# Модель событий

## Маршрутизация событий

Ранее рассматривались различные элементы управления. Но чтобы с ними взаимодействовать, нам надо использовать модель событий. WPF в отличие от других технологий, например, от Windows Forms, предлагает новую концепцию событий - **маршрутизированные события** (**routed events**).

Для элементов управления в WPF определено большое количество событий, которые условно можно разделить на несколько групп:

* События клавиатуры
* События мыши
* События стилуса
* События сенсорного экрана/мультитач
* События жизненного цикла

### Подключение обработчиков событий

Подключить обработчики событий можно декларативно в файле xaml-кода, а можно стандартным способом в файле отделенного кода.

Декларативное подключение:

|  |  |
| --- | --- |
| 1 | <Button x:Name="Button1" Content="Click" Click="Button\_Click" /> |

И подключим еще один обработчик в коде, чтобы при нажатии на кнопку срабатывали сразу два обработчика:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public partial class MainWindow : Window  {      public MainWindow()      {          InitializeComponent();          Button1.Click += Button1\_Click;      }      // обработчик, подключаемый в XAML      private void Button\_Click(object sender, RoutedEventArgs e)      {          MessageBox.Show("Hi from Button\_Click");      }      // обработчик, подключаемый в конструкторе      private void Button1\_Click(object sender, RoutedEventArgs e)      {          MessageBox.Show("Hi from Button1\_Click");      }  } |

### Определение маршрутизированных событий

Определение маршрутизированных событий отличается от стандартного определения событий в языке C#. Для определения маршрутизированных событий в классе создвалось статическое поле по типу **RoutedEvent**:

|  |  |
| --- | --- |
| 1 | public static RoutedEvent СобытиеEvent |

Это поле, как правило, имеет суффикс *Event*. Затем это событие регистрируется в статическом конструкторе.

И также класс, в котором создается событие, как правило определяет объект-обертку над обычным событием. В этой обертке с помощью метода **AddHandler** происходит добавление обработчика для данного события, а с помощью метода **RemoveHandler** - удаление обработчика.

К примеру, возьмем встроенные класс ButtonBase - базовый класс для всех кнопок, который определяет ряд событий, в том числе событие **Click**:

|  |  |
| --- | --- |
| 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 | public abstract class ButtonBase : ContentControl, ...  {      // определение событие      public static readonly RoutedEvent ClickEvent;        static ButtonBase()      {          // регистрация маршрутизированного события          ButtonBase.ClickEvent = EventManager.RegisterRoutedEvent("Click",              RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(ButtonBase));          //................................      }      // обертка над событием      public event RoutedEventHandler Click      {          add          {              // добавление обработчика              base.AddHandler(ButtonBase.ClickEvent, value);          }          remove          {              // удаление обработчика              base.RemoveHandler(ButtonBase.ClickEvent, value);          }      }    // остальное содержимое класса  } |

Маршрутизированные события регистрируются с помощью метода EventManager.RegisterRoutedEvent(). В этот метод передаются последовательно имя события, тип события (поднимающееся, прямое, опускающееся), тип делегата, предназначенного для обработки события, и класс, который владеет этим событием.

### Маршрутизация событий

Модель событий WPF отличается от событий WinForms не только декларативным подключением. События, возникнув на одном элементе, могут обрабатываться на другом. События могут подниматься и опускаться по дереву элементов.

Так, маршрутизируемые события делятся на три вида:

* **Прямые** (direct events) - они возникают и отрабытывают на одном элементе и никуда дальше не передаются. Действуют как обычные события.
* **Поднимающиеся** (bubbling events) - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру и далее, пока не достигнет наивысшего родителя в дереве элементов.
* **Опускающиеся, туннельные** (tunneling events) - начинает отрабатывать в корневом элементе окна приложения и идет далее по вложенным элементам, пока не достигнет элемента, вызвавшего это событие.

Все маршрутизируемые события используют класс **RoutedEventArgs** (или его наследников), который представляет доступ к следующим свойствам:

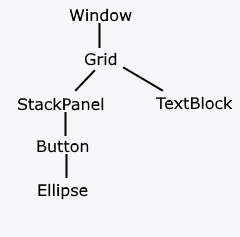
* **Source**: элемент логического дерева, являющийся источником события.
* **OriginalSource**: элемент визуального дерева, являющийся источником события. Обычно то же самое, что и Source
* **RoutedEvent**: представляет имя события
* **Handled**: если это свойство установлено в True, событие не будет подниматься и опускаться, а ограничится непосредственным источником.

