**КУРНОСЕНКО СОФЬЯ**

**№ 2 Windows Forms. Элементы управления**

**Задание**

Создать форму (или формы для ввода агрегируемых объектов).

Разместить на ней ЭУ для ввода/вывода информации об объекте (создать свои

типы). На форме разместить не менее **9-и различных типов ЭУ** (радиокнопки,

списки, поля ввода, метки, кнопки, слайдеры, календарь и т.д.).

Создать дополнительные кнопки для сохранения введенной

информации и вывода (отображения сохраненных данных). Запись

сохраняемых объектов и чтение выполнять в./из файл типа xml и/или json.

Выполнить валидацию вводимых пользователем данных.

**8 вариант: Магазин.** Основной объект «Товар». Поля: название, инвентарный номер, размер, вес, тип, дата поступления, количество, цена, производитель. Агрегируемый объект «Производитель». Поля: организация, страна, адрес, телефон.

*Дополнительно: Агрегируемый объект «Кладовщик». Поля: ФИО, стаж, адрес и т.д.*

**Наследование, композиция и агрегация**

►**Между классами возможны два типа отношений:**

1. Отношение типа **is-a** (есть, является), при котором один класс является подвидом другого класса. При таком отношении один класс расширяет (детализирует) возможности другого класса. Расширение возможностей класса осуществляется благодаря использованию наследования.

2. Отношение, при котором существует взаимосвязь между двумя классами. Здесь выделяют два подвида взаимосвязи между классами:

2.1. Отношение типа **has-a** (класс содержит объект другого класса). В этом случае в классе объявляется один или несколько экземпляров другого класса. При данном отношении возможны два случая взаимодействия:

■ Первый случай, это когда объект (экземпляр), который объявлен в классе, не является составной частью класса (**агрегация**) и его использование не влияет на функциональную работу класса.

**Пример агрегации**

Класс Figures может содержать разное количество разных фигур (даже 0). По возможности к классу Figures могут быть добавлены массивы других фигур, например, Rectangle (прямоугольник). В любом случае, класс Figures будет полноценным функционально, значит, это есть агрегация.

// Класс треугольник

class Triangle

{

// Методы и поля класса Triangle

// ...

};

// Класс, реализующий окружность

class Circle

{

// Методы и поля класса Circle

// ...

};

// Класс, реализующий различные геометрические фигуры.

// Используется тип отношения - агрегация.

class Figures

{

Triangle tr[10]; // массив треугольников

unsigned int n\_tr; // количество треугольников в массиве tr

Circle cr[10]; // массив окружностей

unsigned int n\_cr; // количество окружностей в массиве cr

// Другие поля и методы класса

// ...

};

■ Второй случай, когда объект, который объявлен в классе, является составной частью этого класса (**композиция**).

**Пример композиции**

Класс Bike (Велосипед) содержит экземпляры классов Wheel (Колесо) и Saddle (седло), которые являются его составной частью. Велосипед не может быть без колес или седла, поэтому это есть композиция.

// Класс Колесо

class Wheel

{

// Поля и методы класса

// ...

};

// Класс Велосипед, содержит колеса, седло

// Класс седло

class Saddle

{

// Поля и методы класса

// ...

};

// Класс Велосипед, содержит объекты классов,

// которые есть частью Велосипеда (седло, колеса) - это композиция

class Bike

{

Saddle sd; // одно седло - часть, дополняющая велосипед

Wheel whl[2]; // два колеса - часть велосипеда

// Поля и методы класса Bike

// ...

};

2.2. Отношение типа **uses** (класс «использует другой класс). В этом случае класс содержит программный код другого вложенного класса, к которому он имеет доступ.

**Элементы управления**

Элементы управления представляют собой визуальные классы, которые получают введенные пользователем данные и могут инициировать различные события. 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

▪ 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

**Кнопка**

Наиболее часто используемым элементом управления является кнопка. Обрабатывая событие нажатия кнопки, мы может производить те или иные действия.

При двойном нажатии на кнопку на форме в редакторе Visual Studio мы по умолчанию попадаем в код обработчика события Click, который будет выполняться при нажатии:

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

{ }

**Оформление кнопки**

Чтобы управлять внешним отображением кнопки, можно использовать свойство **FlatStyle**. Оно может принимать следующие значения:

▪ Flat - Кнопка имеет плоский вид

▪ Popup - Кнопка приобретает объемный вид при наведении на нее указателя, в иных случаях она имеет плоский вид

▪ Standard - Кнопка имеет объемный вид (используется по умолчанию)

▪ System - Вид кнопки зависит от операционной системы

**Изображение на кнопке**

Как и для многих элементов управления, для кнопки можно задавать изображение с помощью свойства BackgroundImage. Однако мы можем также управлять размещением текста и изображения на кнопки. Для этого надо использовать свойство TextImageRelation. Оно приобретает следующие значения:

▪ Overlay: текст накладывается на изображение

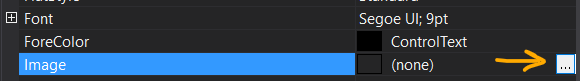
▪ ImageAboveText: изображение располагается над текстом

▪ TextAboveImage: текст располагается над изображением

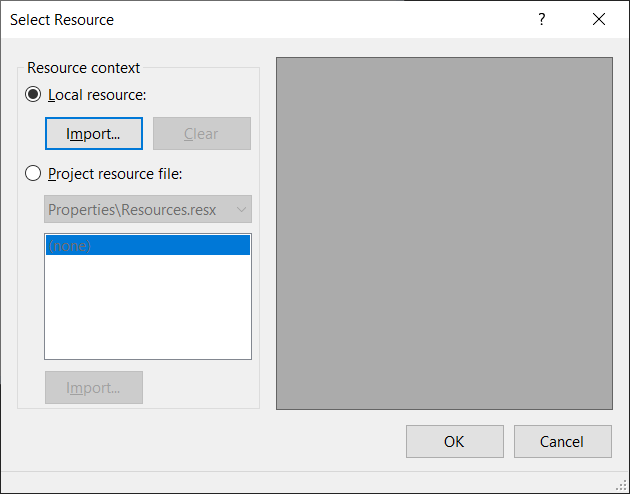
▪ ImageBeforeText: изображение располагается перед текстом

▪ TextBeforeImage: текст располагается перед изображением

Например, установим для кнопки не фоновое изображение. Для этого выберем кнопку и в окне Свойств нажмем на поле Image (не путать с BackgroundImage).

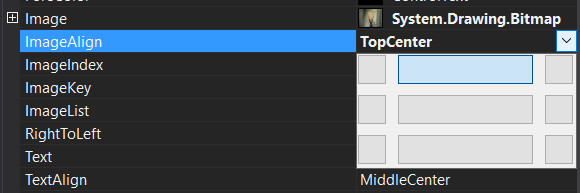


Нам откроется диалоговое окно установки изображения:



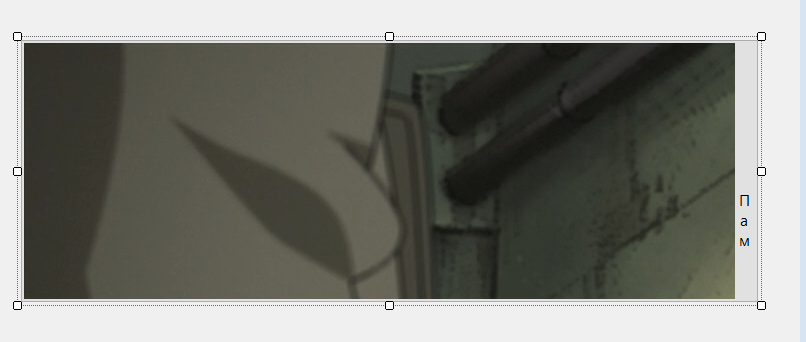
В этом окне выберем опцию Local Resource и нажмем на кнопку Import, после чего нам откроется диалоговое окно для выбора файла изображения.

