

Разработка приложений на платформе .NET

Лекция 17

События в WPF. Команды

Сегодня

- События
- Команды

События в WPF

- Расширенная модель события - маршрутизируемое событие
- Позволяют возникать в одном элементе управления, а обрабатываться в другом
- Типы событий:
 - **Прямые**. Иницируются только в том элементе, в котором произошло
 - Пример: `MouseLeave`
 - **Пузырьковые**. Сначала иницируются элементе управления, в котором произошло, а затем в каждом предшественнике в визуальном дереве
 - Пример: `MouseDown`, `MouseUp`
 - **Туннельные**. Иницируются сначала в контейнере высшего уровня в визуальном дереве, а затем спускается по всем элементам к текущему.
 - Пример: `PreviewMouseDown`, `PreviewMouseUp`

Прикрепление обработчика

● В XAML:

- Атрибут – событие. Атрибут – название события, значение имя метода обработчика события
- `<Button Click="Button_Click">Hi</Button>`

● В C#

- обычная подписка на событие:
 - `button1.Click += new RoutedEventHandler(button1_Click);`
- Расширенная подписка (может быть любое событие)
 - `button1.AddHandler(Button.ClickEvent, new RoutedEventHandler(button1_Click));`

● Сигнатура обработчика события:

- `public delegate void RoutedEventHandler(object sender, RoutedEventArgs e)`
- Если событие передает какую-то дополнительную информацию, например, о нажатой клавише, то используются классы наследники от `RoutedEventArgs`.
- `RoutedEventArgs` наследник от `EventArgs` и поэтому сохраняется общая структура событий .NET

Класс RoutedEventArgs

- Все маршрутизированные события включают в своих сигнатурах экземпляр класса **RoutedEventArgs** (или его наследника)
- Свойства **RoutedEventArgs**
 - **Source, OriginalSource** – возвращают объект, первоначально инициировавший событие. **OriginalSource** – содержит информацию о точном месте генерации события, находящемся в шаблоне элемента управления. **Source** – возвращает объект, сгенерировавший событие, не забираясь в шаблон
 - **RoutedEvent** – представляет само событие
 - **Handled**. Если установить **True**, то событие считается обработанным и не распространяется далее по визуальному дереву. Если установить в **True** для туннельного (**Preview...**) события, то и соответствующее пузырьковое считается обработанным.
- **Sender** в обработчике события содержит элемент, сгенерировавший событие в данный момент, а не элемент первоначально сгенерировавший событие

Прикрепленные события

- Для централизованной обработки события в элементе не имеющем такого события
 - Например, `Grid`, не имеет событие `Click` и подписка на событие
 - `<Grid Click="Grid_Click">` вызовет ошибку.
- В XAML:
 - Атрибут – событие. Атрибут – Класс.Название_События, значение имя метода обработчика события
 - `<Grid ButtonBase.Click="Grid_Click">`
 - `<Button>Hi</Button>`
 - `</Grid>`
- В C#:
 - `grid1.AddHandler(Button.ClickEvent, new RoutedEventHandler(button1_Click));`

Демонстрация

События

События

- Типы событий:
 - **Прямые**. Иницируются только в том элементе, в котором произошло
 - Пример: `MouseLeave`
 - **Пузырьковые**. Сначала иницируются элементе управления, в котором произошло, а затем в каждом предшественнике в визуальном дереве
 - Пример: `MouseDown`, `MouseUp`
 - **Туннельные**. Иницируются сначала в контейнере высшего уровня в визуальном дереве, а затем спускается по всем элементам к текущему.
 - Пример: `PreviewMouseDown`, `PreviewMouseUp`
- В .NET все туннельные события начинаются с **Preview...**
- Как правило, туннельные события определяются в парах с пузырьковыми событиями. Туннельные вызываются перед пузырьковыми.
- Туннельные события и соответствующие им пузырьковые события используют один и тот же экземпляр аргументов события `RoutedEventArgs`. Обозначение туннельного события как обработанного прерывает вызов и соответствующего пузырькового события

События WPF

- События времени существования
 - `Initialized`, `Loaded`, `Unloaded`, `Activated(Window)`, `Deactivated(Window)`, `Closing(Window)`, `Closed(Window)`,
- События мыши
 - `MouseLeftButtonDown`, `MouseRightButtonDown`, `MouseLeftButtonUp`, `MouseRightButtonUp`, `MouseDown`, `MouseUp` и соответствующие `Preview...`, `MouseEnter`, `MouseLeave`, `MouseMove`, `MouseWheel`, `MouseDoubleClick`
- События клавиатуры
 - `KeyDown`, `TextInput`, `KeyUp` и соответствующие `Preview...`
- События пера
- События одновременного касания (**MultiTouch**). Начиная с **Windows 7**
 - `TouchDown`, `TouchUp`, `TouchMove` и соответствующие `Preview...`, `TouchEnter`, `TouchLeave`
 - Поддержка манипуляций (жестов)
 - `IsManipulationEnabled="True"` – включает поддержку жестов элементом управления
 - `ManipulationStarting`, `ManipulationStarted`, `ManipulationDelta`, `ManipulationCompleted`