### Поднимающиеся события

Допустим, у нас имеется такая разметка xaml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <Window x:Class="EventsApp.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:EventsApp"          mc:Ignorable="d"          Title="MainWindow" Height="250" Width="400">      <Grid>          <Grid.ColumnDefinitions>              <ColumnDefinition Width="Auto" />              <ColumnDefinition />          </Grid.ColumnDefinitions>          <StackPanel Grid.Column="0" VerticalAlignment="Center" MouseDown="Control\_MouseDown">              <Button x:Name="button1" Width="80" Height="50" MouseDown="Control\_MouseDown" Margin="10" >                  <Ellipse Width="30" Height="30" Fill="Red" MouseDown="Control\_MouseDown" />              </Button>          </StackPanel>          <TextBlock x:Name="textBlock1" Grid.Column="1" Padding="10" />      </Grid>  </Window> |

В данном случае мы получаем следующее дерево элементов:

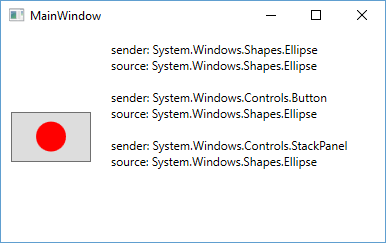


Три элемента имеют привязку к одному обработчику события, которое возникает при нажатии правой кнопки мыши или тачпада. Определим этот обработчик в файле кода C#:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private void Control\_MouseDown(object sender, MouseButtonEventArgs e)  {      textBlock1.Text = textBlock1.Text + "sender: " + sender.ToString() + "\n";      textBlock1.Text = textBlock1.Text + "source: " + e.Source.ToString() + "\n\n";  } |

Обработчик в данном случае выводит информацию о событии в текстовый блок.

И так как это событие **MouseDown** является поднимающимся, то при нажатии правой кнопкой мыши на элемент самого нижнего уровня - Ellipse, событие MouseDown будет подниматься к контейнерам и отработает три раза последовательно для всех элементов Ellipse, Button, StackPanel:



### **Туннельные события**

Туннельные события действуют прямо противоположным способом. Как правило, все они начинаются со слова Preview. Возьмем выше приведенный пример и заменим событие MouseDown на PreviewMouseDown

|  |  |
| --- | --- |
| 1  2  3  4  5 | <StackPanel Grid.Column="0" VerticalAlignment="Center" PreviewMouseDown="Control\_MouseDown">      <Button x:Name="button1" Width="80" Height="50" PreviewMouseDown="Control\_MouseDown" Margin="10" >          <Ellipse Width="30" Height="30" Fill="Red" PreviewMouseDown="Control\_MouseDown" />      </Button>  </StackPanel> |

Нажмем на элемент Ellipse. Тогда событие сначала отработает на элементе StackPanel и затем последовательно на элементе Button и закончится на элементе Ellipse.

### C:\Users\KAB6-Teacher\Pictures\6.2.png

### **Прикрепляемые события (Attached events)**

Если у нас есть несколько элементов одного и того же типа и мы хотим привязать их к одному событию, то мы можем воспользоваться прикрепляемыми событиями.

Так, ранее у нас была группа элементов RadioButton и, чтобы вывести при выборе любого из них выбранное значение, нам приходилось у каждого определять обработчик события. Но это не оптимальная модель. И именно здесь мы и применим прикрепляемые события:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <StackPanel x:Name="menuSelector" Grid.Column="0" RadioButton.Checked="RadioButton\_Click">      <RadioButton GroupName="menu" Content="Салат Оливье" />      <RadioButton GroupName="menu" Content="Котлета по-киевски" />      <RadioButton GroupName="menu" Content="Пицца с овощами" />      <RadioButton GroupName="menu" Content="Мясной рулет" />  </StackPanel> |

Обработчик для прикрепляемого события задается в формате *Имя\_класса.Название\_события="Обработчик"*. Здесь атрибут RadioButton.Checked="RadioButton\_Click" закрепляет все радиокнопки на StackPanel за одним обработчиком. Тогда в коде можно прописать:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private void RadioButton\_Click(object sender, RoutedEventArgs e)  {      RadioButton selectedRadio = (RadioButton)e.Source;      textBlock1.Text = "Вы выбрали: " + selectedRadio.Content.ToString();  } |

И на текстовый блок выводится выбранный пункт.