После выбора изображения мы можем установить свойство ImageAlign, которое управляет позиционированием изображения на кнопке:



Нам доступны 9 вариантов, с помощью которых мы можем прикрепить изображение к определенной стороне кнопки.

Затем перейдем к свойству TextImageRelation и установим для него значение ImageBeforeText. В итоге мы получим кнопку, где сразу после изображения идет надпись на кнопке:



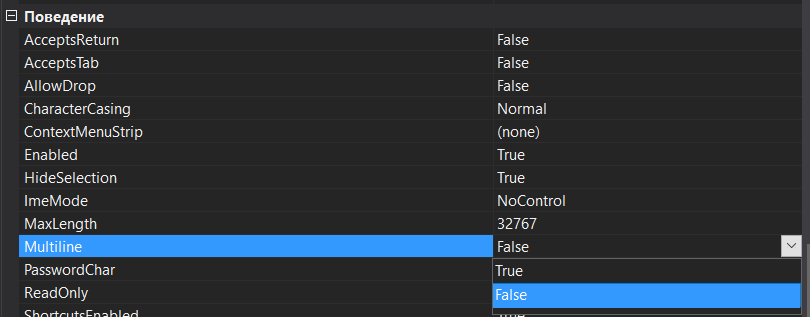
**Label**

Для отображения простого текста на форме, доступного только для чтения, служит элемент Label. Чтобы задать отображаемый текст метки, надо установить свойство Text элемента.

**Текстовое поле TextBox**

Для ввода и редактирования текста предназначены текстовые поля – элемент TextBox. Так же как и у элемента Label текст элемента TextBox можно установить или получить с помощью свойства Text.

По умолчанию при переносе элемента с панели инструментов создается однострочное текстовое поле. Для отображения больших объемов информации в текстовом поле нужно использовать его свойства Multiline и ScrollBars. При установке для свойства Multiline значения true, появляется возможность изменить высоту блока в конструкторе формы.



Кроме того, можно сделать прокрутку текстового поля, установив для его свойства ScrollBars одно из значений:

▪ None: без прокруток (по умолчанию)

▪ Horizontal: создает горизонтальную прокрутку при длине строки, превышающей ширину текстового поля

▪ Vertical: создает вертикальную прокрутку, если строки не помещаются в текстовом поле

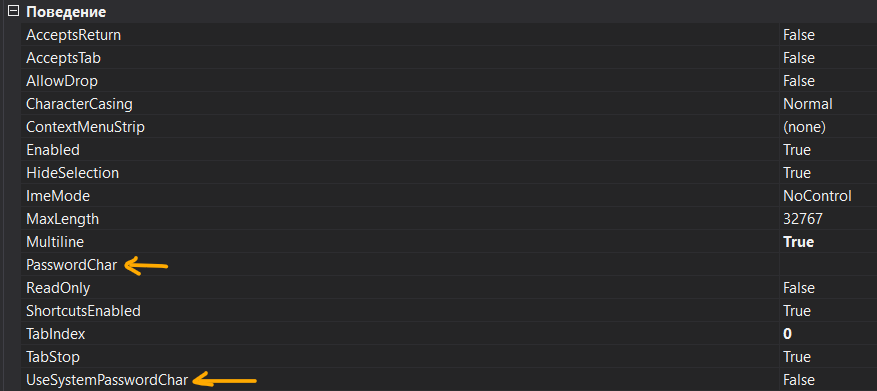
▪ Both: создает вертикальную и горизонтальную прокрутку

**Перенос по словам**

Чтобы текст в элементе TextBox переносился по словам, надо установить свойство WordWrap равным true. То есть если одно слово не умещается на строке, то но переносится на следующую. Данное свойство будет работать только для многострочных текстовых полей.

**Ввод пароля**

Также данный элемент имеет свойства, которые позволяют сделать из него поле для ввода пароля. Так, для этого надо использовать PasswordChar и UseSystemPasswordChar.



Свойство PasswordChar по умолчанию не имеет значение, если мы установим в качестве него какой-нибудь символ, то этот символ будут отображаться при вводе любых символов в текстовое поле.

Свойство UseSystemPasswordChar имеет похожее действие. Если мы установим его значение в true, то вместо введенных символов в текстовом поле будет отображаться знак пароля, принятый в системе, например, точка.

**Событие TextChanged**

Из всех событий элемента TextBox следует отметить событие TextChanged, которое срабатывает при изменении текста в элементе. Например, поместим на форму кроме текстового поля метку и сделаем так, чтобы при изменении текста в текстовом поле также менялся текст на метке:

public partial class Form1 : Form

{

    public Form1()

    {

        InitializeComponent();

        textBox1.TextChanged += textBox1\_TextChanged;

    }

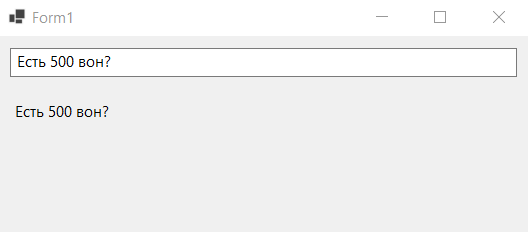
     private void textBox1\_TextChanged(object sender, EventArgs e)

    {

        label1.Text = textBox1.Text;

    }

}



**Элемент MaskedTextBox**

Элемент MaskedTextBox по сути представляет обычное текстовое поле. Однако данный элемент позволяет контролировать ввод пользователя и проверять его автоматически на наличие ошибок.

