стилю `ProductTextStyle` параметры, которые не должны применяться к стилю `ProductTitleStyle` (например, других полей), проблем не миновать. В таком случае придется определять эти параметры в стиле `ProductTextStyle` и затем явно переопределять их в стиле `ProductTitleStyle`. Из-за этого, в конечном счете, получится гораздо более сложная модель, в которой на самом деле использоваться повторно будет всего лишь несколько параметров стиля.

В отсутствие действительно веских причин основывать один стиль на другом (например, когда второй стиль является просто особым случаем первого и изменяет всего лишь несколько характеристик из огромного количества наследуемых параметров) наследование стилей применять не следует.

Автоматическое применение стилей по типу

Пока что было показано, как можно создавать именованные стили и ссылаться на них в коде разметки. Однако существует еще один подход. Можно делать так, чтобы к элементам определенного типа стиль применялся автоматически.

Делается это довольно просто. Все, что требуется — это установить свойство `TargetType` так, чтобы оно указывало на нужный тип (как описывалось ранее), и вообще не использовать ключевое имя. Тогда WPF устанавливает ключевое имя неявно с помощью расширения разметки типа, как показано ниже:

```
x:Key="{x:Type Button}"
```

Далее стиль автоматически применяется к любым кнопкам, встречающимся ниже в дереве элементов. Например, если определить стиль подобным образом в окне, он будет применяться к каждой кнопке в этом окне (при условии отсутствия далее в коде еще одного стиля, заменяющего его).

Ниже показан пример с окном, в котором стили кнопок устанавливаются автоматически для получения того же эффекта, что был у окна на рис. 12.1.

```
<Window.Resources>
  <Style TargetType="Button">
```

```
    <Setter Property="FontFamily" Value="Times New Roman" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="FontWeight" Value="Bold" />
  </Style>
</Window.Resources>
<StackPanel Margin="5">
  <Button Padding="5" Margin="5">Customized Button</Button>
  <TextBlock Margin="5">Normal Content.</TextBlock>
  <Button Padding="5" Margin="5" Style="{x:Null}">A Normal Button</Button>
  <TextBlock Margin="5">More normal Content.</TextBlock>
  <Button Padding="5" Margin="5">Another Customized Button</Button>
</StackPanel>
```

В этом примере средняя кнопка явно заменяет стиль. Вместо того чтобы предоставлять свой собственный новый стиль, она устанавливает для свойства `Style` значение `null`, что эффективно удаляет ранее установленный стиль.

Хотя автоматически применяемые стили и удобны, они могут усложнять дизайн. Ниже перечислено несколько возможных причин.

- В сложном окне с множеством стилей и множество уровней стилей становится трудно отслеживать то, устанавливается ли данное свойство с помощью наследования значений свойств или с помощью стиля (и если оно устанавливается с помощью стиля, то с помощью какого именно). В результате в случае возникновения необходимости изменить даже какую-нибудь простую деталь может потребоваться просматривать код разметки всего окна.
- Форматирование в окне часто сначала является более общим, а потом постепенно усложняется. В случае применения к окну автоматических стилей где-нибудь на раннем этапе, скорее всего, далее эти стили потребуется переопределять во многих местах с помощью явных стилей. А это значительно усложняет весь дизайн. Гораздо проще будет создать именованные стили для каждой комбинации необходимых характеристик форматирования и применять их по имени.
- В случае создания автоматического стиля, например, для элемента `TextBox`, обязательно придется вносить изменения в другие элементы управления, которые используют этот элемент `TextBlock` (например, в управляемый шаблоном элемент управления `ListBox`).

Во избежание таких проблем лучше всего применять автоматические стили рассудительно. Например, может оказаться лучше использовать автоматический стиль не для всего окна, а только для придания однообразного заполнения кнопкам или для управления параметрами полей элементов `TextBox` в конкретном контейнере.

Триггеры

Одна из тем в WPF посвящена тому, что можно делать *декларативно*. Используя хоть стили, хоть ресурсы, хоть связывание данных, нередко довольно много можно сделать, и не прибегая к помощи кода.