Также обработчик для прикрепляемого события мы можем задать в коде c#:

|  |  |
| --- | --- |
| 1 | menuSelector.AddHandler(RadioButton.CheckedEvent, new RoutedEventHandler(RadioButton\_Click)); |

## События клавиатуры

К событиям клавиатуры можно отнести следующие события:

|  |  |  |
| --- | --- | --- |
| **Событие** | **Тип события** | **Описание** |
| **KeyDown** | Поднимающееся | Возникает при нажатии клавиши |
| **PreviewKeyDown** | Туннельное | Возникает при нажатии клавиши |
| **KeyUp** | Поднимающееся | Возникает при освобождении клавиши |
| **PreviewKeyUp** | Туннельное | Возникает при освобождении клавиши |
| **TextInput** | Поднимающееся | Возникает при получении элементом текстового ввода (генерируется не только клавиатурой, но и стилусом) |
| **PreviewTextInput** | Туннельное | Возникает при получении элементом текстового ввода |

Большинство событий клавиатуры (KeyUp/PreviewKeyUp, KeyDown/PreviewKeyDown) принимает в качестве аргумента объект KeyEventArgs, у которого можно отметить следующие свойства:

* **Key** позволяет получить нажатую или отпущенную клавишу
* **SystemKey** позволяет узнать, нажата ли системная клавиша, например, Alt
* **KeyboardDevice** получает объект KeyboardDevice, представляющее устройство клавиатуры
* **IsRepeat** указывает, что клавиша удерживается в нажатом положении
* **IsUp** и **IsDown** указывает, была ли клавиша нажата или отпущена
* **IsToggled** указывает, была ли клавиша включена - относится только к включаемым клавишам Caps Lock, Scroll Lock, Num Lock

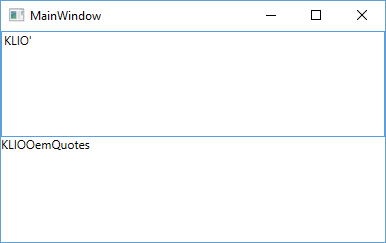
Например, обработаем событие KeyDown для текстового поля и выведем данные о нажатой клавише в текстовый блок:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <Window x:Class="EventsApp.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:EventsApp"          mc:Ignorable="d"          Title="MainWindow" Height="250" Width="400">      <Grid>          <Grid.RowDefinitions>              <RowDefinition />              <RowDefinition />          </Grid.RowDefinitions>          <DockPanel >              <TextBox KeyDown="TextBox\_KeyDown"  />          </DockPanel>          <TextBlock x:Name="textBlock1" Grid.Row="1" />      </Grid>  </Window> |

А в файле кода пропишем обработчик TextBox\_KeyDown:

|  |  |
| --- | --- |
| 1  2  3  4 | private void TextBox\_KeyDown(object sender, KeyEventArgs e)  {      textBlock1.Text += e.Key.ToString();  } |

Здесь в текстовый блок добавляется текстовое представление нажатой клавиши в текстовом поле:



Правда, в данном случае реальную пользу от текстового представления мы можем получить только для алфавитно-цифровых клавиш, в то время как при нажатии специальных клавиш или кавычек будут добавляться их полные текстовые представления, например, для кавычек - OemQuotes.

Если нам надо отловить нажатие какой-то опредленной клавиши, то мы можем ее проверить через перечисление **Key**:

|  |  |
| --- | --- |
| 1  2  3  4 | if (e.Key == Key.OemQuotes)      textBlock1.Text += "'"; // добавляем кавычки  else      textBlock1.Text += e.Key.ToString(); |

Объект KeyboardDevice позволяет нам получить ряд дополнительных данных о собыиях клавиатуры через ряд свойств и методов:

* **Modifiers** позволяет узнать, какая клавиша была нажата вместе с основной (Ctrl, Shift, Alt)
* **IsKeyDown()** определяет, была ли нажата определенная клавиша во время события
* **IsKeyUp()** позволяет узнать, была ли отжата определенная клавиша во время события
* **IsKeyToggled()** позволяет узнать, была ли во время события включена клавиша Caps Lock, Scroll Lock или Num Lock
* **GetKeyStates()** возвращает одно из значений перечисления KeyStates, которое указывает на состояние клавиши

