DependencyObject и свойства зависимостей

Введение в Dependency Property

В прошлой главе были рассмотрены базовые элементы WPF и их основные свойства. Однако рассмотренные свойства элементов, как например, Width или Height, являются не просто стандартными свойствами языка C#. Они фактически скрывают **свойства зависимостей** или **dependency property**. Без свойств зависимостей были бы невозможны многие ключевые особенности WPF, как привязка данных, стили, анимация и т.д.

Рассмотрим, как они определяются. Возьмем, к примеру, элемент TextBlock, у которого есть свойство **Text**:

```
public class TextBlock : FrameworkElement, IContentHost, IAddChildInternal, IServiceProvider
{
    // свойство зависимостей
    public static readonly DependencyProperty TextProperty;

    static TextBlock()
    {
        // Регистрация свойства
        TextProperty = DependencyProperty.Register(
                    "Text",
                    typeof(string),
                    typeof(TextBlock),
                    new FrameworkPropertyMetadata(
                        string.Empty,
                        FrameworkPropertyMetadataOptions.AffectsMeasure |
                        FrameworkPropertyMetadataOptions.AffectsRender,
                        new PropertyChangedCallback(OnTextChanged),
                        new CoerceValueCallback(CoerceText)));
        // остальной код
    }

    // Обычное свойство .NET  - обертка над свойством зависимостей
    public string Text
    {
        get { return (string) GetValue(TextProperty); }
        set { SetValue(TextProperty, value); }
    }

    private static object CoerceText(DependencyObject d, object value)
    {
        //..............................
    }
    // метод, вызываемый при изменении значения свойства
    private static void OnTextChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
    {
        //..............................
    }
    // остальной код
}
```

Статическое свойство TextProperty является свойством зависимостей, представляя объект **System.Windows.DependencyProperty**. По соглашениям по именованию все свойства зависимостей представляют статические публичные поля (public static) с суффиксом Property.

Затем в статическом конструкторе класса происходит регистрация свойства с помощью метода **DependencyProperty.Register()**, в который передается ряд параметров:

имя свойства (в данном случае "Text"). Как правило, соответствует названию свойства зависимостей без суффикса Property

- тип свойства (в данном случае string)
- тип, который владеет свойством - собственно тот тип, в котором свойство определено или в данном случае тип TextBlock
- Необязательный параметр FrameworkPropertyMetadata устанавливает дополнительные настройки свойства
- В качестве пятого необязательного параметра может использоваться ссылка на метод, который производит валидацию свойства. В данном случае этот параметр опущен.

Первые три обязательных свойства довольно просты, поэтому подробнее остановимся на четвертом необязательном параметре, представляющем объект **FrameworkPropertyMetadata**. Данный объект содержит ряд свойств для конфигурации свойства зависимостей:

- AffectsArrange: если имеет значение true, то свойство зависимостей будет влиять на процесс компоновки элемента
- AffectsMeasure: если имеет значение true, то свойство зависимостей будет учитываться при установке размеров элемента при компоновке
- AffectsParentArrange: если имеет значение true, то свойство зависимостей будет влиять на процесс компоновки в родительском элементе
- AffectsParentMeasure: если имеет значение true, то свойство зависимостей будет учитываться при установке размеров родительского элемента при его компоновке
- AffectsRender: если имеет значение true, то свойство зависимостей будет влиять на рендеринг и визуализацию элемента
- BindsTwoWayByDefault: если имеет значение true, то свойство зависимостей будет использовать двустороннюю привязку данных
- CoerceValueCallback: хранит ссылку на метод, который применяется для проверки допустимости значения до его валидации. Если значение не допустимо, то оно может корректироваться, чтобы соответствовать допустимым диапазонам.
- DefaultValue: устанавливает значение по умолчанию для свойства зависимостей
- Inherits: если имеет значение true, то вложенные элементы применительно к себе могут изменять значение свойства зависимостей. Например, если контейнер Windows задает свойство FontSize, то TextBlock автоматически подхватываетего значение, если в нем самом это свойство не установлено
- IsAnimationProhibited: если имеет значение true, то свойство зависимостей не применяется при анимации
- IsNotDataBindable: если имеет значение true, то свойство зависимостей не будет поддерживать привязку данных