Триггеры являются еще одним примером такой тенденции. Используя триггеры, можно автоматизировать процесс внесения простых изменений в стили, обычно требующий создания стандартной логики обработки событий. Например, можно обеспечить реакцию на изменение значения свойства и автоматическое корректирование стиля соответствующим образом.

Триггеры связываются со стилями через коллекцию `Style.Triggers`. Каждый стиль может иметь неограниченное количество триггеров, а каждый триггер является экземпляром класса, который наследуется от `System.Windows.TriggerBase`. Доступные варианты перечислены в табл. 12.2.

С помощью коллекции `FrameworkElement.Triggers` триггеры можно применять к элементам напрямую, не создавая никакого стиля. Однако здесь имеется одно серьезное ограничение. Эта коллекция `Triggers` поддерживает только триггеры событий. (Никаких технических оснований для этого ограничения нет; просто разработчики WPF не успели завершить эту функциональную возможность и возможно сделают это в следующих версиях.)

Таблица 12.2. Классы, унаследованные от `TriggerBase`

Имя	Описание
`Trigger`	Это триггер самого простого типа. Он следит за появлением изменений в свойстве зависимостей и затем использует объект `Setter` для изменения стиля.
`MultiTrigger`	Этот триггер похож на предыдущий, но подразумевает проверку множества условий. Все условия должны вернуть `true`, прежде чем он вступит в силу.
`DataTrigger`	Этот триггер работает со связыванием данных. Он похож на первый, но только следит за появлением изменений не в свойстве зависимостей, а в любых связанных данных.
`MultiDataTrigger`	Этот триггер состоит из множества триггеров данных.
`EventTrigger`	Это наиболее сложный триггер. При возникновении события он применяет соответствующую анимацию.

Простой триггер

Простой триггер (`Trigger`) можно присоединять к любому свойству зависимостей. Например, создать эффект наведения курсора мыши и фокуса можно путем ответа на изменения в свойствах `IsFocused`, `IsMouseOver` и `IsPressed` класса `Control`.

В каждом простом триггере идентифицируется свойство, за которым должно вестись наблюдение, и значение, которого следует ожидать. Когда появляется необходимое значение, в действие вступают те установщики (`Setter`), которые были сохранены в коллекции `Trigger.Setters`. (К сожалению, использовать с триггером более сложную логику, предусматривающую сравнение значения для проверки того, подпадает ли оно под нужный диапазон, выполнение вычислений и другие операции, нельзя. В таких случаях лучше применять обработчик события.)

Ниже показан триггер, который ожидает, когда кнопка получит фокус клавиатуры, в случае чего устанавливает для нее темно-красный цвет фона.

```
<Style x:Key="BigFontButton">
  <Style.Setters>
    <Setter Property="Control.FontFamily" Value="Times New Roman" />
    <Setter Property="Control.FontSize" Value="18" />
  </Style.Setters>

  <Style.Triggers>
    <Trigger Property="Control.IsFocused" Value="True">
      <Setter Property="Control.Foreground" Value="DarkRed" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Преимуществом триггеров является то, что нет необходимости писать какую-либо логику для отмены их действия. Как только триггер становится недействительным, элементу сразу же возвращается его обычный внешний вид. В данном примере это означает, что как только пользователь убирает с кнопки фокус, нажав клавишу <Tab>, ее фон сразу же снова становится обычного серого цвета.

На заметку! Чтобы понять, как это работает, необходимо помнить о системе свойств зависимостей, подробно рассмотренной в главе 6. По сути, триггер является одним из множества поставщиков свойств, умеющих переопределять значение, которое возвращает свойство зависимостей. Однако исходное значение (устанавливаемое как локально, так и с помощью стиля) все равно остается. Как только триггер перестает действовать, упомянутое значение просто снова становится доступным.

Допускается создавать и множество триггеров, применяемых одновременно к одному и тому же элементу. Если эти триггеры устанавливают разные свойства, никакой неоднозначности не возникает. Однако если несколько триггеров изменяют одно и то же свойство, "побеждает" тот из них, который идет последним в списке.

Например, ниже показаны триггеры, которые изменяют внешний вид элемента управления в зависимости от того, находится ли на нем фокус, наведен ли на него курсор мыши или был ли на нем выполнен щелчок кнопкой мыши.