Чтобы контролировать вводимые в поле символы, надо задать маску. Для задания маски можно применять следующие символы:

▪ 0: Позволяет вводить только цифры

▪ 9: Позволяет вводить цифры и пробелы

▪ #: Позволяет вводить цифры, пробелы и знаки '+' и '-'

▪ L: Позволяет вводить только буквенные символы

▪ ?: Позволяет вводить дополнительные необязательные буквенные символы

▪ A: Позволяет вводить буквенные и цифровые символы

▪.: Задает позицию разделителя целой и дробной части

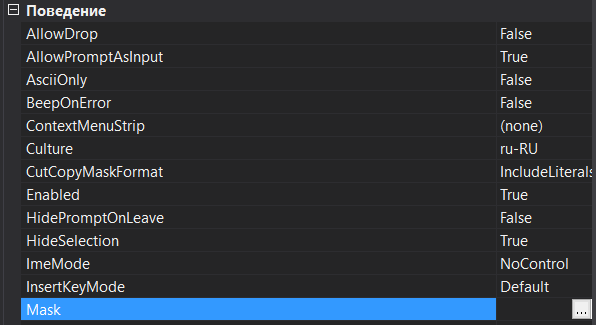
▪ ,: Используется для разделения разрядов в целой части числа

▪ :: Используется в временных промежутках - разделяет часы, минуты и секунды

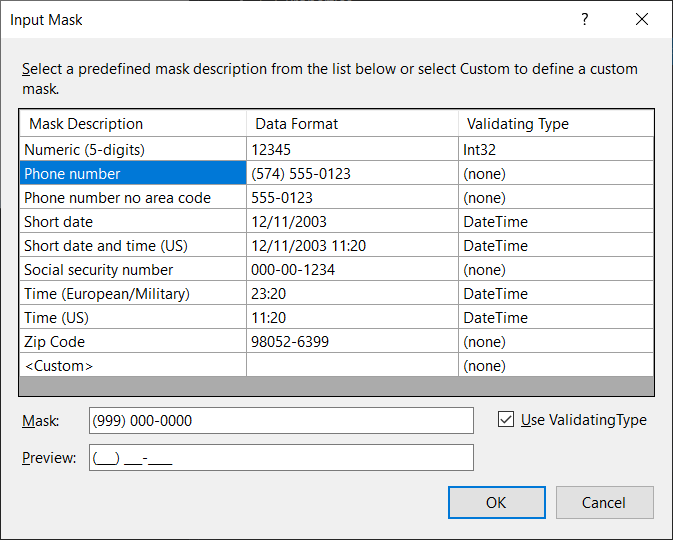
▪ /: Используется для разделения дат

▪ $: Используется в качестве символа валюты

Чтобы задать маску, надо установить ***свойство Mask*** элемента.

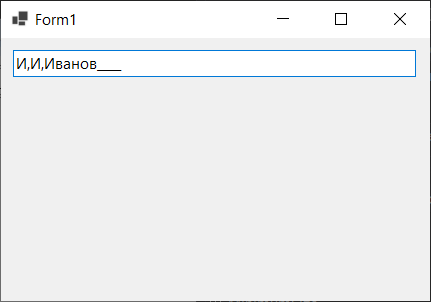


Найдя это свойство в окне свойств, нажмем на него и нам отобразится окно для задания одного из стандартных шаблонов маски. В частности мы можем выбрать Phone number (Телефонный номер), который подразумевает ввод в текстовое поле только телефонного номера:



Теперь при запуске мы сможем ввести в текстовое поле только цифры, получив в итоге телефонный номер.

Теперь сделаем свою маску. Например, создадим маску для ввода инициалов имени и отчества и фамилий ограниченной длины в текстовое поле. Для этого присвоим свойству Mask значение L.L.L?????????. Тогда ввод в текстовое поле будет выглядеть следующим образом:



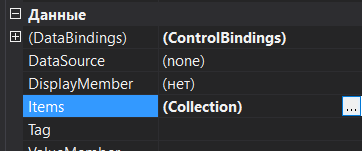
Свойство **PromptChar** указывает на символ, который отображается в поле на месте ввода символов. По умолчанию стоит знак подчеркивания.

Свойство **AsciiOnly** при значении true позволяет вводить только ascii-символы, то есть символы из диапазона A-Z и a-z.

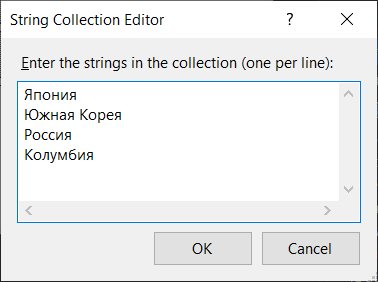
**ListBox**

Элемент ListBox представляет собой простой список. Ключевым свойством этого элемента является свойство Items, которое как раз и хранит набор всех элементов списка.

Элементы в список могут добавляться как во время разработки, так и программным способом. В Visual Studio в окне Properties (Свойства) для элемента ListBox мы можем найти свойство Items.



После щелчка по троеточию нам отобразится окно для добавления элементов в список:



В пустое поле мы вводим по одному элементу списка - по одному на каждой строке. После этого все добавленные нами элементы окажутся в списке, и мы сможем ими управлять.

Получаем элемент, где выбор вариантов из коллекции осуществляется при нажатии на стрелки вверх и вниз:



**Программное управление элементами в ListBox**

**Добавление элементов**

Итак, все элементы списка входят в свойство Items, которое представляет собой коллекцию. Для добавления нового элемента в эту коллекцию, а значит и в список, надо использовать метод Add, например: listBox1.Items.Add("Новый элемент");. При использовании этого метода каждый добавляемый элемент добавляется в конец списка.

Можно добавить сразу несколько элементов, например, массив. Для этого используется метод AddRange:

string[] countries = { "Бразилия", "Аргентина", "Чили", "Уругвай", "Колумбия" };

listBox1.Items.AddRange(countries);

**Вставка элементов**

В отличие от простого добавления вставка производится по определенному индексу списка с помощью метода Insert:

listBox1.Items.Insert(1, "Парагвай");

В данном случае вставляем элемент на вторую позицию в списке, так как отсчет позиций начинается с нуля.

**Удаление элементов**

Для удаления элемента по его тексту используется метод Remove:

listBox1.Items.Remove("Чили");

Чтобы удалить элемент по его индексу в списке, используется метод RemoveAt:

listBox1.Items.RemoveAt(1);

Кроме того, можно очистить сразу весь список, применив метод Clear:

listBox1.Items.Clear();

Доступ к элементам списка

Используя индекс элемента, можно получить сам элемент в списке. Например, получим первый элемент списка:

string firstElement = listBox1.Items[0];

Свойство Count возвращает количество элементов в списке:

int number = listBox1.Items.Count;

**Выделение элементов списка**

При выделении элементов списка мы можем ими управлять как через индекс, так и через сам выделенный элемент. Получить выделенные элементы можно с помощью следующих свойств элемента ListBox:

▪ SelectedIndex: возвращает или устанавливает номер выделенного элемента списка. Если выделенные элементы отсутствуют, тогда свойство имеет значение -1

▪ SelectedIndices: возвращает или устанавливает коллекцию выделенных элементов в виде набора их индексов

▪ SelectedItem: возвращает или устанавливает текст выделенного элемента

▪ SelectedItems: возвращает или устанавливает выделенные элементы в виде коллекции

По умолчанию список поддерживает выделение одного элемента. Чтобы добавить возможность выделения нескольких элементов, надо установить у его свойства *SelectionMode* значение MultiSimple.

Чтобы выделить элемент программно, надо применить метод SetSelected(int index, bool value), где index - номер выделенного элемента. Если второй параметр - value имеет значение true, то элемент по указанному индексу выделяется, если false, то выделение наоборот скрывается:

listBox1.SetSelected(2, true); // будет выделен третий элемент

Чтобы снять выделение со всех выделенных элементов, используется метод *ClearSelected*.

