**1.WindowsForms. Модель обработки событий. Взаимодействие между формами.**

Для создания графических интерфейсов с помощью платформы .NET применяются разные технологии - Window Forms, WPF). Однако наиболее простой и удобной платформой до сих пор остается Window Forms или формы.

Для взаимодействия с пользователем в Windows Forms используется механизм событий. События в Windows Forms представляют стандартные события на C#, только применяемые к визуальным компонентам и подчиняются тем же правилам, что события в C#.

Для добавления обработчика используется стандартный синтаксис C#: this.Load += new System.EventHandler(this.Form1\_Load)

Допустим, первая форма по нажатию на кнопку будет вызывать вторую форму. Во-первых, добавим на первую форму Form1 кнопку и двойным щелчком по кнопке перейдем в файл кода. Итак, мы попадем в обработчик события нажатия кнопки, который создается по умолчанию после двойного щелчка по кнопке.

private void button1\_Click(object sender, EventArgs e)

{

Form2 newForm = new Form2();

newForm.Show();

}

Пока вторая форма не знает о существовании первой. Чтобы это исправить, надо второй форме как-то передать сведения о первой форме. Для этого воспользуемся передачей ссылки на форму в конструкторе.

public Form2(Form1 f)

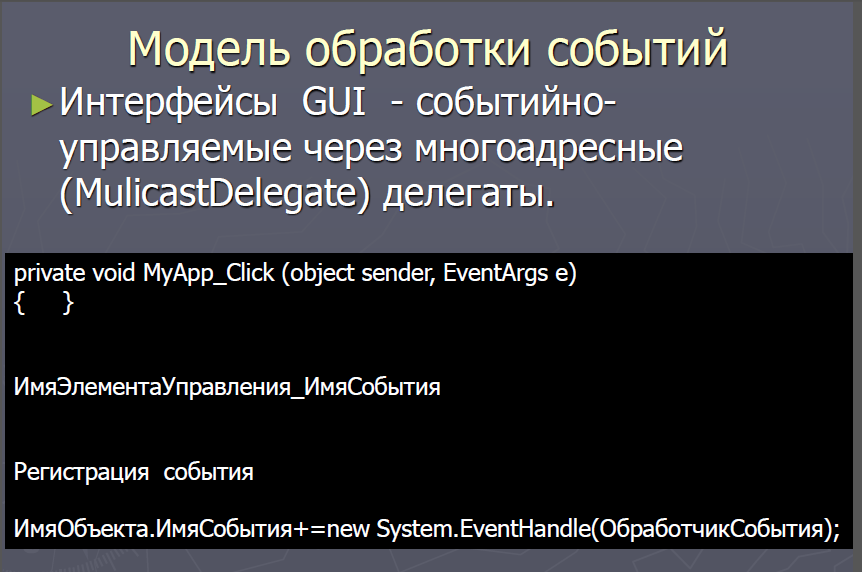
{

InitializeComponent();

f.BackColor = Color.Yellow;

}

Форма – это объект, который задается свойствами, определяющими их внешний вид, методами, определяющими их поведение, и событиями, определяющими их взаимодействие с пользователем.



В Windows Forms модель обработки событий основана на применении делегатов и событий. Когда происходит определенное событие, такое как щелчок мыши или нажатие клавиши, форма генерирует событие, и код может быть написан для обработки этого события.

Вот основные шаги в модели обработки событий в Windows Forms:

1) Определение событий: Форма может определить различные события, которые могут произойти, например, событие клика на кнопке или изменение текста в текстовом поле. Для каждого события определяется соответствующий делегат, который указывает на метод, который будет обрабатывать событие.

2) Подписка на события: Код должен подписаться на события, которые он хочет обрабатывать. Это делается путем присоединения методов-обработчиков событий к соответствующим событиям формы. Обработчик события - это метод, который будет вызываться, когда происходит событие.

3) Обработка событий: Когда событие происходит, вызывается соответствующий метод-обработчик события. В этом методе можно выполнять необходимые действия в ответ на событие. Например, при клике на кнопку можно выполнять определенный код для обработки этого события.

4) Отписка от событий: При необходимости можно отписаться от событий, чтобы прекратить обработку определенного события. Это делается путем удаления метода-обработчика из списка подписчиков события.

Взаимодействие между формами в Windows Forms может быть реализовано с использованием различных подходов и методов. Вот несколько способов взаимодействия между формами:

1) Передача данных через конструктор: Один из простых способов передачи данных между формами - это передача значений через конструктор второй формы при её создании. Это позволяет передать данные из одной формы в другую при их инициализации.

2) Использование свойств и методов: Формы могут содержать публичные свойства и методы, которые могут быть доступны другим формам для получения или изменения данных. Например, вы можете определить свойство на одной форме для получения значения текстового поля на другой форме.

3) События и делегаты: Формы могут взаимодействовать друг с другом через события и делегаты. Например, вы можете определить событие на одной форме и подписаться на него с другой формы для получения уведомлений о событиях, происходящих на первой форме.

4) Использование статических членов: Вы можете определить статические свойства или методы, которые будут доступны из любой формы в приложении. Это позволяет обмениваться данными и вызывать методы без явной ссылки на экземпляр формы.

5) Использование глобального хранилища данных: Вы можете использовать глобальные переменные, файлы конфигурации, базы данных или другие формы хранилища данных для обмена информацией между формами. В этом случае каждая форма будет обращаться к общим данным для получения или обновления информации.

**2.Элементы управления и их объектная модель.**

Элемент управления играет роль графической связи между пользователем и программой. Элемент управления может предоставлять или обрабатывать данные, принимать данные, введенные пользователем, реагировать на события, а также выполнять любые другие функции, связывающие пользователя и приложение.

Все элементы управления наследуются от класса Control и поэтому имеют ряд общих свойств:

**Anchor:** Определяет, как элемент будет растягиваться

**BackColor:** Определяет фоновый цвет элемента

**BackgroundImage:** Определяет фоновое изображение элемента

**ContextMenu:** Контекстное меню, которое открывается при нажатии на элемент правой кнопкой мыши. Задается с помощью элемента ContextMenu

**Cursor:** Представляет, как будет отображаться курсор мыши при наведении на элемент

**Dock:** Задает расположение элемента на форме

**Enabled:** Определяет, будет ли доступен элемент для использования. Если это свойство имеет значение False, то элемент блокируется.