- Journal: если имеет значение true, то значение свойства зависимостей будет журналироваться (сохраняться)
- PropertyChangedCallback: хранит ссылку на метод, который вызывается при изменении значения свойства
- SubPropertiesDoNotAffectRender: если имеет значение true, то элемент не будет перерисовываться, если если какое-то подсвойства у свойства зависимостей изменит свое значение

В данном случае применяется один из конструкторов:

```
new FrameworkPropertyMetadata(string.Empty,
    FrameworkPropertyMetadataOptions.AffectsMeasure | FrameworkPropertyMetadataOptions.AffectsRe
    new PropertyChangedCallback(OnTextChanged), new CoerceValueCallback(CoerceText)))
```

Этот конструктор устанавливает в качестве значени по умолчанию пустую строку, указывает, что при изменении значения элемент будет перерисовываться (собственно, что мы и видим - при изменении значения свойства Text новое значение отображается), и при изменении значения свойства будут вызываться методы OnTextChanged и CoerceText.

Далее после регистрации свойства идет обертка - обычное свойство .NET, которое имеет сеттер и геттер и которое вызывает методы **GetValue** и **SetValue** для получения и установки значения соответственно. Эти методы определены в классе **System.Windows.DependencyObject**, который является базовым для всех элементов WPF, в том числе и для TextBlock.

Последнее обстоятельство привносит ограничение - свойства зависимостей могут быть определены только в тех классах, которые наследутся от DependencyObject.

Кроме того, DependencyObject поддерживает еще ряд свойств для управления свойствами зависимостей:

ClearValue: очищает значение объекта DependencyProperty

InvalidateProperty: повторно вычисляет действующее значение объекта DependencyProperty

ReadLocalValue: считывает значение объекта DependencyProperty

Например:

```
TextBlock textBlock = new TextBlock();
textBlock.Text = "Hello";
string text = (string) textBlock.ReadLocalValue(TextBlock.TextProperty); // Hello
textBlock.ClearValue(TextBlock.TextProperty); // теперь значение отсутствует
```

Провайдеры свойств

В WPF определение значения свойств представляет многоэтапный процесс. На различных стадиях этого процесса применяются различные провайдеры свойств, которые помогают получить значение для свойства зависимостей.

При извлечении значения свойства система использует 10 провайдеров:

1.Получение локального значение свойства (то есть то, которое установлено разработчиком через XAML или через код C#)

2.Вычисление значения с помощью триггеров из шаблона родительского элемента

3.Вычисление значения из шаблона родительского элемента

4.Вычисление значения с помощью триггеров из применяемых стилей

5.Вычисление значения с помощью триггеров из применяемого шаблона

6.Получение значения из сеттеров применяемых стилей

7.Вычисление значения с помощью триггеров из применяемых тем

8.Получение значения из сеттеров применяемых тем

9.Получение унаследованного значения (если свойство FrameworkPropertyMetadata.Inherits имеет значение true)

10.Извлечение значения по умолчанию, которое устанавливается через объект FrameworkPropertyMetadata

Все эти этапы выполняются последовательно сверху вниз. Если на одном этапе было получено значения, то этапы ниже уже не выполняются. Далее мы подробнее рассмотрим триггеры, стили, темы, шаблоны, но в данном случае достаточно понимать, что получение значения свойства - представляет сложный многоэтапный процесс. И даже если в XAML для элемента не установлено значение какого-либо свойства, то все равно оно может иметь значение, полученное на одном из выше перечисленных шагов.

Все десять перечисленных этапов обычно объединяются в одну стадию - **получение базового значения**.