```
<Style x:Key="BigFontButton">
   <Style.Setters>
     ...
  </Style.Setters>
  <Style.Triggers>
    <Trigger Property="Control.IsFocused" Value="True">
      <Setter Property="Control.Foreground" Value="DarkRed" />
    </Trigger>
    <Trigger Property="Control.IsMouseOver" Value="True">
      <Setter Property="Control.Foreground" Value="LightYellow" />
      <Setter Property="Control.FontWeight" Value="Bold" />
    </Trigger>
    <Trigger Property="Button.IsPressed" Value="True">
      <Setter Property="Control.Foreground" Value="Red" />
   </Trigger>
  </Style.Triggers>
</Style>
```

Очевидно, что курсор мыши может быть наведен на кнопку, на которой в текущий момент уже находится фокус. Это не представляет проблемы, потому что данные триггеры изменяют разные свойства. Но при выполнении щелчка на кнопке установить цвет переднего плана (`Foreground`) пытаются одновременно два разных триггера. В этом случае "побеждает" триггер для свойства `Button.IsPressed`, поскольку он идет последним в списке. То, какой из триггеров *срабатывает* первым, не играет никакой роли — например, WPF все равно, что кнопка получает фокус перед выполнением на ней щелчка. Значение имеет только порядок, в котором триггеры перечислены в коде разметки.

На заметку! В этом примере триггеры не являются единственными элементами, которые необходимы для придания кнопке привлекательного внешнего вида. Еще имеется управляющий шаблон кнопки, который ограничивает определенные возможности, касающиеся ее внешнего вида. Для получения наилучших результатов при настройке элементов в такой степени обязательно нужно использовать шаблон элемента управления. Однако шаблоны не заменяют триггеры — на самом деле в них даже часто используются триггеры для получения преимуществ обеих технологий, а именно — элементов управления, которые могут полностью настраиваться *и* реагировать на наведение курсора мыши, щелчки и другие события, изменяя какой-нибудь аспект своего внешнего вида

При желании создать триггер, срабатывающий только при соблюдении сразу нескольких условий, можно применять класс `MutliTrigger`. Он предоставляет коллекцию `Conditions`, которая позволяет определять цепочки комбинаций свойств и значений. Ниже показан пример, в котором форматирование применяется только в том случае, если на кнопке находится фокус и наведен курсор.

```
<Style x:Key="BigFontButton">
<Style.Setters>
...
</Style.Setters>
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
       <Condition Property="Control.IsFocused" Value="True">
       <Condition Property="Control.IsMouseOver" Value="True">
      </MultiTrigger.Conditions>
      <MultiTrigger.Setters>
        <Setter Property="Control.Foreground" Value="DarkRed" />
      </MultiTrigger.Setters>
     </MultiTrigger>
  </Style.Triggers>
</Style>
```

В данном случае порядок, в котором объявляются условия, не имеет значения, потому для того, чтобы цвет фона изменился, вернуть `true` должны они все.

Триггер события

Если обычный триггер ожидает изменения свойства, то триггер события (`EventTrigger`) ожидает возникновения конкретного события. Читатель может предположить, что на этом этапе используются установщики (`Setter`) для измене-

ния элемента, но это не так. Вместо этого триггер событий требует предоставления серии изменяющих элемент управления действий. Эти действия используются для применения анимации.

Хотя об анимационных эффектах подробно будет рассказываться в главе 21, получить общее представление о них можно, рассмотрев простой пример. Для этого ниже показан триггер событий, который ожидает события `MouseEnter` и затем анимирует свойство `FontSize` кнопки, увеличивая размер шрифта до 22 единиц за 0,2 секунды.