**Font:** Устанавливает шрифт текста для элемента

**ForeColor:** Определяет цвет шрифта

**Location:** Определяет координаты верхнего левого угла элемента управления

**Name**: Имя элемента управления

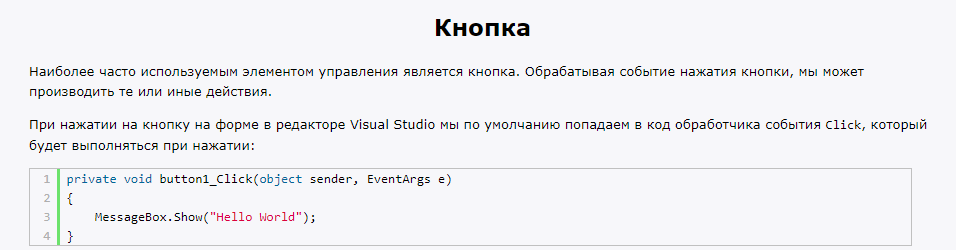
**Size:** Определяет размер элемента

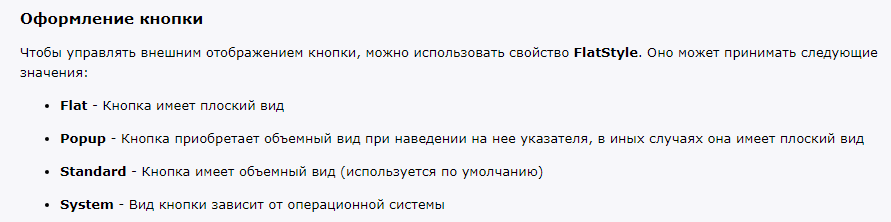
**Width:** ширина элемента

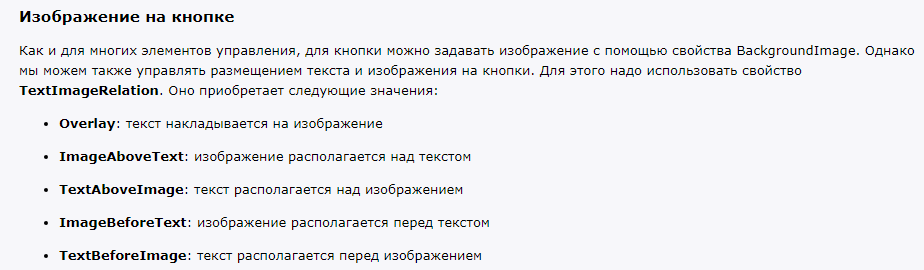
**Height:** высота элемента

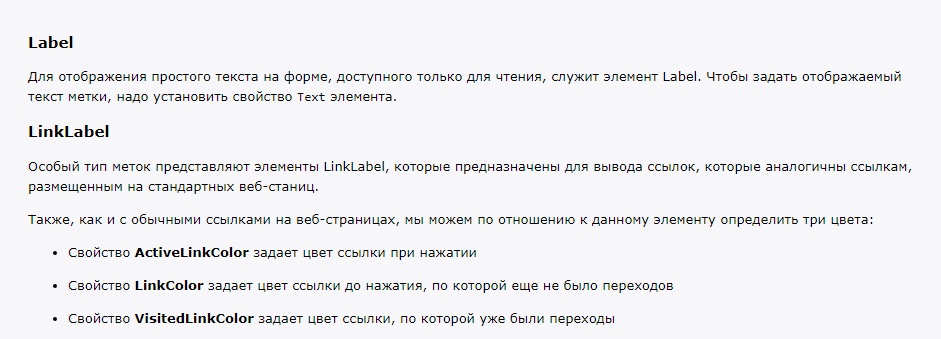
**TabIndex:** Определяет порядок обхода элемента по нажатию на клавишу Tab

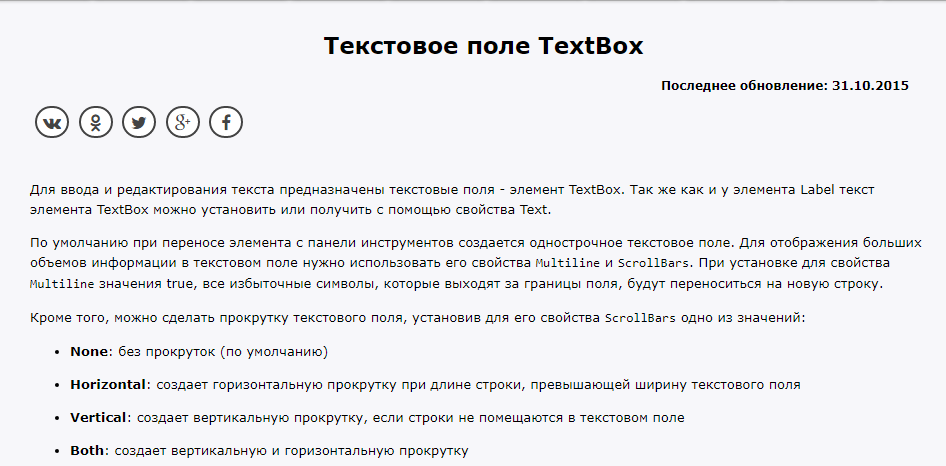
**Tag:** Позволяет сохранять значение, ассоциированное с этим элементом управления

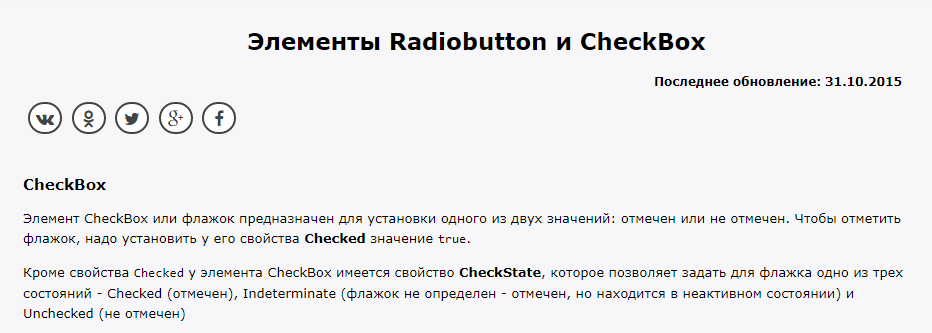


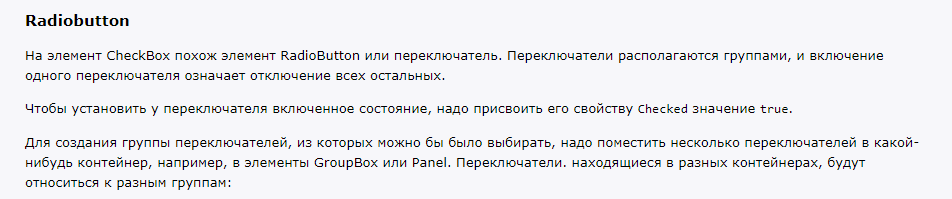


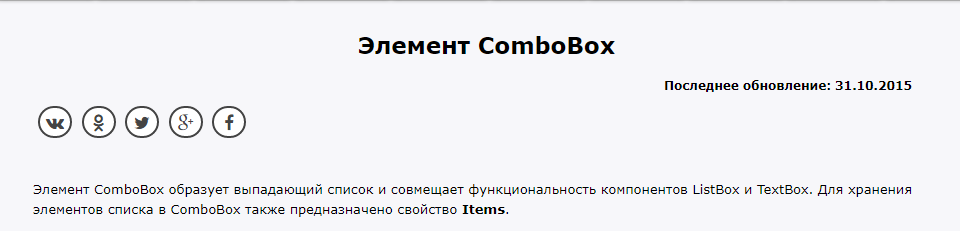
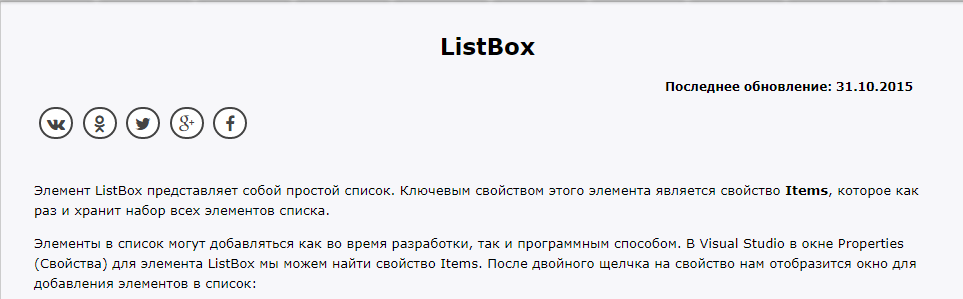