Но кроме получения значения есть еще процесс установки значения. Он вовлекает ряд дополнительных шагов:

Вышеописанные 10 шагов - получение базового значения

Если значение свойства, полученное на шаге 1, представляет собой сложное выражение (например, выражение привязки данных), то WPF вычисляет значение этого выражения и получает конкретный результат

Если для свойства применяется анимация, то далее она используется для получения нового значения

После получения значения WPF применяет делегат **CoerceValueCallback**, который задается в объекте FrameworkPropertyMetadata при регистрации свойства. С помощью метода, на который указывает данный делегат, проверяется, входит ли значение в диапазон допустимых значений. Если не входит, то в заисимости от логики задается новое значение

В конце применяется делегат **ValidateValueCallback** (если он указан при регистрации свойства в качестве пятого параметра), который выполняет валидацию. Метод, на который ссылается делегат, возвращает true при прохождении валидации. Иначе возвращается false и генерируется исключение

Прикрепляемые свойства

Прикрепляемые свойства (attached properties) также являются свойствами зависимостей с той разницей, что они определяются в одном классе, а применяются в другом. Например, при установке столбца или строки грида, в которых размещается элемент управления, используются свойства Grid.Row и Grig.Column, которые как раз и представляют прикрепляемые свойства. То есть эти свойства определены в классе Grid, но используются в других вложенных элементах:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Button x:Name="button1" Content="Hello" Grid.Column="1" Grid.Row="0" />
</Grid>
```

В коде C# установка значения для прикрепленных свойств производится с помощью статических методов в формате Класс.SetСвойство. Например, установим для выше

определенной кнопки столбец и строку грида:

```
Grid.SetRow(button1, 1); // вторая строка
Grid.SetColumn(button1, 1); // второй столбец
```

С помощью методов типа Класс.GetСвойство() мы можем получать в коде значения прикрепляемых свойств:

```
int column = Grid.GetColumn(button1); // получаем номер столбца
```

Если более детально взглянуть на прикрепляемые свойства, то можно увидеть, что их определение немного отличается от стандартных свойств зависимостей. Для регистрации прикрепляемого свойства применяется метод RegisterAttached():

```
Grid.ColumnProperty = DependencyProperty.RegisterAttached(
                        "Column",
                        typeof(int),
                        typeof(Grid),
                        new FrameworkPropertyMetadata(0,
                            new PropertyChangedCallback(Grid.OnCellAttachedPropertyChanged)),
                        new ValidateValueCallback(Grid.IsIntValueNotNegative));
```

Метод RegisterAttached() принимает в прицнипе те же параметры, что и метод Register().

Другое отличие от обычных свойств зависимостей состоит в том, что для прикрепляемых свойств не создается обертка в виде стандартного свойства C#. Вместо этого используется пара статических методов SetСвойство() GetСвойство():

```
public static int GetColumn(UIElement element)
{
    if (element == null)
    {
        throw new ArgumentNullException(...);
    }
    return (int)element.GetValue(Grid.ColumnProperty);
}
public static void SetColumn(UIElement element, int value)
{
    if (element == null)
    {
        throw new ArgumentNullException();
    }
    element.SetValue(Grid.ColumnProperty, value);
}
```

В принципе выше мы уже рассмотрели их применение на примере Grid.SetColumn/Grid.SetRow и Grid.GetColumn.

Создание свойств зависимостей

В предыдущих темах мы рассмотрели теоретические основы свойств зависимостей, теперь посмотрим, как мы можем определять какие-то свои свойства зависимостей. Итак, определим в нашем проекте новый класс Phone:

```
public class Phone : DependencyObject
{
    public static readonly DependencyProperty TitleProperty;
    public static readonly DependencyProperty PriceProperty;

    static Phone()
    {
        TitleProperty = DependencyProperty.Register("Title", typeof(string), typeof(Phone));
        PriceProperty = DependencyProperty.Register("Price", typeof(int), typeof(Phone));
    }
    public string Title
    {
        get { return (string)GetValue(TitleProperty); }
        set { SetValue(TitleProperty, value); }
    }
    public int Price
    {
        get { return (int)GetValue(PriceProperty); }
        set { SetValue(PriceProperty, value); }
    }
}
```

Если мы хотим применять свойства зависимостей, то нам надо унаследовать свой класс от абстрактного класса DependencyObject. В нашем классе мы определяем два свойства зависимостей: TitleProperty и PriceProperty. Обратите внимание, что они объявляются с модификаторами public static readonly.

Затем свойства регистрируются в статическом конструкторе нашего класса с помощью метода **Register**. И в конце для них создаются обычные свойства-обертки, в которых мы получаем доступ к значению свойств с помощью методов GetValue и SetValue.

Теперь используем наш класс. Для этого определим следующую разметку XAML:

```
<Window x:Class="DependencyApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:DependencyApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="300" FontSize="20">
    <Window.Resources>
        <local:Phone Price="600" Title="iPhone 6S" x:Key="iPhone6s" />
    </Window.Resources>
    <Grid x:Name="grid1" DataContext="{StaticResource iPhone6s}">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <TextBlock Text="Модель" />
        <TextBlock Text="{Binding Title}" Grid.Column="1"  />
        <TextBlock Text="Цена" Grid.Row="1" />
        <TextBox Text="{Binding Price, Mode=TwoWay}" Grid.Column="1" Grid.Row="1"  />
        <Button Content="Check" Grid.Row="2" Grid.Column="2" Click="Button_Click" />
    </Grid>
</Window>
```

Здесь в ресурсах окна определяется объект Phone с установкой его свойств Price и Name:

```
<local:Phone Price="600" Title="iPhone 6S" x:Key="iPhone6s" />
```

Данный ресурс имеет ключ iPhone6s, по которому мы можем к нему обратиться. Далее для контейнера Grid мы задаем этот ресурс как контекст данных:

```
<Grid x:Name="grid1" DataContext="{StaticResource iPhone6s}">
```

Установка контекста данных позволяет внутри грида привязать отдельные элеметы к свойства ресурса, то есть объекта Phone:

```
<TextBox Text="{Binding Price, Mode=TwoWay}" Grid.Column="1" Grid.Row="1"  />
```

То же самое и для элемента TextBlock. Однако для TextBox у нас действует не просто привязка, а двусторонняя привязка, обозначенная параметром Mode=TwoWay. А это значит, что любые

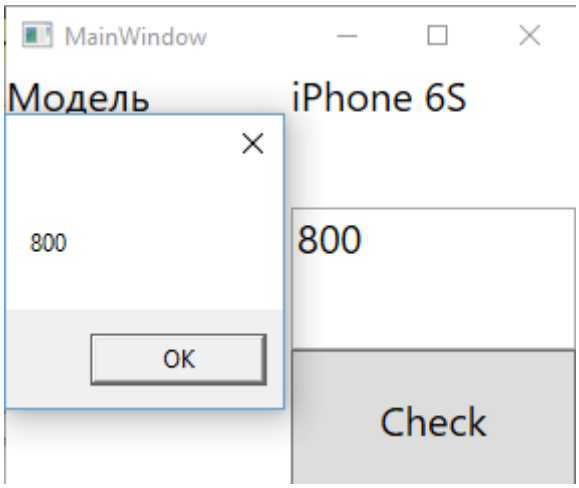
изменения свойства Price в ресурсе будут отображаться в текстовом поле. И наоборот - любые изменения в текстовом поле будут менять значения в ресурсе.

В этом состоит одно из преимуществ использования свойств зависимостей - для обычных свойств подобные привязки бы не работали.

Для проверки значения ресурса я добавил кнопку, у которой установил обработчик нажатия - Button_Click. И также добавим код этого обработчика в файл кода C#:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Phone phone = (Phone)this.Resources["iPhone6s"]; // получаем ресурс по ключу
    MessageBox.Show(phone.Price.ToString());
}
```

Запустим приложение, изменим значение в текстовом поле, например, на 800 и нажмем на кнопку:



Добавление валидации

WPF предоставляет два способа валидации значения свойства:

- ValidateValueCallback: делегат, который возвращает true, если значение проходит валидацию, и false - если не проходит
- CoerceValueCallback: делегат, который может подкорректировать уже существующее значение свойства, если оно вдруг не попадает в диапазон допустимых значений

При установке значения сначала срабатывает делегат ValidateValueCallback, который возвращает true, если значение проходит валидацию. Далее срабатывает делегат

CoerceValueCallback, который может модифицировать это значение, если оно вдруг является некорректным. В случае, если эти два делегата успешно отработали, то срабатывает делегат PropertyChangedCallback, который извещает систему об изменении значени свойства.

При этом нам необязательно использовать сразу оба эти делеата, можно применять лишь один из них, а можно их комбинировать вместе.