Создание маршрутизируемого события

- Определение события

```
public static readonly RoutedEvent MyClickEvent;
```

- Регистрация события

- В статическом конструкторе

```
static MyClass()  
{  
    MyClickEvent = EventManager.RegisterRoutedEvent(  
        "MyClick",  
        RoutingStrategy.Bubble,  
        typeof(RoutedEventHandler),  
        typeof(MyClass));  
}
```

- Традиционная оболочка события

```
public event RoutedEventHandler MyClick  
{  
    add { AddHandler(MyClickEvent, value); }  
    remove { RemoveHandler(MyClickEvent, value); }  
}
```

- Генерация события

```
RoutedEventArgs e = new RoutedEventArgs(MyClickEvent);  
RaiseEvent(e);
```

Класс `EventManager`

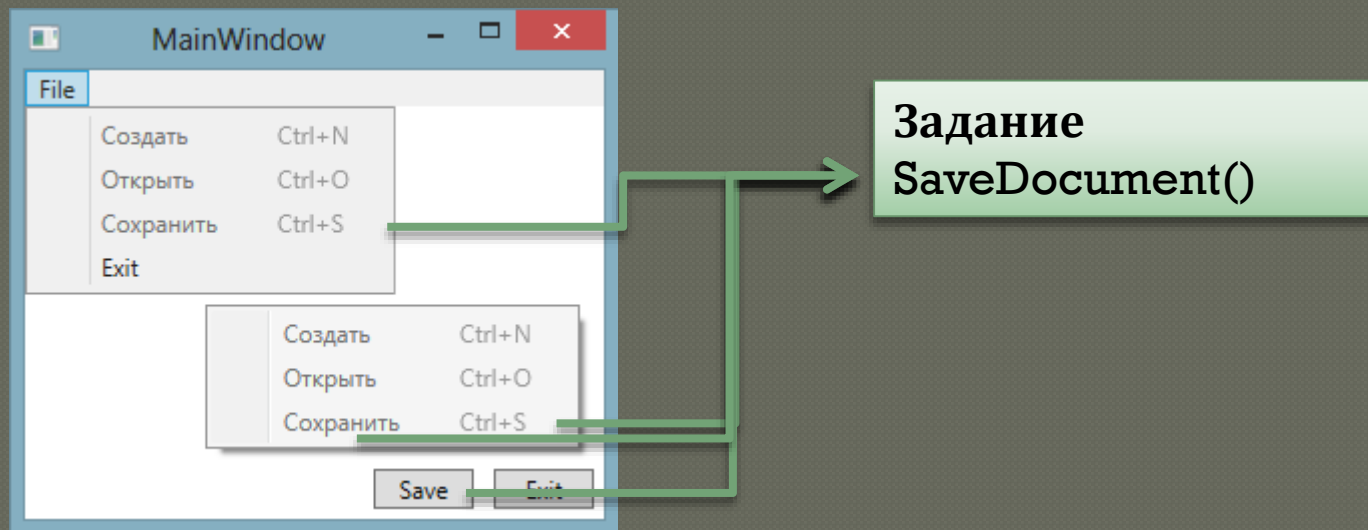
- `EventManager` – Статический класс, управляющий регистрацией всех событий в WPF
 - `GetRoutedEvents` – возвращает все маршрутизированные события приложения
 - `GetRoutedEventsForOwner` – возвращает все маршрутизированные события для указанного элемента приложения
 - `RegisterRoutedEvent` – регистрирует новое событие

Сегодня

- События
- Команды

Команда

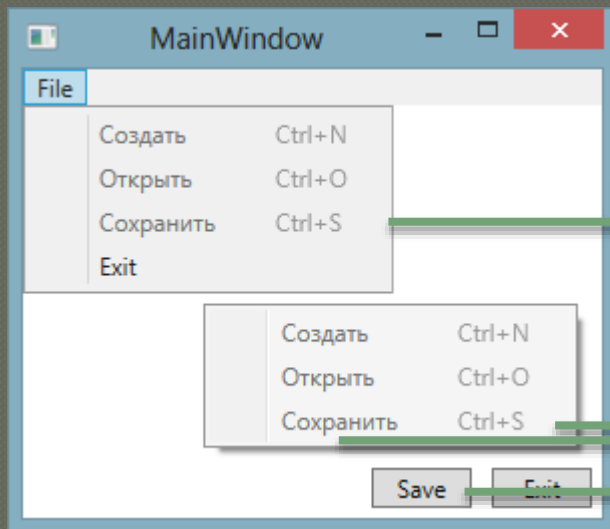
- Одни и те же задания(tasks) могут инициироваться пользователем с помощью разных элементов пользовательского интерфейса - кнопок, меню, контекстных меню, сочетание клавиш, двойных щелчков мыши и т.д.
- При этом цель – выполнить одну и ту же задачу