**Событие SelectedIndexChanged**

Из всех событий элемента ListBox надо отметить в первую очередь событие SelectedIndexChanged, которое возникает при изменении выделенного элемента:

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

string[] countries = { "Бразилия", "Аргентина", "Чили", "Уругвай", "Колумбия" };

listBox1.Items.AddRange(countries);

listBox1.SelectedIndexChanged += listBox1\_SelectedIndexChanged;

}

void listBox1\_SelectedIndexChanged(object sender, EventArgs e)

{

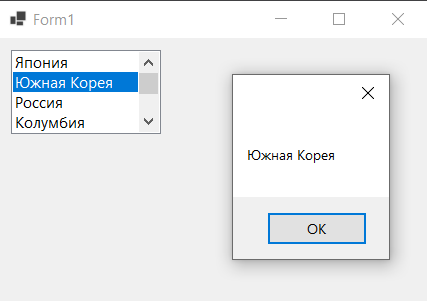
string selectedCountry = listBox1.SelectedItem.ToString();

MessageBox.Show(selectedCountry);

}

}

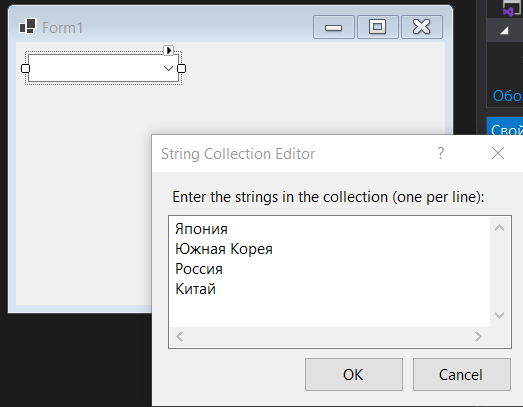
В данном случае по выбору элемента списка будет отображаться сообщение с выделенным элементом.



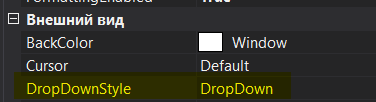
**Элемент ComboBox**

Элемент ComboBox образует выпадающий список и совмещает функциональность компонентов ListBox и TextBox. Для хранения элементов списка в ComboBox также предназначено свойство Items.

Подобным образом, как и с ListBox, мы можем в окне свойств на свойство Items и нам отобразится окно для добавления элементов ComboBox:



В отличие от ListBox в ComboBox мы можем ввести *свое* значение в поле, однако это зависит от значения свойства DropDownStyle (об этом ниже):



Помимо этого ComboBox позволяет открывать выпадающий список элементов коллекции, а ListBox – нет.

И как и с компонентом ListBox, здесь мы также можем программно управлять элементами.

**Добавление элементов:**

// добавляем один элемент

comboBox1.Items.Add("Парагвай");

// добавляем набор элементов

comboBox1.Items.AddRange(new string[] { "Уругвай", "Эквадор" });

// добавляем один элемент на определенную позицию

comboBox1.Items.Insert(1, "Боливия");

При добавлении с помощью методов Add / AddRange все новые элементы помещаются в конец списка. Однако если мы зададим у ComboBox *свойство Sorted* равным true, тогда при добавлении будет автоматически производиться сортировка.

**Удаление элементов:**

// удаляем один элемент

comboBox1.Items.Remove("Аргентина");

// удаляем элемент по индексу

comboBox1.Items.RemoveAt(1);

// удаляем все элементы

comboBox1.Items.Clear();

Мы можем получить элемент по индексу и производить с ним разные действия. Например, изменить его:

comboBox1.Items[0] = "Парагвай";

**Настройка оформления ComboBox**

С помощью ряда свойств можно настроить стиль оформления компонента. Так, свойство DropDownWidth задает ширину выпадающего списка. С помощью свойства DropDownHeight можно установить высоту выпадающего списка.

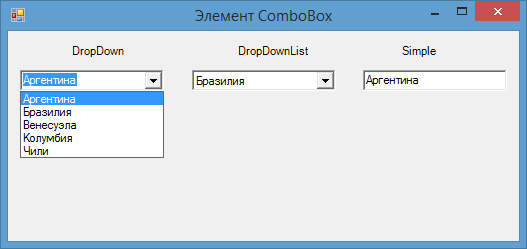
Еще одно свойство MaxDropDownItems позволяет задать число видимых элементов списка - от 1 до 100. По умолчанию это число равно 8.

Другое свойство DropDownStyle задает стиль ComboBox. Оно может принимать три возможных значения:

▪ Dropdown: используется по умолчанию. Мы можем открыть выпадающий список вариантов при вводе значения в текстовое поле или нажав на кнопку со стрелкой в правой части элемента, и нам отобразится собственно выпадающий список, в котором можно выбрать возможный вариант

▪ DropdownList: чтобы открыть выпадающий список, надо нажать на кнопку со стрелкой в правой стороне элемента. В данном случае пользователь *не может ввести текст в поле*, а может только выбрать значение из списка

▪ Simple: ComboBox представляет простое текстовое поле, в котором для перехода между элементами мы можем использовать клавиши клавиатуры вверх/вниз



**Событие SelectedIndexChanged**

Наиболее важным событием для ComboBox также является событие SelectedIndexChanged, позволяющее отследить выбор элемента в списке:

public partial class Form1 : Form

{

    public Form1()

    {

        InitializeComponent();

         comboBox1.SelectedIndexChanged += comboBox1\_SelectedIndexChanged;

    }

     void comboBox1\_SelectedIndexChanged(object sender, EventArgs e)

    {

        string selectedState = comboBox1.SelectedItem.ToString();

        MessageBox.Show(selectedState);

    }

}

Здесь также свойство SelectedItem будет ссылаться на выбранный элемент.

Также мы можем обратиться к значению свойства Text элемента ComboBox. В примере ниже значение свойства Text, то есть выбранный элемент списка отображается в элементе label, находящимся под полем:

using System;

using System.Windows.Forms;

namespace Laba2

{

    public partial class Form1 : Form

    {

        public Form1()

        {

            InitializeComponent();

comboBox1.SelectedIndexChanged += comboBox1\_SelectedIndexChanged;

}

  private void comboBox1\_SelectedIndexChanged(object sender, EventArgs e)

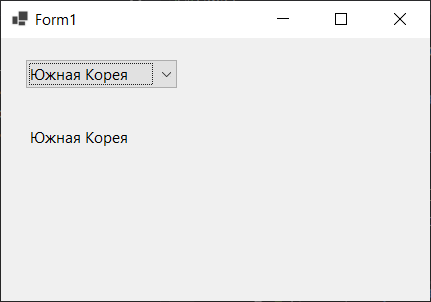
{

label1.Text = comboBox1.Text;

}

}

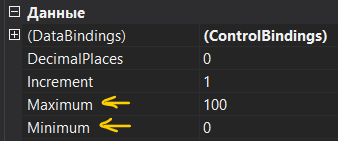
}



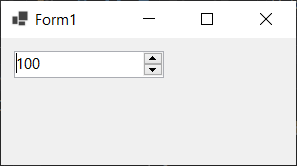
**Элементы NumericUpDown и DomainUpDown**

**NumericUpDown**

Элемент NumericUpDown предоставляет пользователю выбор числа из определенного диапазона. Для определения диапазона чисел для выбора NumericUpDown имеет два свойства: Minimum (задает минимальное число) и Maximum (задает максимальное число).