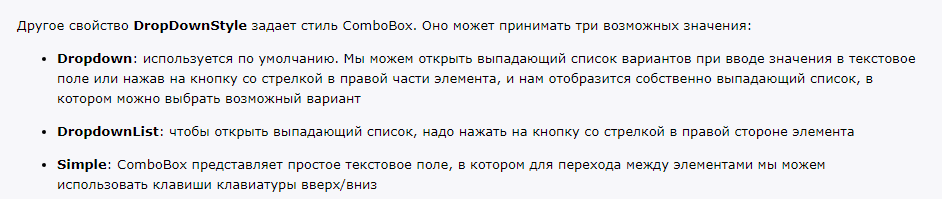




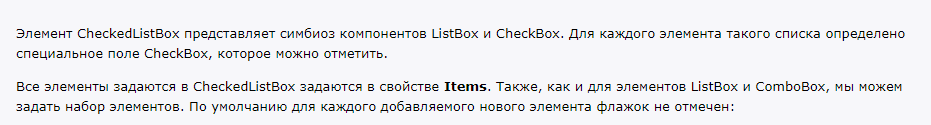


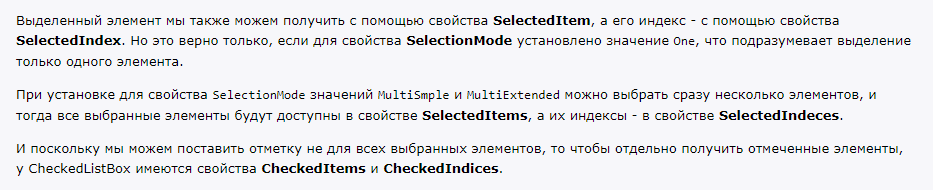


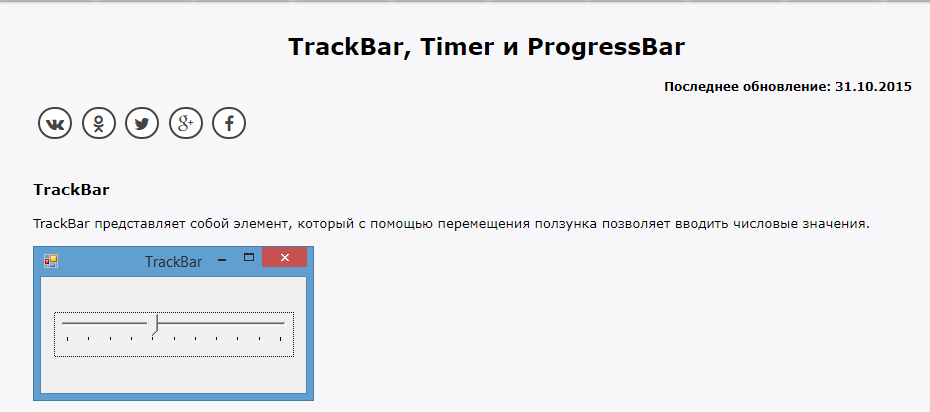


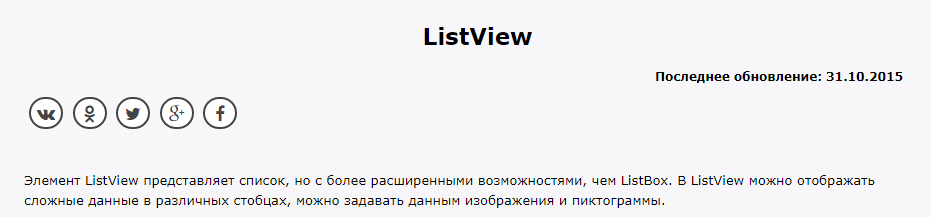


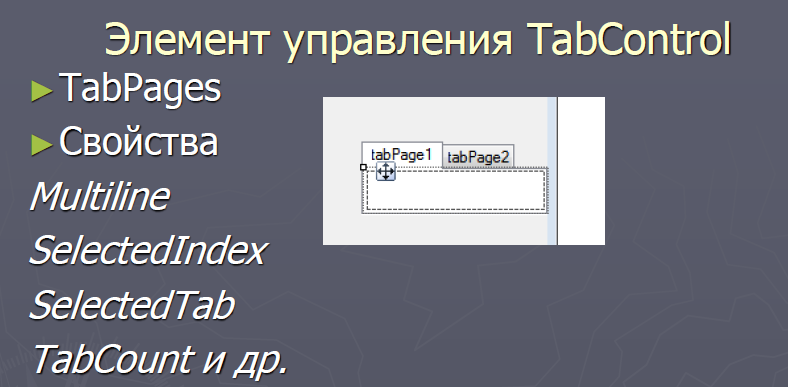












**3.Регулярные выражения. Поиск и чтение результатов.**

Регулярные выражения (regular expressions) - это мощный инструмент для работы с текстом, который позволяет осуществлять поиск и сопоставление шаблонов в строках. Они широко используются для выполнения различных операций с текстом, таких как поиск, замена, извлечение данных и проверка формата.

В .NET Framework, для работы с регулярными выражениями, используется класс `Regex` из пространства имен System.Text.RegularExpressions. Вот некоторые основные методы, которые можно использовать для работы с регулярными выражениями:

1. `Regex.IsMatch`: Метод позволяет проверить, соответствует ли строка заданному регулярному выражению. Он возвращает `true`, если соответствие найдено, и `false` в противном случае.

2. `Regex.Match`: Метод выполняет поиск первого соответствия регулярному выражению в строке. Он возвращает объект `Match`, который содержит информацию о найденном совпадении.

3. `Regex.Matches`: Метод выполняет поиск всех совпадений регулярного выражения в строке. Он возвращает коллекцию объектов `Match`, каждый из которых содержит информацию о найденном совпадении.

4. `Regex.Replace`: Метод позволяет заменить все совпадения регулярного выражения в строке на заданную подстановку.

Регулярные выражения обладают множеством возможностей

### **Параметр RegexOptions**

Класс Regex имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление RegexOptions. Некоторые из значений, принимаемых данным перечислением:

* **Compiled**: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
* **CultureInvariant**: при установке этого значения будут игнорироваться региональные различия
* **IgnoreCase**: при установке этого значения будет игнорироваться регистр
* **IgnorePatternWhitespace**: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
* **Multiline**: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
* **RightToLeft**: приписывает читать строку справа налево
* **Singleline**: устанавливает однострочный режим, а весь текст рассматривается как одна строка

**4.Регулярные выражения. Классы символов. Привязки и кванторы.**

В регулярных выражениях классы символов и кванторы используются для создания более гибких и точных шаблонов поиска. Они позволяют указать набор символов, которые могут соответствовать определенной позиции в тексте, а также определить количество повторений символов.

1. Классы символов:

- `[abc]`: Соответствует одному символу, который может быть либо `a`, либо `b`, либо `c`.

- `[0-9]`: Соответствует любому цифровому символу.

- `[a-zA-Z]`: Соответствует любой букве в верхнем или нижнем регистре.

- `[^abc]`: Соответствует одному символу, который НЕ является `a`, `b` или `c`.

2. Привязки:

- `^`: Соответствует началу строки. Например, `^Hello` найдет слово "Hello", только если оно находится в начале строки.

- `$`: Соответствует концу строки. Например, `world$` найдет слово "world", только если оно находится в конце строки.

3. Кванторы:

- `\*`: Соответствует 0 или более повторений предыдущего символа или группы. Например, `a\*` найдет ноль или более символов "a".

- `+`: Соответствует 1 или более повторений предыдущего символа или группы. Например, `a+` найдет один или более символов "a".

- `?`: Соответствует 0 или 1 повторению предыдущего символа или группы. Например, `colou?r` найдет слова "color" или "colour".

- `{n}`: Соответствует ровно n повторениям предыдущего символа или группы. Например, `a{3}` найдет три символа "a".

- `{n,}`: Соответствует n или более повторениям предыдущего символа или группы. Например, `a{2,}` найдет два или более символов "a".

- `{n,m}`: Соответствует от n до m повторений предыдущего символа или группы. Например, `a{2,4}` найдет от двух до четырех символов "a".

Примеры:

- `[aeiou]`: Соответствует любому гласному символу.

- `^[A-Z]\w\*$`: Соответствует строке, начинающейся с большой буквы и содержащей только буквы и цифры.

- `\d

**5.Атрибуты валидации модели (диапазоны, типы, сообщения об ошибках, регулярные выражения,**

### Атрибут Required

Применение этого атрибута к свойству модели означает, что данное свойство должно быть обязательно установлено.

[Required (ErrorMessage="Поле должно быть установлено")]

### Атрибут StringLength

Чтобы пользователь не мог ввести очень длинный текст, используется атрибут StringLength. Особенно это актуально, если в базе данных установлено ограничение на размер хранящихся строк.

[StringLength (50, MinimumLength=3,ErrorMessage="Длина строки должна быть от 3 до 50 символов")]

### Атрибут RegularExpression

Применение данного атрибута предполагает, что вводимое значение должно соответствовать указанному в этом атрибуте регулярному выражению.