- синхронизации состояния команды и элементов управления (отключение при невозможности выполнить задание)

Команда

- Команда представляет прикладную задачу и следит за тем когда она может быть выполнена.
- Команда не содержит код, который должен быть выполнен



Команда
SaveCommand

Задание
SaveDocument()

Интерфейс ICommand

```
public interface ICommand
{
    void Execute( Object parameter );
    bool CanExecute( Object parameter );
    event EventHandler CanExecuteChanged;
}
```

- Интерфейс ICommand реализуется классом RoutedCommand и наследуется классом RoutedUICommand. Это единственные классы WPF, реализующие интерфейс ICommand.
- **CanExecute()** - возвращает значение true, если команду можно выполнить для целевого объекта (command target).
- **Execute()** выполнение действия, ассоциированных с командой.
- Через параметры методов CanExecute() и Execute() можно передать ссылку на данные (допускается значение null).
- Событие CanExecuteChanged происходит, когда менеджер команд диагностирует изменение в источнике команды, которое может привести к невозможности выполнения уже инициированной(raised) команды, но еще не выполненной. Обычно в ответ на это событие источник команды вызывает метод CanExecute().

Типичная реализация ICommand

```
public class RelayCommand : ICommand
{
    private readonly Action<object> execute;
    private readonly Predicate<object> canExecute;

    public RelayCommand(Action<object> execute): this(execute, _ => true)    {    }
    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        if (execute == null) throw new ArgumentNullException("execute");
        this.execute = execute;
        this.canExecute = canExecute;
    }

    public bool CanExecute(object parameter)
    {
        return canExecute == null ? true : canExecute(parameter);
    }

    public void Execute(object parameter)    {    execute(parameter);    }

    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }
}
```


Использование команд в MVVM

- В ViewModel

```
public MainViewModel() //ctor
{
    ...
    SaveCommand = new RelayCommand(o => Save(), o => CanSave());
    LoadCommand = new RelayCommand(o => Load());
}
```

```
public ICommand LoadCommand { get; private set; }
public ICommand SaveCommand { get; private set; }
```

```
private void Load() { ... }
```

```
private bool CanSave()
{
    return Employees != null && Employees.Count > 0;
}
```

```
private void Save() { ..... }
```

- В View (XAML)

```
<Button Content="Загрузить" Command="{Binding Path=LoadCommand}" />
<Button Content="Сохранить" Command="{Binding Path=SaveCommand}" />
```

Демонстрация

Команды

Модель команд в WPF

- Модель маршрутизируемых команд WPF включает следующие основные компоненты:
 - Команда (**Command**) – действие, которое должно быть выполнено.
 - Источник команды – объект, вызывающий команду.
 - Целевые объекты команд – объекты, на которых должна быть выполнена команда (например **Paste**).
 - Привязка команд (**CommandBinding**) – настройка команд
- Существующие команды в WPF представляет собой экземпляр класса **RoutedUICommand** и не реализует логику команды. Логика команды присоединяется к команде с помощью объекта **CommandBinding**.

Предопределенные команды

- Библиотека WPF содержит более 100 предопределенных команд, которые находятся в 5 статических классах:
 - ApplicationCommands
 - NavigationCommands
 - MediaCommands
 - EditingCommands
 - ComponentCommands.
- Все статические свойства классов имеют сигнатуру
 - `public static RoutedUICommand CommandName { get; }`
- Статический класс ApplicationCommands содержит статические свойства:
 - New, Open, Save, SaveAs, Close,
 - Print, PrintPreview, CancelPrint,
 - Copy, Cut, Past, Delete, Undo, Redo, Find, Replace, SelectAll,
 - ContextMenu, Help, Properties, CorrectionList

Выполнение команд

- Задание команды

```
<Button Command="ApplicationCommands.Open">New</Button>
```

- Поскольку имена имеющихся команд не пересекаются, то можно задавать и так

```
<Button Command="Open">New</Button>
```

- Привязки команд:

- В XAML:

```
<Window.CommandBindings>
```

```
<CommandBinding Command="ApplicationCommands.Open"
```

```
    Executed="OpenCommandBinding_Executed"
```

```
    CanExecute="OpenCommandBinding_CanExecute"/>
```

```
</Window.CommandBindings>
```

- В коде:

```
CommandBinding binding = new
```

```
    CommandBinding(ApplicationCommands.New,
```

```
        New_Executed, New_CanExecute);
```

```
CommandBindings.Add(binding);
```