Само значение элемента хранится в свойстве **Value**:



По умолчанию элемент отображает десятичные числа. Однако если мы установим его свойство Hexadecimal равным true, то элемент будет отображать все числа в шестнадцатеричной системе.

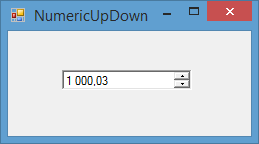
Даже если мы в коде установим обычное десятичное значение:

numericUpDown1.Value = 66;

то элемент все равно отобразит его в шестнадцатиричной системе.

Если мы хотим отображать в поле дробные числа, то можно использовать свойство **DecimalPlaces**, которое указывает, сколько знаков после запятой должно отображаться. По умолчанию это свойство равно нулю.

Также можно задать отображение тысячного разделителя. Для этого для свойства ThousandsSeparator надо установить значение true. Например, numericUpDown при Value=1000,03, DecimalPlaces=2 и ThousandsSeparator=true:



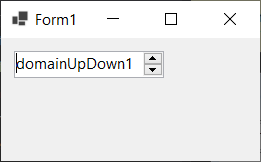
При этом надо учитывать, что если мы устанавливаем значение для свойства Value в окне свойств, то там в качестве разделителя целой и дробной части используется запятая. Если же мы устанавливаем данное свойство в коде, тогда в качестве разделителя используется точка.

По умолчанию при нажатии на стрелочки вверх-вниз на элементе значение будет увеличиваться, либо уменьшаться на единицу. Но с помощью свойства **Increment** можно задать другой шаг приращения, в том числе и дробный.

При работе с NumericUpDown следует учитывать, что его ***свойство Value*** (как и свойства Minimum и Maximum) ***хранит значение decimal***. Поэтому в коде мы также должны с ним работать как с decimal, а не как с типом int или double.

**DomainUpDown**

Элемент DomainUpDown предназначен для ввода текстовой информации. Он имеет текстовое поле для ввода строки и две стрелки для перемещения по списку строк:



Список для DomainUpDown задается с помощью свойства **Items**. Список можно сразу упорядочить по алфавиту. Для этого надо свойству **Sorted** присвоить значение true.

Чтобы можно было циклично перемещаться по списку, то есть при достижении конца или начала списка его просмотр начинался с первого или последнего элемента, надо установить для свойства **Wrap** значение true.

В коде выбранное значение в DomainUpDown доступно через свойство **Text**. Например, добавим программно список строк в DomainUpDown и обработаем изменение выбора в списке:

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

List<string> states = new List<string>

{

"Аргентина", "Бразилия", "Венесуэла", "Колумбия", "Чили"

};

// добавляем список элементов

domainUpDown1.Items.AddRange(states);

domainUpDown1.TextChanged += domainUpDown1\_TextChanged;

}

// обработка изменения текста в элементе

void domainUpDown1\_TextChanged(object sender, EventArgs e)

{

MessageBox.Show(domainUpDown1.Text);

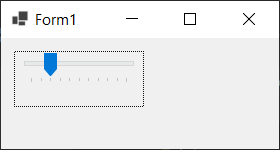
}

}

Для обработки изменения текста здесь также, как и для элемента TextBox, можно использовать событие TextChanged, в обработчике которого мы выводим выбранный текст в сообщение.

**TrackBar**

TrackBar представляет собой элемент, который с помощью перемещения ползунка позволяет вводить числовые значения.



Некоторые важные свойства TrackBar:

▪ Orientation: задает ориентацию ползунка - расположение по горизонтали или по вертикали

▪ TickStyle: задает расположение делений на ползунке

▪ TickFrequency: задает частоту делений на ползунке

▪ Minimum: минимальное возможное значение на ползунке (по умолчанию 0)

▪ Maximum: максимальное возможное значение на ползунке (по умолчанию 10)

▪ Value: текущее значение ползунка. Должно находиться между Minimum и Maximum

Свойство TickStyle может принимать ряд значений:

▪ None: деления отсутствуют

▪ Both: деления расположены по обеим сторонам ползунка

▪ BottomRight: у вертикального ползунка деления находятся справа, а у горизонтального - снизу

▪ TopLeft: у вертикального ползунка деления находятся слева, а у горизонтального - сверху (применяется по умолчанию)

К наиболее важным событиям элемента следует отнести событие **Scroll**, которое позволяет обработать перемещение ползунка от одного деления к другому. Что может быть полезно, если нам надо, например, устанавливать соответствующую громкость звука в зависимости от значения ползунка, либо какие-нибудь другие настройки:

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

// установка обработчика события Scroll

trackBar1.Scroll += trackBar1\_Scroll;

}

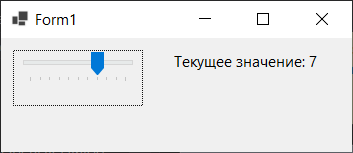
private void trackBar1\_Scroll(object sender, EventArgs e)

{

label1.Text = String.Format("Текущее значение: {0}", trackBar1.Value);

}

}

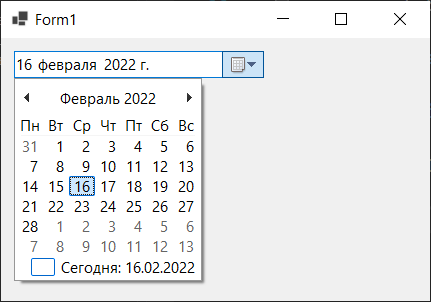


**DateTimePicker и MonthCalendar**

Для работы с датами в Windows Forms имеются элементы DateTimePicker и MonthCalendar.

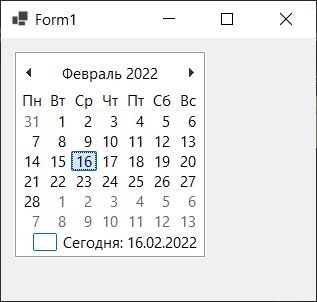
**DateTimePicker**

DateTimePicker представляет раскрывающийся по нажатию календарь, в котором можно выбрать дату.



**MonthCalendar**

С помощью MonthCalendar также можно выбрать дату, только в данном случае этот элемент представляет сам календарь, который не надо раскрывать:



Свойства для определения дат в календаре:

▪ MinDate: определяет минимальную дату для выбора в календаре

▪ MaxDate: задает наибольшую дату для выбора в календаре

▪ FirstDayOfWeek: определяет день недели, с которого должна начинаться неделя в календаре

▪ ShowToday: при значении true отображает внизу календаря текущую дату

▪ TodayDate: определяет текущую дату. По умолчанию используется системная дата на компьютере, но с помощью данного свойства мы можем ее изменить

Наиболее интересными событиями элемента являются события **DateChanged** и **DateSelected**, которые возникают при изменении выбранной в элементе даты. Однако надо учитывать, что выбранная дата будет представлять первую дату из диапазона выделенных дат:

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

monthCalendar1.DateChanged += monthCalendar1\_DateChanged;

}

void monthCalendar1\_DateChanged(object sender, DateRangeEventArgs e)

{

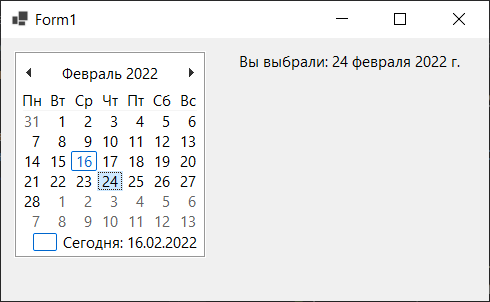
label1.Text = String.Format("Вы выбрали: {0}", e.Start.ToLongDateString());

// или так - аналогичный код

//label1.Text = String.Format("Вы выбрали: {0}", monthCalendar1.SelectionStart.ToLongDateString());

}

}



**Окно сообщения MessageBox**

Как правило, для вывода сообщений применяется элемент MessageBox. Однако кроме собственно вывода строки сообщения данный элемент может устанавливать ряд настроек, которые определяют его поведение.