[RegularExpression(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}", ErrorMessage = "Некорректный адрес")]

### Атрибут Range

Атрибут Range определяет минимальные и максимальные ограничения для числовых данных.

[Range(1700,2000,ErrorMessage="Недопустимый год")]

Атрибуты валидации модели в программировании (например, в контексте ASP.NET MVC или Entity Framework) позволяют определить правила и ограничения для свойств модели. Они обеспечивают проверку данных, вводимых пользователем или получаемых из других источников, и предоставляют сообщения об ошибках в случае нарушения этих правил.

Ниже приведены некоторые распространенные атрибуты валидации модели и их функции:

1. **Required[Required]**: Указывает, что свойство должно быть заполнено (не может быть пустым или null).
2. **Range(minValue, maxValue)[Range(minValue, maxValue)]**: Определяет диапазон допустимых значений для числовых свойств.
3. **[StringLength(maxLength)]StringLength(maxLength)**: Ограничивает максимальную длину строки для свойств типа **string**.
4. **RegularExpression(pattern)[RegularExpression(pattern)]**: Устанавливает шаблон регулярного выражения, которому должно соответствовать значение свойства.
5. **EmailAddress[EmailAddress]**: Проверяет, что значение свойства является допустимым адресом электронной почты.
6. **DataType(DataType.Date)[DataType(DataType.Date)]**: Указывает, что свойство должно быть датой.
7. **[Compare(otherProperty)]Compare(otherProperty)**: Сравнивает значение свойства с другим свойством модели.

public class Person

{

[Required(ErrorMessage = "Имя является обязательным полем.")]

public string Name { get; set; }

[Range(18, 99, ErrorMessage = "Возраст должен быть от 18 до 99.")]

public int Age { get; set; }

[EmailAddress(ErrorMessage = "Недопустимый адрес электронной почты.")]

public string Email { get; set; }

}

**6.Атрибуты валидации. Создание атрибута валидации (пример).**

Для создания атрибута нам надо унаследовать свой класс от класса **ValidationAttribute** и реализовать его метод IsValid().

public class UserNameAttribute : ValidationAttribute

{

public override bool IsValid(object value)

{

if (value != null)

{

string userName = value.ToString();

if (!userName.StartsWith("T"))

return true;

else

this.ErrorMessage = "Имя не должно начинаться с буквы T";

}

return false;

}

}

**7.WPF -Windows Presentation Foundation. Архитектура. Объектная модель.Отличие от WinForms.**

Технология WPF (Windows Presentation Foundation) является часть экосистемы платформы .NET и представляет собой подсистему для построения графических интерфейсов.

Схематически архитектуру WPF можно представить следующим образом:

Как видно на схеме, WPF разбивается на два уровня: managed API и unmanaged API (уровень интеграции с DirectX). Managed API (управляемый API-интерфейс) содержит код, исполняемый под управлением общеязыковой среды выполнения .NET - Common Language Runtime. Этот API описывает основной функционал платформы WPF

Архитектура WPF: WPF имеет следующую архитектуру:

Уровень представления (Presentation Layer): Это верхний уровень WPF, отвечающий за визуальное представление элементов пользовательского интерфейса. Он определяет, как элементы отображаются, их стиль, анимации, макет и другие свойства. Разметка XAML используется для определения интерфейса.

Уровень логики (Logic Layer): Этот уровень содержит бизнес-логику приложения. Здесь вы можете определить логику обработки событий, обработку данных и взаимодействие с другими частями системы. Язык программирования, такой как C#, используется для написания кода логики.

Уровень данных (Data Layer): Этот уровень отвечает за предоставление данных, необходимых для приложения. Он может включать в себя базы данных, сервисы веб-служб, модели данных и другие источники данных. WPF предоставляет связывание данных (data binding), которое позволяет легко связывать элементы пользовательского интерфейса с данными.



Ключевые возможности

Веб

-подобная модель компоновки

-Богатая модель рисования

-Развитая текстовая модель

-Поддержка аудио и видео

-Приложения на основе страниц

-Декларативный пользовательский интерфейс(XAML) (сочетание)

-Стили и шаблоны. Команды

**8.WPF. Контейнеры компоновки**

**Canvas**

Элемент управления Canvas поддерживает абсолютное позиционирование и предоставляет минимум встроенных возможностей макета для содержащихся в нем элементов управления. Элемент управления Canvas позволяет размещать содержащиеся в нем элементы управления со сдвигом относительно любого угла панели.

<Canvas Background="LightBlue">

<Label Canvas.Top="30" Canvas.Left="20">Enter here:</Label>

<TextBox Canvas.Top="30" Canvas.Left="120" Width="100" />

<Button Canvas.Top="70" Canvas.Left="130" Content="Click Me!" Padding="5" />

</Canvas>

DockPanel

Элемент управления DockPanel обеспечивает поддержку закрепления для упрощения расположения панелей инструментов и других элементов управления, которые требуется закрепить вдоль одной из границ панели. Элемент управления DockPanel предоставляет свойство с именем DockStyle для содержащихся в нем элементов управления, которое определяет, как они будут расположены.

<DockPanel>

<Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">

<TextBlock>Menu</TextBlock>

</Border>

<Border Height="25" Background="Aqua" DockPanel.Dock="Top">

<TextBlock>Toolbar</TextBlock>

</Border>

<Border Height="30" Background="LightSteelBlue" DockPanel.Dock="Bottom">

<TextBlock>Status</TextBlock>

</Border>

<Border Height="80" Background="Azure" DockPanel.Dock="Left">

<TextBlock>Left Side</TextBlock>

</Border>

<Border Background="HotPink">

<TextBlock>Remaining Part</TextBlock>

</Border>

</DockPanel>

**Grid**

Наиболее знакомым контейнерным элементом управления является элемент управления Grid. По умолчанию каждое новое окно Window, открытое в Windows Presentation Foundation (WPF) для Visual Studio (конструктор), включает элемент управления Grid. Элемент управления Grid позволяет позиционировать элементы управления внутри ячеек, задаваемых пользователем. Элементы управления, помещенные в ячейки, поддерживают фиксированное поле между двумя или более краями элемента управления и краями ячеек при изменении размера окна Window.

<Grid ShowGridLines="True">

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition />

<RowDefinition />

</Grid.RowDefinitions>

<Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" VerticalAlignment="Center" HorizontalAlignment="Center" Content="Title" />

<Label Grid.Column="0" Grid.Row="1" VerticalAlignment="Center" Content="Firstname:" Margin="10" />

<TextBox Grid.Column="1" Grid.Row="1" Width="100" Height="30" />

<Label Grid.Column="0" Grid.Row="2" VerticalAlignment="Center" Content="Lastname:" Margin="10" />

<TextBox Grid.Column="1" Grid.Row="2" Width="100" Height="30" />

</Grid>

**StackPanel**

Элемент управления StackPanel располагает содержащиеся в нем элементы управления либо в вертикальном столбце, либо в горизонтальной строке, в зависимости от значения свойства Orientation.

<StackPanel Orientation="Vertical">

<Label>Label</Label>

<TextBox>TextBox</TextBox>

<CheckBox>CheckBox</CheckBox>

<CheckBox>CheckBox</CheckBox>

<ListBox>

<ListBoxItem>ListBoxItem One</ListBoxItem>

<ListBoxItem>ListBoxItem Two</ListBoxItem>

</ListBox>

<Button>Button</Button>

</StackPanel>

**WrapPanel**

Элемент управления WrapPanel аналогичен элементу управления StackPanel в том, что в нем элементы управления располагаются в столбце или в строке в зависимости от свойства Orientation. В дополнение к подобному расположению, элемент управления WrapPanel поддерживает возможность переноса в следующий столбец или строку для содержащихся в нем элементов управления.

<WrapPanel>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

</WrapPanel>

**9.XAML. Декларативная разработка. Простые и сложные свойства.**

XAML (eXtensible Application Markup Language) - язык разметки, используемый для инициализации объектов в технологиях на платформе .NET. Применительно к WPF (а также к Silverlight) данный язык используется прежде всего для создания пользовательского интерфейса декларативным путем.