Вначале применим делегат ValidateValueCallback. Для этого изменим код класса Phone на следующий:

```
public class Phone : DependencyObject
{
    public static readonly DependencyProperty TitleProperty;
    public static readonly DependencyProperty PriceProperty;

    static Phone()
    {
        TitleProperty = DependencyProperty.Register("Title", typeof(string), typeof(Phone));

        FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();

        PriceProperty = DependencyProperty.Register("Price", typeof(int), typeof(Phone), metadat
            new ValidateValueCallback(ValidateValue));
    }

    private static bool ValidateValue(object value)
    {
        int currentValue = (int)value;
        if (currentValue >= 0) // если текущее значение от нуля и выше
            return true;
        return false;
    }

    public string Title
    {
        get { return (string)GetValue(TitleProperty); }
        set { SetValue(TitleProperty, value); }
    }
    public int Price
    {
        get { return (int)GetValue(PriceProperty); }
        set { SetValue(PriceProperty, value); }
    }
}
```

Делегат ValidateValueCallback указывает на метод ValidateValue, который в качестве параметра принимает новое значение свойства. Если оно равно нулю или выше, так как отрицательные

цены не имеют смысла, то значение проходит валидацию, и метод возвращает true.

При подобных проверках надо учитывать, что в процессе запуска приложения значения ресурса Phone устанавливаются два раза. Первый раз свойство Price получает значение по умолчанию для типов int, то есть число 0. И только потом на следующем этапе инициализации оно получает число 600, которое и установлено в ресурсе Phone. Поэтому если мы, к примеру, изменим логику валидации:

```
if (currentValue > 0)
    return true;
```

То мы получим ошибку. Ведь число 0 теперь не является допустимым, но на начальной инициализации объекта Phone, именно 0 устанавливается в качестве значения для свойства Price. Поэтому подобные моменты надо учитывать.

Теперь, если мы запустим приложение и попробуем установить число меньше 0, то оно не пройдет валидацию, и ресурс Phone сохранит старое значение для свойства Price.

Теперь применим делегат **CoerceValueCallback**:

```
public class Phone : DependencyObject
{
    public static readonly DependencyProperty TitleProperty;
    public static readonly DependencyProperty PriceProperty;

    static Phone()
    {
        TitleProperty = DependencyProperty.Register("Title", typeof(string), typeof(Phone));

        FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();
        metadata.CoerceValueCallback = new CoerceValueCallback(CorrectValue);

        PriceProperty = DependencyProperty.Register("Price", typeof(int), typeof(Phone), metadat
            new ValidateValueCallback(ValidateValue));
    }

    private static object CorrectValue(DependencyObject d, object baseValue)
    {
        int currentValue = (int)baseValue;
        if (currentValue > 1000)  // если больше 1000, возвращаем 1000
            return 1000;
        return currentValue; // иначе возвращаем текущее значение
    }

    private static bool ValidateValue(object value)
    {
        int currentValue = (int)value;
        if (currentValue >= 0) // если текущее значение от нуля и выше
            return true;
        return false;
    }

    public string Title
    {
        get { return (string)GetValue(TitleProperty); }
        set { SetValue(TitleProperty, value); }
    }
    public int Price
    {
        get { return (int)GetValue(PriceProperty); }
        set { SetValue(PriceProperty, value); }
    }
}
```

В данном случае мы будем корректировать значение свойства Price, так как оно изменяется через текстовое поле. Для этого делегат CoerceValueCallback будет вызывать метод CorrectValue. Данный метод должен принимать два параметра: **DependencyObject** -

валидируемый объект (в данном случае объект Phone) и **object** - новое значение для свойства Price (либо для другого устанавливаемого свойства).

В самом методе CorrectValue() мы получаем новое значение и модифицируем его, если оно некорректно.

Единственное, что надо учитывать при работе с делегатом CoerceValueCallback, то, что он вызывается только тогда, когда вызов делегата ValidateValueCallback возвращает значение true.

Больше нам ничего менять не надо, разметка xaml остается прежней. И если мы запустим приложение и введем в текстоое поле некорректное значение, то сработает этот делегат, который подкорректирует значение.