Для вывода сообщения в классе MessageBox предусмотрен метод **Show**, который имеет различные версии и может принимать ряд параметров. Рассмотрим одну из наиболее используемых версий:

public static DialogResult Show(

    string text,

    string caption,

    MessageBoxButtons buttons,

    MessageBoxIcon icon,

    MessageBoxDefaultButton defaultButton,

    MessageBoxOptions options

)

Здесь применяются следующие параметры:

*→ text*: текст сообщения

*→ caption*: текст заголовка окна сообщения

*→ buttons*: кнопки, используемые в окне сообщения. Принимает одно из значений перечисления MessageBoxButtons:

▪ AbortRetryIgnore: три кнопки Abort (Отмена), Retry (Повтор), Ignore (Пропустить)

▪ OK: одна кнопка OK

▪ OKCancel: две кнопки OK и Cancel (Отмена)

▪ RetryCancel: две кнопки Retry (Повтор) и Cancel (Отмена)

▪ YesNo: две кнопки Yes и No

▪ YesNoCancel: три кнопки Yes, No и Cancel (Отмена)

Таким образом, в зависимости от выбора окно сообщения может иметь от одной до трех кнопок.

*→ icon*: значок окна сообщения. Может принимать одно из следующих значений перечисления MessageBoxIcon:

▪ Asterisk, Information: значок, состоящий из буквы i в нижнем регистре, помещенной в кружок

▪ Error, Hand, Stop: значок, состоящий из белого знака "X" на круге красного цвета.

▪ Exclamation, Warning: значок, состоящий из восклицательного знака в желтом треугольнике

▪ Question: значок, состоящий из вопросительного знака на периметре круга

▪ None: значок у сообщения отсутствует

*→ defaultButton*: кнопка, на которую по умолчанию устанавливается фокус. Принимает одно из значений перечисления MessageBoxDefaultButton:

▪ Button1: первая кнопка из тех, которые задаются перечислением MessageBoxButtons

▪ Button2: вторая кнопка

▪ Button3: третья кнопка

*→ options*: параметры окна сообщения. Принимает одно из значений перечисления MessageBoxOptions:

▪ DefaultDesktopOnly: окно сообщения отображается на активном рабочем столе.

▪ RightAlign: текст окна сообщения выравнивается по правому краю

▪ RtlReading: все элементы окна располагаются в обратном порядке справа налево

▪ ServiceNotification: окно сообщения отображается на активном рабочем столе, даже если в системе не зарегистрирован ни один пользователь

Нередко используется только один параметр – текст сообщения. Но посмотрим, как использовать остальные параметры. Пусть у нас есть кнопка, в обработчике нажатия которой открывается следующее окно сообщения:

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

{

MessageBox.Show(

"Выберите один из вариантов",

"Сообщение",

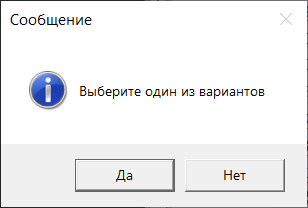
MessageBoxButtons.YesNo,

MessageBoxIcon.Information,

MessageBoxDefaultButton.Button1,

MessageBoxOptions.DefaultDesktopOnly);

}



Однако нам не просто дается возможность установки кнопок в окне сообщения. Метод MessageBox.Show возвращает объект DialogResult, с помощью которого мы можем узнать, какую кнопку в окне сообщения нажал пользователь. **DialogResult** представляет перечисление, в котором определены следующие значения:

▪ Abort: нажата кнопка Abort

▪ Retry: нажата кнопка Retry

▪ Ignore: нажата кнопка Ignore

▪ OK: нажата кнопка OK

▪ Cancel: нажата кнопка Cancel

▪ None: отсутствие результата

▪ Yes: нажата кнопка Yes

▪ No: нажата кнопка No

Используем обработку выбора пользователя, изменив обработчик нажатия кнопки следующим образом:

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

{

DialogResult result = MessageBox.Show(

"Окрасить кнопку в красный цвет?",

"Сообщение",

MessageBoxButtons.YesNo,

MessageBoxIcon.Information,

MessageBoxDefaultButton.Button1,

MessageBoxOptions.DefaultDesktopOnly);

if (result == DialogResult.Yes)

button1.BackColor = Color.Red;

}

И теперь, если в окне сообщения мы выберем выриант Yes, то кнопка окрасится в красный цвет.