С помощью атрибутов мы можем задать различные свойства кнопки. Height и Width являются простыми свойствами. Они хранят числовое значение. А например, свойства HorizontalAlignment или Background являются более сложными по своей структуре.

Чтобы выровнять кнопку по центру, применяется перечисление HorizontalAlignment, а для установки фонового цвета - класс SolidColorBrush. Хотя в коде XAML мы ничего такого не увидели и устанавливали эти свойства гораздо проще с помощью строк: Background="Red". Дело в том, что по отношению к коду XAML применяются специальные объекты - **type converter** или конвертеры типов, которые могут преобразовать значения из XAML к тем типам тех объектов, которые используются в коде C#.

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<TextBlock Grid.Row="0" Text="Заголовок" />

<Button Grid.Row="1" Content="Нажми меня" />

</Grid>

**10.WPF. Понятие свойств зависимостей. Правила определения. Aattached properties**

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.

public class Phone : DependencyObject

{

public static readonly DependencyProperty TitleProperty;

static Phone()

{

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

}

public string Title

{

get { return (string)GetValue(TitleProperty); }

set { SetValue(TitleProperty, value); }

}

}

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.

<Button x:Name="button1" Content="Hello" Grid.Column="1" Grid.Row="0" />

**11.WPF.Обработка событий. Виды маршрутизации событий.**

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.

**12.WPF.Панели.Ресурсы статические и динамические**

**логических ресурсах**, которые могут представлять различные объекты - элементы управления, кисти, коллекции объектов и т.д. Логические ресурсы можно установить в коде XAML или в коде C# с помощью свойства Resources.

В чем смысл использования ресурсов? Они повышают эффективность: мы можем определить один раз какой-либо ресурс и затем многократно использовать его в различных местах приложения. В связи с этим улучшается поддержка - если возникнет необходимость изменить ресурс, достаточно это сделать в одном месте, и изменения произойдут глобально в приложении.

Свойство **Resources** представляет объект **ResourceDictionary** или словарь ресурсов, где каждый хранящийся ресурс имеет определенный ключ.

Причем каждый ресурс обязательно имеет свойство **x:Key**, которое и определяе ключ в словаре.

**13.WPF.Команды (Command). Модель команд. Встроенные команды**

**Команды** представляют механизм выполнения какой-нибудь задачи, например, копирования текста - когда мы нажимаем Ctrl+C, то мы копируем текст в буффер. В процессе копирования выполняется ряд действий, и все вместе эти действия объединяются в одну команду.

Все команды реализуют интерфейс System.Window.Input.ICommand:

public interface ICommand

{

event EventHandler CanExecuteChanged;

void Execute (object parameter);

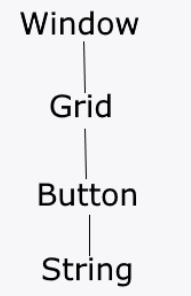
bool CanExecute (object parameter);

}

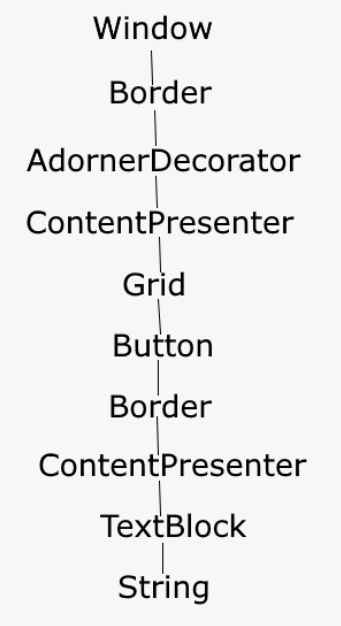
Некоторые элементы управления вводом умеют обрабатывать события команд самостоятельно. Например, класс TextBox обрабатывает команды Cut, Сору и Paste (а также команды Undo и Redo и часть команд класса EditingCommands, которые позволяют выделять текст и перемещать курсор в разные позиции)

**14.WPF.Логические и визуальные деревья. Шаблоны.**

Визуализация в WPF тесно связана с такими понятиями как логическое и визуальное дерево. Эти деревья являются своего рода каркасом приложения. Так мы можем представить приложение как некий набор вложенных элементов.



То есть в Window есть Grid, в Gride - элемент Button, в кнопке в качестве содержимого установлен некоторый текст в виде объекта String. В итоге получается некое дерево элементов, которое называется **логическим**. В WPF оно представлено классом **System.Windows.LogicalTreeHelper**. Логическое дерево имеет дело с визуализацией как таковой, оно образует модель доступа к дочерним элементам.



От него отличается **визуальное дерево**, представленное классом **System.Windows.Media.VisualTreeHelper**. Так, визуальное дерево для вышеприведенной разметки xaml будет выглядеть следующим образом. Визуальное дерево получается гораздо сложнее, оно показывает, как с визуальной точки зрения устроен элемент, из каких частей он состоит.

Визуальное дерево элемента управления опредлеляет, как будет выглядеть этот элемент или иными словами его **шаблон**. Шаблон элемента - это своего рода визуальный скелет элемента управления. Например, для элемента Button упрощенно шаблон выглядит следующим образом:

**15.WPF.Стили.**

Стили позволяют определить набор некоторых свойств и их значений, которые потом могут применяться к элементам в xaml. Стили хранятся в ресурсах и отделяют значения свойств элементов от пользовательского интерфейса. Также стили могут задавать некоторые аспекты поведения элементов с помощью триггеров. Аналогом стилей могут служить каскадные таблицы стилей (CSS), которые применяются в коде html на веб-страницах.

<Window.Resources>

<Style x:Key="BlackAndWhite">

<Setter Property="Control.FontFamily" Value="Verdana" />

<Setter Property="Control.Background" Value="Black" />

<Setter Property="Control.Foreground" Value="White" />

<Setter Property="Control.Margin" Value="10" />

</Style>

</Window.Resources>

Результат будет тот же, однако теперь мы избегаем не нужного повторения. Более того теперь мы можем управлять всеми нужными нам свойствами как единым целым - одним стилем.

Стиль создается как ресурс с помощью объекта **Style**, который представляет класс **System.Windows.Style**. И как любой другой ресурс, он обязательно должен иметь ключ. С помощью коллекции **Setters** определяется группа свойств, входящих в стиль. В нее входят объекты **Setter**, которые имеют следующие свойства:

**16.WPF.Триггеры. Типы триггеров**

Тригерры позволяют декларативно задать некоторые действия, которые выполняются при изменении свойств стиля. Существует три вида триггеров:

* **Триггеры свойств**: вызываются в ответ на изменения свойствами зависимостей своего значения

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="FontSize" Value="14" />

<Setter Property="Foreground" Value="Red" />

</Trigger>

* **Триггеры данных**: вызываются в ответ на изменения значений любых свойств (они необязательно должны быть свойствами зависимостей)

<DataTrigger Binding="{Binding ElementName=checkBox1, Path=IsChecked}"

Value="True">

<Setter Property="IsEnabled" Value="False"/>

</DataTrigger>

* **Триггеры событий**: вызываются в ответ на генерацию событий

<EventTrigger RoutedEvent="Click">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimation Storyboard.TargetProperty="Width" Duration="0:0:1" To="220" AutoReverse="True" />

<DoubleAnimation Storyboard.TargetProperty="Height" Duration="0:0:1" To="80" AutoReverse="True" />

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

* **Мультитриггеры**: вызываются при выполнении ряда условий

<MultiTrigger>

<MultiTrigger.Conditions>

<Condition Property="IsMouseOver" Value="True" />

<Condition Property="IsPressed" Value="True" />

</MultiTrigger.Conditions>

<MultiTrigger.Setters>

<Setter Property="FontSize" Value="14" />

<Setter Property="Foreground" Value="Red" />

</MultiTrigger.Setters>

</MultiTrigger>

**17.WPF. Привязки (binding). Направление и обновление привязок. Интерфейс INotifyPropertyChanged**

Привязка подразумевает взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника. В случае модификации объекта-источника, объект-приемник также будет модифицирован.