```
<Style x:Key="BigFontButtonStyle">
  <Style.Setters>
  ...
  </Style.Setters>

  <Style.Triggers>
    <EventTrigger RoutedEvent="Mouse.MouseEnter">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation
              Duration="0:0:0.2"
              Storyboard.TargetProperty="FontSize"
              To="22" />
              </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Style.Triggers>
    ...
```

В XAML каждый анимационный эффект должен определяться в элементе `Storyboard`, который предоставляет временную шкалу для анимации. Внутри элемента `Storyboard` определяется объект или объекты анимации, которые должны использоваться. Каждый объект анимации выполняет, по сути, одну и ту же задачу: он изменяет свойство зависимостей на протяжении какого-то периода времени.

В данном примере применяется заготовленный класс анимации под названием `DoubleAnimation` (который, как и все классы анимации, расположен в пространстве имен `System.Windows.Media.Animation`). Класс `DoubleAnimation` умеет постепенно изменять любое двойное значение (вроде значения `FontSize`) для получения целевого результата за определенный период времени. Поскольку двойное значение изменяется небольшими частями, на вид шрифт увеличивается постепенно. Фактический размер изменения зависит от общего количества времени и общего объема изменений, которые требуется внести. В данном примере размер шрифта изменяется с текущего значения до 22 единиц за 0,2 секунды. (За счет корректировки свойств класса `DoubleAnimation` можно настроить эти детали более точно и создать анимацию с ускорением или замедлением.)

В отличие от триггеров свойств, действие триггеров событий нужно реверсировать, если требуется, чтобы элемент возвращался в свое исходное состояние. (Дело в том, что по умолчанию завершенная анимация остается активной, сохраняя для свойства последнее значение. Подробнее о том, как работает эта система, будет рассказываться в главе 21.)

Для возврата размеров шрифта в исходное состояние в этом примере в стиле используется триггер событий, который реагирует на событие `MouseLeave` и уменьшает шрифт до исходного размера за две секунды. Указывать целевой размер шрифта в данном случае не нужно — если таковой не указывается, WPF предполагает, что кнопке требуется вернуть тот исходной размер шрифта, который у нее был перед началом выполнения анимации.

```
...
<EventTrigger RoutedEvent="Mouse.MouseLeave">
  <EventTrigger.Actions>
   <BeginStoryboard>
    <Storyboard>
     <DoubleAnimation
      Duration="0:0:1"
      Storyboard.TargetProperty="FontSize" />
    </Storyboard>
   </BeginStoryboard>
  </EventTrigger.Actions>
 </EventTrigger>
</Style.Triggers>
</Style>
```

Интересно, что также можно делать и так, чтобы анимация выполнялась при достижении свойством зависимости определенного значения. Это удобно, если требуется запускать анимацию, а подходящего для этого события нет.

Для применения этого приема необходим триггер свойств, о котором рассказывалось в предыдущем разделе. Предоставлять для него объекты `Setter` не нужно. Вместо этого следует установить свойства `Trigger.EnterActions` и `Trigger.ExitActions`. Оба эти свойства в качестве значения принимают коллекцию действий наподобие действия `BeginStoryboard`, которое запускает анимацию. Действия `EnterActions` выполняются при достижении свойством указанного значения, а действия `ExitActions` — при смене свойством указанного значения на какое-то другое.

Более подробно об использовании триггеров событий и триггеров свойств для запуска анимационных эффектов речь пойдет в главе 21.

Резюме

В этой главе было показано, как стили позволяют определять именованные наборы значений свойств и легко применять их к соответствующему элементу.

Стили являются ключевым компонентом, который позволяет поддерживать множество других функциональных возможностей WPF. Например, стили предоставляют возможность применять новые управляющие шаблоны к ряду элементов управления; использовать различное форматирование в зависимости от текущей темы системы; динамически изменять обложку приложения и улучшать внешний вид элементов с помощью автоматических анимационных эффектов. Подробно эти приемы будут рассматриваться позже в этой книге (шаблоны элементов управления, темы и обложки приложений — в главе 15, анимационные эффекты — в главе 21). Но сначала нужно ознакомиться с еще одной важной темой в WPF — богатыми возможностями двумерного рисования.