# DependencyObject и свойства зависимостей

## Введение в Dependency Property

Вами 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**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | 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:

|  |  |
| --- | --- |
| 1  2  3 | new FrameworkPropertyMetadata(string.Empty,      FrameworkPropertyMetadataOptions.AffectsMeasure | FrameworkPropertyMetadataOptions.AffectsRender,      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:

|  |  |
| --- | --- |
| 1  2  3  4 | 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:

1. Вышеописанные 10 шагов - получение базового значения
2. 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
3. Если для свойства применяется анимация, то далее она используется для получения нового значения
4. 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
5. 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