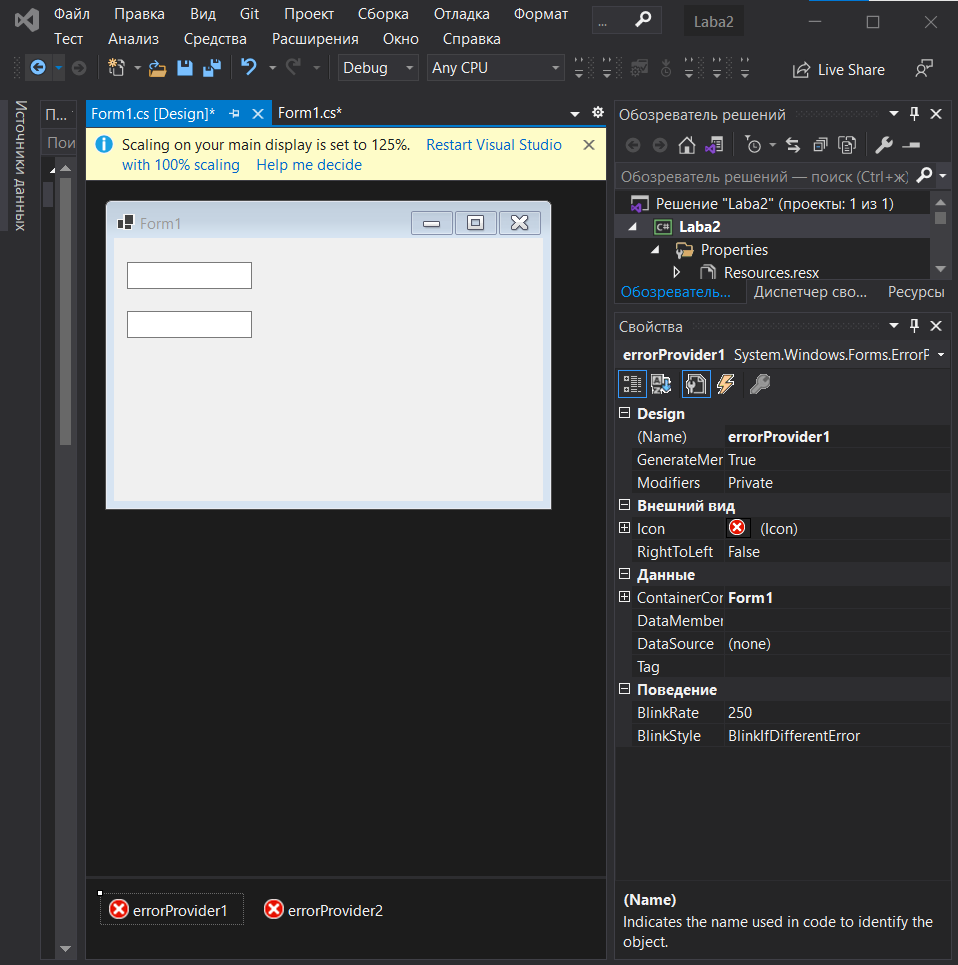
**ErrorProvider**

ErrorProvider не является полноценным визуальным компонентом, тем не менее он позволяет настраивать визуальное отображение ошибок при вводе пользователя. Этот элемент применяется преимущественно для проверки и индикации ошибок.

Так, определим на форме два текстовых поля с именами nameBox и ageBox. И далее перетащим с панели инструментов на форму элемент ErrorProvider.



ErrorProvider отобразится под формой.



А в окне свойств мы также сможем управлять его свойствами:

Среди его свойств можно выделить следующие:

▪ BlinkRate: задает частоту мигания значка ошибки

▪ BlinkStyle: задает, когда значок ошибки будет мигать

▪ Icon: устанавливает сам значок ошибки. По умолчанию это красный кружок с восклицательным знаком, но можно установить любую другую иконку.

В коде формы приложения пропишем следующее:

using System;

using System.ComponentModel;

using System.Windows.Forms;

namespace Laba2

{

    public partial class Form1 : Form

    {

        public Form1()

        {

            InitializeComponent();

            nameBox.Validating += nameBox\_Validating;

            ageBox.Validating += ageBox\_Validating;

        }

        private void nameBox\_Validating(object sender, CancelEventArgs e)

        {

            if (String.IsNullOrEmpty(nameBox.Text))

            {

                errorProvider1.SetError(nameBox, "Не указано имя!");

            }

            else if (nameBox.Text.Length < 4)

            {

                errorProvider1.SetError(nameBox, "Слишком короткое имя!");

            }

            else

            {

                errorProvider1.Clear();

            }

        }

        private void ageBox\_Validating(object sender, CancelEventArgs e)

        {

            int age = 0;

            if (String.IsNullOrEmpty(ageBox.Text))

            {

                errorProvider1.SetError(ageBox, "Не указан возраст!");

            }

            else if (!Int32.TryParse(ageBox.Text, out age))

            {

                errorProvider1.SetError(ageBox, "Некорретный возраст!");

            }

            else

            {

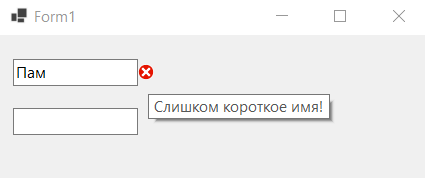
                errorProvider1.Clear();

            }

        }

    }

}



**GroupBox**

GroupBox представляет собой специальный контейнер, который ограничен от остальной формы границей. Он имеет заголовок, который устанавливается через свойство Text. Чтобы сделать GroupBox без заголовка, в качестве значения свойства Text просто устанавливается пустая строка.

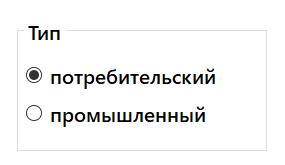
Нередко этот элемент используется для группирования переключателей - элементов RadioButton, так как позволяет разграничить их группы.

**Radiobutton**

RadioButton или переключатель. Переключатели располагаются группами, и включение одного переключателя означает отключение всех остальных.

Чтобы установить у переключателя включенное состояние, надо присвоить его свойству Checked значение true.

Для создания группы переключателей, из которых можно бы было выбирать, надо поместить несколько переключателей в какой-нибудь контейнер, например, в элементы GroupBox или Panel. Переключатели. находящиеся в разных контейнерах, будут относиться к разным группам:



Мы можем перехватывать переключение переключателей в группе, обрабатывая событие *CheckedChanged*. Связав каждый переключатель группы с одним обработчиком данного события, мы сможем получить тот переключатель, который в данный момент выбран:

private void radioButton\_CheckedChanged(object sender, EventArgs e)

{

    // приводим отправителя к элементу типа RadioButton

    RadioButton radioButton = (RadioButton)sender;

    if (radioButton.Checked)

    {

        MessageBox.Show("Вы выбрали " + radioButton.Text);

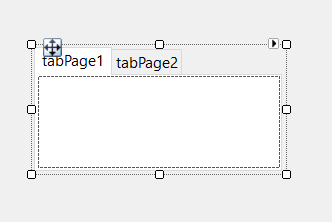
    }

}

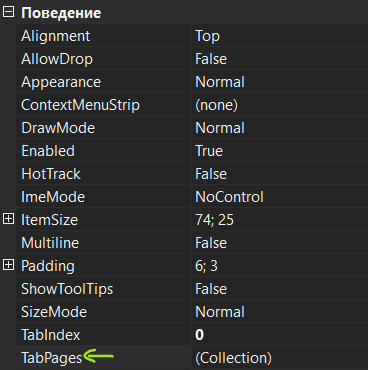
**Панель вкладок TabControl**

Элемент TabControl позволяет создать элемент управления с несколькими вкладками. И каждая вкладка будет хранить некоторый набор других элементов управления, как кнопки, текстовые поля и др. Каждая вкладка представлена классом TabPage.

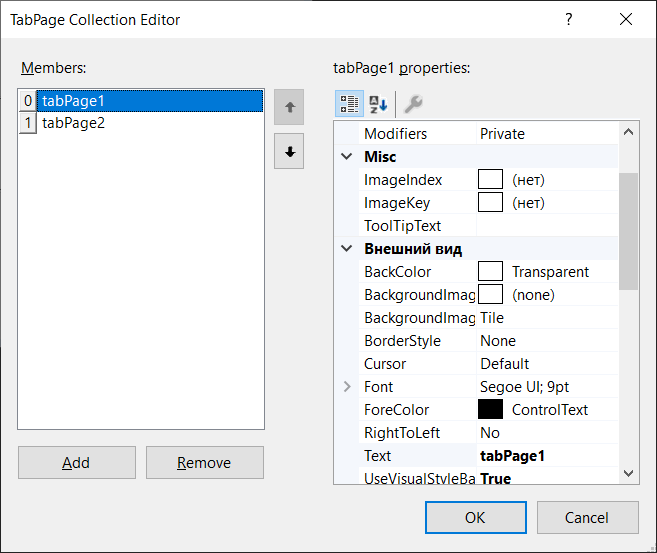
Чтобы настроить вкладки элемента TabControl используем свойство TabPages. При переносе элемента TabControl с панели инструментов на форму по умолчанию создаются две вкладки - tabPage1 и tabPage2.