Пример использования KeyEventArgs при одновременном нажатии двух клавиш Shift и F1:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private void TextBox\_KeyDown(object sender, KeyEventArgs e)  {      if (e.KeyboardDevice.Modifiers == ModifierKeys.Shift && e.Key == Key.F1)          MessageBox.Show("HELLO");  } |

События TextInput/PreviewTextInput в качестве параметра принимают объект **TextCompositionEventArgs**. Из его свойств стоит отметить, пожалуй, только свойство Text, которое получает введенный текст, именно текст, а не текстовое представление клавиши. Для этого добавим к текстовому полю обработчик:

|  |  |
| --- | --- |
| 1 | <TextBox Height="40" Width="260" PreviewTextInput="TextBox\_TextInput" /> |

И определим обработчик в файле кода:

|  |  |
| --- | --- |
| 1  2  3  4 | private void TextBox\_TextInput(object sender, TextCompositionEventArgs e)  {      textBlock1.Text += e.Text;  } |

Причем в данном случае я обрабатываю именно событие PreviewTextInput, а не TextInput, так как элемент TextBox подавляет событие TextInput, и вместо него генерирует событие TextChanged. Для большинства других элементов управления, например, кнопок, событие TextInput прекрасно срабатывает.

### **Валидация текстового ввода**

События открывают нам большой простор для валидации текстового ввода. Нередко при вводе используются те или иные ограничения: нельзя вводить цифровые символы или, наоборот, можно только цифровые и т.д. Посмотрим, как мы можем провети валидацию ввода. К примеру, возьмем ввод номера телефона. Сначала зададим обработку двух событий в xaml:

|  |  |
| --- | --- |
| 1 | <TextBox PreviewTextInput="TextBox\_PreviewTextInput" PreviewKeyDown="TextBox\_PreviewKeyDown"  /> |

И определим в файле кода обработчики:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | private void TextBox\_PreviewTextInput(object sender, TextCompositionEventArgs e)  {      int val;      if (!Int32.TryParse(e.Text, out val) && e.Text!="-")      {          e.Handled = true; // отклоняем ввод      }  }    private void TextBox\_PreviewKeyDown(object sender, KeyEventArgs e)  {      if (e.Key == Key.Space)      {          e.Handled = true; // если пробел, отклоняем ввод      }  } |

Для валидации ввода нам надо использовать обработчики для двух событий - PreviewKeyDown и PreviewTextInput. Дело в том, что нажатия не всех клавиш PreviewTextInput обрабатывает. Например, нажатие на клавишу пробела не обрабтывается. Поэтому также применяется обработка и PreviewKeyDown.

Сами обработчики проверяют ввод и если ввод соответствует критериям, то он отклоняется с помощью установки **e.Handled = true**. Тем самым мы говорим, что событие обработано, а введенные текстовые сиволы не будут появляться в текстовом поле. Конкретно в данном случае пользователь может вводить только цифровые символы и пробел в соответствии с форматом телефонного номера.

## События мыши и фокуса

В WPF для мыши определены следующие события:

|  |  |  |
| --- | --- | --- |
| **Событие** | **Тип события** | **Описание** |
| **GotMouseCapture** | Поднимающееся | Возникает при получении фокуса с помощью мыши |
| **LostMouseCapture** | Поднимающееся | Возникает при потере фокуса с помощью мыши |
| **MouseEnter** | Прямое | Возникает при вхождении указателя мыши в пределы элемента |
| **MouseLeave** | Прямое | Возникает, когда указатель мыши выходит за пределы элемента |
| **MouseLeftButtonDown** | Поднимающееся | Возникает при нажатии левой кнопки мыши |
| **PreviewMouseLeftButtonDown** | Туннельное | Возникает при нажатии левой кнопки мыши |
| **MouseLeftButtonUp** | Поднимающееся | Возникает при освобождении левой кнопки мыши |
| **PreviewMouseLeftButtonUp** | Туннельное | Возникает при освобождении левой кнопки мыши |
| **MouseRightButtonDown** | Поднимающееся | Возникает при нажатии правой кнопки мыши |
| **PreviewMouseRightButtonDown** | Туннельное | Возникает при нажатии правой кнопки мыши |
| **MouseRightButtonUp** | Поднимающееся | Возникает при освобождении правой кнопки мыши |
| **PreviewMouseRightButtonUp** | Туннельное | Возникает при освобождении правой кнопки мыши |
| **MouseDown** | Поднимающееся | Возникает при нажатии кнопки мыши |
| **PreviewMouseDown** | Туннельное | Возникает при нажатии кнопки мыши |
| **MouseUp** | Поднимающееся | Возникает при освобождении кнопки мыши |
| **PreviewMouseUp** | Туннельное | Возникает при освобождении кнопки мыши |
| **MouseMove** | Поднимающееся | Возникает при передвижении указателя мыши |
| **PreviewMouseMove** | Туннельное | Возникает при передвижении указателя мыши |
| **MouseWheel** | Поднимающееся | Возникает при передвижении колесика мыши |
| **PreviewMouseWheel** | Туннельное | Возникает при передвижении колесика мыши |

Если вдруг мы не хотим, чтобы элемент генерировал события мыши, то мы можем у него установить свойство IsHitTestVisible="False"

Большинство обработчиков событий мыши в качестве параметра получают объект MouseEventArgs, имеющий ряд интересных свойств и методов, которые мы можем использовать:

* **ButtonState**: возвращает состояние кнопки мыши. Хранит одно из значений перечисления **MouseButtonState**:
  + Pressed: кнопка наата
  + Released: кнопка отжата
* **ChangedButton**: получает кнопку, которая ассоциирована с данным событием. Хранит одно из значений перечисления **MouseButton**:
  + Left: левая кнопка мыши
  + Middle: средняя кнопка мыши
  + Right: правая кнопка мыши
  + XButton1: дополнительная кнопка мыши
  + XButton2: дополнительная кнопка мыши
* **ClickCount**: хранит число сделанных нажатий
* **LeftButton**: хранит состояние левой кнопки мыши в виде MouseButtonState
* **MiddleButton**: хранит состояние средней кнопки мыши в виде MouseButtonState
* **RightButton**: хранит состояние правой кнопки мыши в виде MouseButtonState
* **XButton1**: хранит состояние первой дополнительной кнопки
* **XButton2**: хранит состояние второй дополнительной кнопки
* **GetPosition()**: метод, который возвращает координаты нажатия в виде объекта Point

Например, используем метод GetPosition(). Для этого установим для грида обработчик:

|  |  |
| --- | --- |
| 1 | <Grid MouseDown="Grid\_MouseDown"> |

И определим этот обработчик:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private void Grid\_MouseDown(object sender, MouseButtonEventArgs e)  {      Point p = e.GetPosition(this);      MessageBox.Show("Координата x=" +p.X.ToString()+ " y="+p.Y.ToString());  } |

### События перетаскивания

События перетаскивания (drag & drop) связаны с перетаскиванием элементов, когда пользователь, нажимая на элементе мышкой и удерживая мышь нажатой, перемещает указатель на другой элемент, тем самым перемещая на этот элемент ранее нажатый.

|  |  |  |
| --- | --- | --- |
| **Событие** | **Тип события** | **Описание** |
| **DragEnter** | Поднимающееся | Возникает при перетаскивании при вхождении указателя мыши в пределы элемента |
| **DragOver** | Поднимающееся | Возникает при перемещении курсора в пределах границ элемента управления |
| **DragLeave** | Поднимающееся | Возникает при перемещении курсора мыши за пределы элемента |
| **Drop** | Поднимающееся | Возникает при завершении перетаскивания |
| **PreviewDragEnter** | Тунельное | Возникает при перетаскивании при вхождении указателя мыши в пределы элемента |
| **PreviewDragOver** | Тунельное | Возникает при перемещении курсора в пределах границ элемента управления |
| **PreviewDragLeave** | Тунельное | Возникает при перемещении курсора мыши за пределы элемента |
| **PreviewDrop** | Тунельное | Возникает при завершении перетаскивания |

Эти события используют объект **DragEventArgs**, который имеет ряд свойств и методов:

* **GetPosition**: возвращает позицию мыши
* **Data**: объект, представляющий буфер обмена - то есть те данные, которые перемещаются
* **Effects** и **AllowedEffects**: представляют эффект перетаскивния. Хранят одно из значений перечисления **DragDropEffects**:
  + All: данные копируются из источника в целевой элемент с удалением из источника
  + Copy: данные просто копируются из источника в целевой элемент
  + Link: данные из источника связываются с данными из целевого элемента
  + Move: данные перемещаются из источника в целевой элемент
  + None: отсутствие эффекта
  + Scroll: данные прокручиваются при копировании в целевой элемент
* **KeyStates**: хранит значение из перечисления **DragDropKeyStates**, которое указывает, какая клавиша клавиатуры или мыши зажата во время перетаскивания: LeftMouseButton, RightMouseButton, MiddleMouseButton, ShiftKey, ControlKey, AltKey, None

Посмотрим на примере. Допустим, у нас следующая разметка xaml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <Window x:Class="EventsApp.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:EventsApp"          mc:Ignorable="d"          Title="MainWindow" Height="250" Width="400">      <Grid>          <Grid.RowDefinitions>              <RowDefinition />              <RowDefinition />          </Grid.RowDefinitions>          <DockPanel >              <TextBox x:Name="textBox1" MouseDown="textBox1\_MouseDown" />          </DockPanel>            <Button x:Name="button1" Grid.Row="1" AllowDrop="True" Drop="button1\_Drop"  />      </Grid>  </Window> |

Здесь мы будем перемещать введенный текст из текстового поля на кнопку. Чтобы кнопка могла принимать перемещаемые объекты, установим ее свойство AllowDrop="True". Одни элементы на другие, то нам надо у элементов-приемников всегда устанавливать данное свойство.

Здесь также подключены два обработчика события, которые мы зададим в коде C#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | private void textBox1\_MouseDown(object sender, MouseButtonEventArgs e)  {      DragDrop.DoDragDrop(textBox1, textBox1.Text, DragDropEffects.Copy);  }    private void button1\_Drop(object sender, DragEventArgs e)  {      button1.Content = e.Data.GetData(DataFormats.Text);  } |

Чтобы захватить элемент для переноса, нам надо вызвать метод DragDrop.DoDragDrop, который в качестве первого параметра принимает элемент-источник, с которого идет перетаскивание, второй параметр - что перетаскиваем (в данном случае текст), и третий параметр - тип эффекта. Так как в данном случае у нас копирование, то устанавливаем DragDropEffects.Copy. Также мы можем использовать и другие константы: Move, None, Link, Scroll, All.

Введем текст в текстовое поле, выделим его, нажмем левой кнопкой и, не отпуская, переместим курсор в пределы кнопки. И отпустим. Здесь уже возникнет событие Drop кнопки, обработчик которого также прост: мы присваиваем ее содержимому данные перетаскивания. И поскольку мы перетаскиваем текст, то в качестве параметра выставляем **DataFormats.Text**

### **События получения/потери фокуса**

При обработке событий фокуса следует помнить, что элемент может получать фокус только в том случае, если его свойство Focusable имеет значение true.

Чтобы программным способом передать элементу фокус, надо вызвать у него методы Focus или MoveFocus:

|  |  |
| --- | --- |
| 1 | textBox1.Focus(); |

|  |  |  |
| --- | --- | --- |
| **Событие** | **Тип события** | **Описание** |
| **GotFocus** | Поднимающееся | Возникает при получении фокуса |
| **LostFocus** | Поднимающееся | Возникает при потере фокуса |
| **GotKeyboardFocus** | Поднимающееся | Возникает при получении фокуса с помощью клавиатуры |
| **PreviewGotKeyboardFocus** | Туннельное | Возникает при получении фокуса с помощью клавиатуры |
| **LostKeyboardFocus** | Поднимающееся | Возникает при потере фокуса с помощью клавиатуры |
| **PreviewLostKeyboardFocus** | Туннельное | Возникает при потере фокуса с помощью клавиатуры |

Обработем событие получения фокуса для текстового поля:

|  |  |
| --- | --- |
| 1 | <TextBox GotFocus="TextBox\_GotFocus"  /> |

В файле кода пропишем обработчик:

|  |  |
| --- | --- |
| 1  2  3  4 | private void TextBox\_GotFocus(object sender, RoutedEventArgs e)  {      MessageBox.Show("Получение фокуса");  } |