### **Режимы привязки**

Свойство **Mode** объекта Binding, которое представляет режим привязки, может принимать следующие значения:

* **OneWay**: свойство объекта-приемника изменяется после модификации свойства объекта-источника.
* **OneTime**: свойство объекта-приемника устанавливается по свойству объекта-источника только один раз. В дальнейшем изменения в источнике никак не влияют на объект-приемник.
* **TwoWay**: оба объекта - применки и источник могут изменять привязанные свойства друг друга.
* **OneWayToSource: объект-приемник, в котором объявлена привязка, меняет объект-источник.**
* **Default**: по умолчанию (если меняется свойство TextBox.Text, то имеет значение TwoWay, в остальных случаях OneWay).

Чтобы объект мог полноценно реализовать механизм привязки, нам надо реализовать в его классе интерфейс INotifyPropertyChanged. Когда объект класса изменяет значение свойства, то он через событие PropertyChanged извещает систему об изменении свойства. А система обновляет все привязанные объекты.

**18.Работа с данными СУБД в соединенной среде. Создание и настройка Command. Выполнение SQL, хранимой процедуры, операции с каталогами.**

**string** command = "SELECT \* FROM Employees";   
 SqlDataAdapter adapter = **new** SqlDataAdapter(command, connect);

<connectionStrings>

<add name="Test" connectionString="Data Source=PAUL\MSSQLSERVER1;Initial Catalog=Company;Integrated Security=True;" providerName="System.Data.SqlClient" />

</connectionStrings>

using System.Data.SqlClient;

public class DB

{

private SqlConnection sqlConnection= new SqlConnection(@"Data Source=PAUL\MSSQLSERVER1;Initial Catalog=Company;Integrated Security=True");

public void openConnection()

{

if (sqlConnection.State == System.Data.ConnectionState.Closed)

{

sqlConnection.Open();

}

}

public void closeConnection()

{

if (sqlConnection.State == System.Data.ConnectionState.Open)

{

sqlConnection.Close();

}

}

public SqlConnection GetConnection()

{

return sqlConnection;

}

}

**READER**

List<Product> list = new List<Product>();

DB database = new DB();

database.openConnection();

string sqlQuery = "SortProductByName";

using (SqlCommand command = new SqlCommand(sqlQuery, database.GetConnection()))

{

using (SqlDataReader reader = command.ExecuteReader())

{

while (reader.Read())

{

list.Add(new Product(reader.GetString(0), (double)reader.GetDecimal(1), reader.GetString(2)));

}

}

}

database.closeConnection();

**ExecuteScalar, ExecuteNonQuesrySelector**

DB database = new DB();

database.openConnection();

string sqlQuery = $"SELECT COUNT(\*) FROM TUser WHERE Name\_User = '{UserControlElement.tb1}' AND Password\_User = '{tb2.Text}'";

using (SqlCommand command = new SqlCommand(sqlQuery, database.GetConnection()))

{

int count = Convert.ToInt32(command.ExecuteScalar());

if (count <= 0)

{

string sqlQuery1 = $"INSERT INTO TUser (Name\_User, Password\_User) VALUES ('{UserControlElement.tb1}','{tb2.Text}')";

SqlCommand command1 = new SqlCommand(sqlQuery1, database.GetConnection());

{

command1.ExecuteNonQuery();

}

}

}

Операции с каталогами

SqlCommand createTableCommand = new SqlCommand("CREATE TABLE YourTable (Id INT, Name VARCHAR(50))", connection);

createTableCommand.ExecuteNonQuery();

SqlCommand command = new SqlCommand("YourStoredProcedureName", connection);

command.CommandType = CommandType.StoredProcedure;

command.Parameters.Add(new SqlParameter("@ParameterName", SqlDbType.Int) { Value = parameterValue });

command.ExecuteNonQuery();

1. **Работа с данными СУБДв соединенной среде. Создание и выполнение параметризированного запроса.**

**параметризированная переменная**

DB database = new DB();

database.openConnection();

string sqlQuery = "UPDATE Product SET Name\_Product = @Name, Price\_Product = @Price, Img\_Product = @Image WHERE IDProduct = @ID";

SqlCommand command = new SqlCommand(sqlQuery, database.GetConnection());

{

command.Parameters.AddWithValue("@Name", tb1.Text);

command.Parameters.AddWithValue("@Price", Decimal.Parse(tb2.Text));

command.Parameters.AddWithValue("@Image", tb3.Text);

command.Parameters.AddWithValue("@ID", Int32.Parse(tb4.Text));

command.ExecuteNonQuery();

}

database.closeConnection();

**20.Классы DataAdapter. Connection. DataReader. Command. Transaction. Constraint.**

Чтобы извлечь записи из базы и использовать их для наполнения таблицы в DataSet, нужно использовать другой объект ADO.NET — **DataAdapter**. DataAdapter служит мостом между одним DataTable в DataSet и источником данных. Он включает все доступные команды для выполнения запросов и обновления источника данных.

string connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString; Подключение

Класс DataReader позволяет читать данные, возвращенные командой SELECT, по одной строке за раз, в однонаправленном, доступном только для чтения потоке. Иногда это называют курсором. Использование DataReader — простейший путь получения данных, но ему недостает возможностей сортировки и связывания автономного объекта DataSet.

***Класс Command*** позволяет выполнить SQL-оператор любого типа. Хотя класс Command можно использовать для решения задач определения данных (таких как создание и изменение баз данных, таблиц и индексов), все же более вероятно его применение для выполнения задач манипулирования данными (вроде извлечения и обновления записей в таблице).

В контексте работы с данными в WPF, классы DataAdapter, Connection, DataReader, Command, Transaction и Constraint относятся к технологии ADO.NET, которая используется для взаимодействия с базами данных. Вот краткое описание каждого из этих классов:

DataAdapter:

* + DataAdapter является мостом между базой данных и набором данных (DataSet) в ADO.NET.
  + Он предоставляет методы для извлечения данных из базы данных и заполнения DataSet.
  + DataAdapter также используется для обновления изменений из DataSet в базу данных.

Connection:

* + Connection (подключение) представляет соединение с базой данных.
  + Он предоставляет методы для открытия, закрытия и управления подключением к базе данных.
  + Connection используется для передачи командам (Command) и получения данных из базы данных.

DataReader:

* + DataReader предоставляет только чтение и предоставляет доступ к данным в базе данных в режиме только для чтения.
  + Он предоставляет оптимальную производительность при чтении больших объемов данных.
  + DataReader используется для выполнения SQL-запросов и получения набора результатов.

Command:

* + Command (команда) представляет SQL-запрос или хранимую процедуру, которую нужно выполнить в базе данных.
  + Он связывается с Connection и может принимать параметры для передачи значений в SQL-запрос или хранимую процедуру.
  + Command используется для выполнения запросов и получения результатов из базы данных.

Transaction:

* + Transaction (транзакция) представляет единицу работы в базе данных, которая должна быть выполнена как единое целое.
  + Он позволяет гарантировать целостность данных и обеспечивает возможность отката (отмены) изменений, если происходит ошибка.
  + Transaction используется для управления группой команд, которые должны быть выполнены в рамках одной транзакции.

Constraint:

* + Constraint (ограничение) определяет правила целостности данных для столбцов в таблице базы данных.
  + Он может быть использован для определения уникальности значений, проверки условий или связей между таблицами.
  + Constraint помогает поддерживать целостность данных в базе данных.

В контексте WPF, эти классы могут использоваться для извлечения данных из базы данных, заполнения моделей данных, связывания данных с элементами управления WPF и обеспечения манипуляций с данными в приложении WPF.

**21.Подключение к СУБД, открытие и чтение данных с помощью DataSet. Настройка нетипизированных объектов набора DataSet.**

string connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString;

SqlCeDataAdapter da = new SqlCeDataAdapter();  
DataSet ds = new DataSet();  
DataTable dt = new DataTable();  
  
da.SelectCommand = new SqlCommand(@"SELECT \* FROM FooTable", connString);  
da.Fill(ds, "FooTable");  
dt = ds.Tables["FooTable"];  
  
foreach (DataRow dr in dt.Rows)  
{  
 MessageBox.Show(dr["Column1"].ToString());  
}

Чтение данных с использованием **DataSet**

DataSet dataSet = new DataSet();

string sqlQuery = "SELECT \* FROM MyTable";

SqlDataAdapter adapter = new SqlDataAdapter(sqlQuery, connection);

adapter.Fill(dataSet);

Использование данных из **DataSet**

DataTable table = dataSet.Tables[0];

foreach (DataRow row in table.Rows)

{

int id = (int)row["ID"];

string name = (string)row["Name"];

// Используйте данные...

}

Пример настройки нетипизированных объектов набора данных:

DataSet dataSet = new DataSet();

DataTable table = dataSet.Tables.Add("MyTable");

// Добавление столбцов

table.Columns.Add("ID", typeof(int));

table.Columns.Add("Name", typeof(string));

// Добавление других столбцов...

// Добавление данных в таблицу

table.Rows.Add(1, "John");

table.Rows.Add(2, "Jane");

// Добавление других строк...

// Использование данных

foreach (DataRow row in table.Rows)

{

int id = (int)row["ID"];

string name = (string)row["Name"];

// Используйте данные...

}

**22.Работа с данными в отсоединенной среде. Объекты DataTable. Создание DataAdapter.**

**string** connectionString =   
 WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;  
 SqlConnection connect = **new** SqlConnection(connectionString);  
 **string** command = "SELECT \* FROM Employees";  
   
 SqlDataAdapter adapter = **new** SqlDataAdapter(command, connect);  
  
 *// Заполнить DataSet*  
 *DataSet dataset =* **new** DataSet();  
 adapter.Fill(dataset, "Employees");

Работа с данными в отсоединенной среде включает использование объектов DataTable и DataAdapter. Вот примеры их создания и использования:

1. Создание объекта DataTable:

// Создание нового объекта DataTable

DataTable dataTable = new DataTable("YourTable");

// Определение столбцов

dataTable.Columns.Add("Id", typeof(int));

dataTable.Columns.Add("Name", typeof(string));

// Определение первичного ключа

dataTable.PrimaryKey = new DataColumn[] { dataTable.Columns["Id"] };

// Добавление строк в таблицу

DataRow row1 = dataTable.NewRow();

row1["Id"] = 1;

row1["Name"] = "John";

dataTable.Rows.Add(row1);

DataRow row2 = dataTable.NewRow();

row2["Id"] = 2;

row2["Name"] = "Jane";

dataTable.Rows.Add(row2);

1. Создание объекта DataAdapter:

// Создание объекта DataAdapter

SqlDataAdapter dataAdapter = new SqlDataAdapter();

// Установка команды SELECT для извлечения данных

dataAdapter.SelectCommand = new SqlCommand("SELECT \* FROM YourTable", connection);

// Создание команды INSERT для вставки данных

dataAdapter.InsertCommand = new SqlCommand("INSERT INTO YourTable (Id, Name) VALUES (@Id, @Name)", connection);

dataAdapter.InsertCommand.Parameters.Add("@Id", SqlDbType.Int, 0, "Id");

dataAdapter.InsertCommand.Parameters.Add("@Name", SqlDbType.VarChar, 50, "Name");

// Создание команды UPDATE для обновления данных

dataAdapter.UpdateCommand = new SqlCommand("UPDATE YourTable SET Name = @Name WHERE Id = @Id", connection);

dataAdapter.UpdateCommand.Parameters.Add("@Name", SqlDbType.VarChar, 50, "Name");

dataAdapter.UpdateCommand.Parameters.Add("@Id", SqlDbType.Int, 0, "Id").SourceVersion = DataRowVersion.Original;

// Создание команды DELETE для удаления данных

dataAdapter.DeleteCommand = new SqlCommand("DELETE FROM YourTable WHERE Id = @Id", connection);

dataAdapter.DeleteCommand.Parameters.Add("@Id", SqlDbType.Int, 0, "Id").SourceVersion = DataRowVersion.Original;

// Привязка DataAdapter к DataTable

dataAdapter.TableMappings.Add("Table", "YourTable");

// Заполнение DataTable данными из базы данных

dataAdapter.Fill(dataTable);

1. Использование DataAdapter для обновления данных:

// Внесение изменений в DataTable

dataTable.Rows[0]["Name"] = "Updated Name";

// Обновление базы данных

dataAdapter.Update(dataTable);

В отсоединенной среде объекты DataTable и DataAdapter позволяют работать с данными в памяти без прямого подключения к базе данных. Вы можете заполнить DataTable данными из базы данных с помощью DataAdapter, вносить изменения в DataTable и затем использовать DataAdapter для обновления изменений обратно в базу данных.

**23.Entity Framework. Поянтие Entity Data Model (EDM). Архитектура.**

Entity Framework представляет ORM-технологию, которая позволяет абстрагироваться от структуры базы данных и может выполнять автоматически сопоставление таблиц и их данных с моделями классов, определенных разработчиком.

Другим ключевым понятием является **Entity Data Model**. Эта модель сопоставляет классы сущностей с реальными таблицами в БД.

Entity Data Model состоит из трех уровней: концептуального, уровень хранилища и уровень сопоставления (маппинга).

На концептуальном уровне происходит определение классов сущностей, используемых в приложении.

Уровень хранилища определяет таблицы, столбцы, отношения между таблицами и типы данных, с которыми сопоставляется используемая база данных.

Уровень сопоставления (маппинга) служит посредником между предыдущими двумя, определяя сопоставление между свойствами класса сущности и столбцами таблиц.

Архитектура Entity Framework состоит из следующих основных компонентов:

Object Services:

* + Object Services предоставляет API для работы с концептуальной моделью и выполнения операций CRUD (Create, Read, Update, Delete) над данными.
  + Он позволяет выполнять запросы к базе данных с использованием языка LINQ (Language-Integrated Query) или SQL.

Entity SQL:

* + Entity SQL представляет язык запросов, который позволяет выполнять запросы к концептуальной модели Entity Framework.
  + Это аналог SQL, но ориентированный на работы с концептуальной моделью и объектами.

Entity Client Provider:

* + Entity Client Provider предоставляет доступ к данным из базы данных с использованием Entity Framework.
  + Он является поставщиком данных, который обеспечивает соединение с базой данных и выполнение запросов.

Entity Data Model Designer:

* + Entity Data Model Designer - инструмент визуального проектирования, который позволяет разработчикам создавать и изменять концептуальную модель EDM.
  + Это интегрированное средство разработки в Visual Studio, которое позволяет создавать классы сущностей на основе концептуальной модели.

**24.Entity Framework. Подходы к проектированию. Database-First. Model-First.Code-First**

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

* **Database first**: Entity Framework создает набор классов, которые отражают модель конкретной базы данных
* **Model first**: сначала разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере.
* **Code first**: разработчик создает класс модели данных, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблиц

**25.Entity Framework. Настройка конфигураций при CodeFirst. Способы получения связанных данных.**

Настройка конфигураций при Code First

-Аннотачции (настройка сопоставления моделей и таблиц с помощью атрибутов)

-Fluent API (набор методов, которые определяются сопоставление между классами и их свойствами и таблицами и их столбцами)

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<User>()

.ToTable("Users")

.HasKey(u => u.Id);

modelBuilder.Entity<User>()

.Property(u => u.Name)

.HasColumnName("UserName")

.HasMaxLength(50)

.IsRequired();

modelBuilder.Entity<User>()

.HasMany(u => u.Roles)

.WithMany(r => r.Users)

.Map(m =>

{

m.ToTable("UserRoles");

m.MapLeftKey("UserId");

m.MapRightKey("RoleId");

});

}

"жадная загрузка" или **eager loading**

using(DB db= new DB())

{IEnumerable<User> users = db.User.Include(p => p.Actor);

foreach( User p in users)

{

MessageBox.Show(p.Actor.Role);

}

}

"ленивая загрузка" или **lazy loading**

при первом обращении к объекту, если связанные данные не

нужны, то они не подгружаются. Однако при первом же обращении

к навигационному свойству эти данные автоматически

подгружаются из бд.

**// Жадная загрузка**

**var users = context.Users.Include(u => u.Roles).ToList();**

**// Ленивая загрузка**

**var user = context.Users.FirstOrDefault();**

**var roles = user.Roles.ToList();**

**26.Entity Framework. Аннотации. Соглашение конфигураций Fluent API. выше**

**27.Интерфейс работы с базами данных. Паттерн Repository**

Одним из наиболее часто используемых паттернов при работе с данными является паттерн 'Репозиторий'. Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.

Интерфейс работы с базами данных (Database Access Layer, DAL) представляет собой абстракцию над конкретной базой данных и предоставляет удобные методы для выполнения операций чтения, записи, обновления и удаления данных. Он обеспечивает разделение бизнес-логики приложения от деталей взаимодействия с базой данных, что повышает гибкость и облегчает сопровождение кода.

Интерфейс работы с базами данных (Data Access Layer, DAL) представляет абстракцию над способом взаимодействия с базой данных в приложении. Он определяет контракты и методы, которые позволяют приложению выполнять операции чтения, записи, обновления и удаления данных.

Один из популярных паттернов проектирования, связанных с DAL, - это паттерн Repository. Паттерн Repository разделяет логику доступа к данным от логики бизнес-объектов и предоставляет единый интерфейс для работы с данными.

public interface IRepository<T>

{

T GetById(int id);

void Add(T entity);

void Update(T entity);

void Delete(T entity);

}

public class UserRepository : IRepository<User>

{

private readonly DbContext \_context;

public UserRepository(DbContext context)

{

\_context = context;

}

public User GetById(int id)

{

return \_context.Users.FirstOrDefault(u => u.Id == id);

}

public void Add(User entity)

{

\_context.Users.Add(entity);

\_context.SaveChanges();

}

// Реализация остальных методов интерфейса

}

ПРИМЕР

var userRepository = new UserRepository(context);

var user = userRepository.GetById(1);

userRepository.Add(new User { Name = "John Doe" });

**28.Назначение и принцип использования паттерна Unit Of Work**

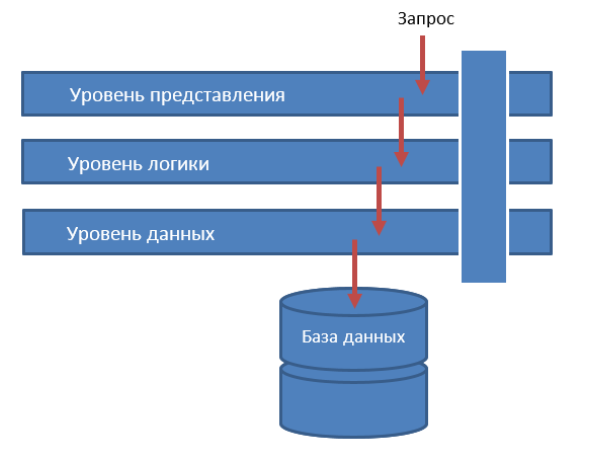
Паттерн Unit of Work позволяет упростить работу с различными репозиториями и дает уверенность, что все репозитории будут использовать один и тот же контекст данных.

Класс UnitOfWork предоставляет доступ к репозиториям через отдельные свойства и определяет общий контекст для обоих репозиториев.

Кроме того, данный класс содержит дополнительные методы Save() и Dispose(), которые в иной ситуации мы могли б определить в репозиториях. Но так как этот функционал будет общим для обоих репозиториев, то его лучше вынести в класс UnitOfWork.

**29.Архитектурные стили проектирования. Многоуровневая архитектура**

Является одной из самых известных архитектур, в которой каждый слой выполняет определенную функцию. В зависимости от ваших нужд вы можете реализовать любое количество уровней, но слишком большое их количество приведет к чрезмерному усложнению системы. Часто выделяют три основных уровня: уровень представления, уровень логики и уровень данных.



Многоуровневая архитектура обычно включает следующие слои:

Представление (Presentation Layer): Этот слой отвечает за отображение данных пользователю и обработку пользовательского ввода. Он включает в себя пользовательский интерфейс, элементы управления (controls) и логику представления (View Logic). В WPF-приложениях это может быть слой XAML-разметки и кода-behind.

Бизнес-логика (Business Logic Layer): Этот слой содержит бизнес-логику приложения. Он отвечает за обработку бизнес-правил, вычисления и взаимодействие с данными. Здесь могут находиться сервисы, менеджеры, модели и другие классы, реализующие бизнес-логику приложения.

Уровень доступа к данным (Data Access Layer): Этот слой отвечает за взаимодействие с базой данных или другими источниками данных. Он содержит логику доступа к данным, такую как запросы к базе данных, маппинг объектов данных и управление транзакциями.

Слой данных (Data Layer): Этот слой представляет собой физическое хранилище данных, такое как база данных, файловая система или веб-сервисы. Он отвечает за хранение и извлечение данных.

**30.WPF. Архитектурный паттерн Model-View-ViewModel.**

Паттерн **MVVM (Model-View-ViewModel)** позволяет отделить логику приложения от визуальной части (представления). Данный паттерн является архитектурным, то есть он задает общую архитектуру приложения.

MVVM состоит из трех компонентов: модели (Model), модели представления (ViewModel) и представления (View).

### **Model**

Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную этими данными, например, логику валидации свойств модели. В то же время модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления.

Нередко модель реализует интерфейсы INotifyPropertyChanged или INotifyCollectionChanged, которые позволяют уведомлять систему об изменениях свойств модели. Благодаря этому облегчается привязка к представлению, хотя опять же прямое взаимодействие между моделью и представлением отсутствует.

### **View**

View или представление определяет визуальный интерфейс, через который пользователь взаимодействует с приложением. Применительно к WPF представление - это код в xaml, который определяет интерфейс в виде кнопок, текстовых полей и прочих визуальных элементов.

Хотя окно (класс Window) в WPF может содержать как интерфейс в xaml, так и привязанный к нему код C#, однако в идеале код C# не должен содержать какой-то логики, кроме разве что конструктора, который вызывает метод InitializeComponent и выполняет начальную инициализацию окна. Вся же основная логика приложения выносится в компонент ViewModel.

Однако иногда в файле связанного кода все может находиться некоторая логика, которую трудно реализовать в рамках паттерна MVVM во ViewModel.

Представление не обрабатывает события за редким исключением, а выполняет действия в основном посредством команд.

### **ViewModel**

ViewModel или модель представления связывает модель и представление через механизм привязки данных. Если в модели изменяются значения свойств, при реализации моделью интерфейса INotifyPropertyChanged автоматически идет изменение отображаемых данных в представлении, хотя напрямую модель и представление не связаны.

ViewModel также содержит логику по получению данных из модели, которые потом передаются в представление. И также VewModel определяет логику по обновлению данных в модели.

Поскольку элементы представления, то есть визуальные компоненты типа кнопок, не используют события, то представление взаимодействует с ViewModel посредством команд.

Например, пользователь хочет сохранить введенные в текстовое поле данные. Он нажимает на кнопку и тем самым отправляет команду во ViewModel. А ViewModel уже получает переданные данные и в соответствии с ними обновляет модель.

Итогом применения паттерна MVVM является функциональное разделение приложения на три компонента, которые проще разрабатывать и тестировать, а также в дальнейшем модифицировать и поддерживать.

Преимущества MVVM:

* Разделение ответственности: MVVM позволяет разделить логику представления, бизнес-логику и данные, что облегчает сопровождение и повторное использование кода.
* Тестирование: MVVM обеспечивает высокую тестируемость, поскольку ViewModel может быть легко тестируемым без зависимости от представления или модели.
* Гибкость и удобство разработки: MVVM позволяет разработчикам более гибко работать над представлением и моделью, не заботясь о зависимостях между ними.