Изменим их отображение с помощью свойства *TabPages*:



Нам откроется окно редактирования/добавления и удаления вкладок:

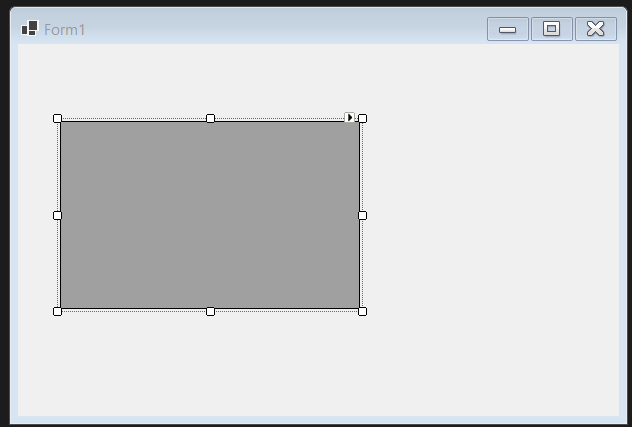


Каждая вкладка представляет своего рода панель, на которую мы можем добавить другие элементы управления, а также заголовок, с помощью которого мы можем переключаться по вкладкам. Текст заголовка задается с помощью свойства Text.

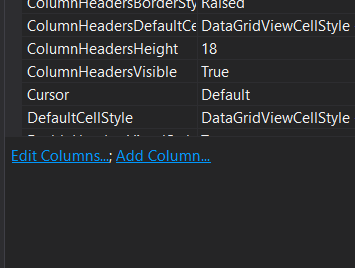
**DataGridView**

Данный элемент управления позволяет организовывать данные в виде таблицы. Данные могут быть получены из базы данных, коллекции, внутренних переменных — массивов или других объектов программы.

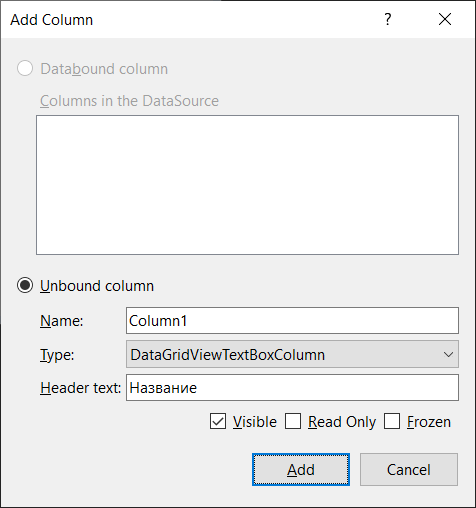
И так создадим новый проект WinForms и добавим на форму элемент DataGridView.



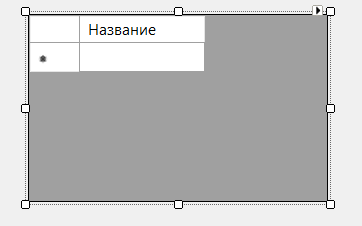
В окне свойств нажмем Add columns для добавления колонок:



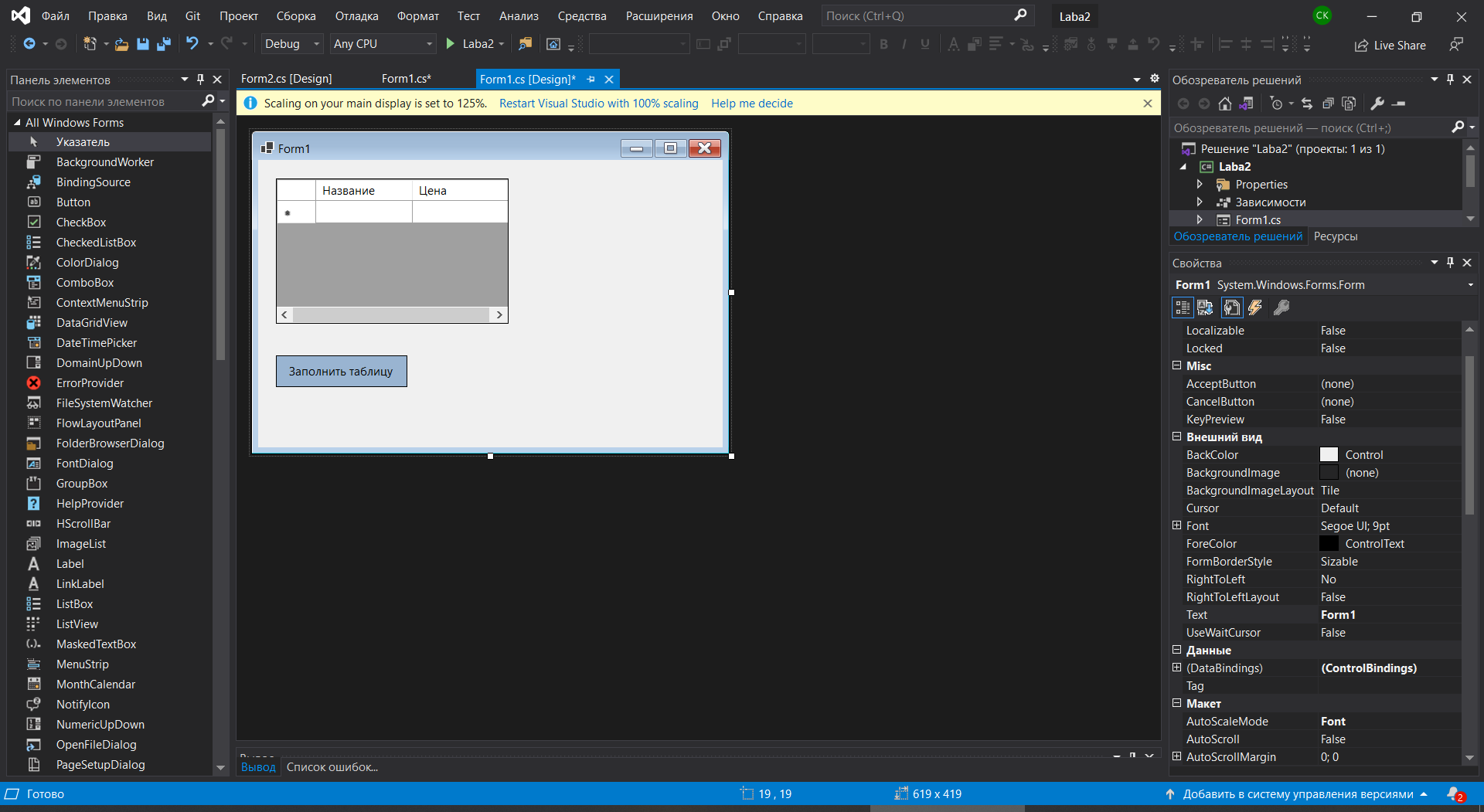
В открывшемся окне названия столбцов вводим в поле Header Text и нажимаем Add:



Итог:



Добавим на форму кнопку, при нажатии которой ясейки таблицы будут заполняться:



Код обработки события:

using System;

using System.Windows.Forms;

namespace Laba2

{

    public partial class Form1 : Form

    {

        public Form1()

        {

            InitializeComponent();

}

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

{

string name = "Orange";

int price = 2;

dataGridView1.Rows.Add(name, price);

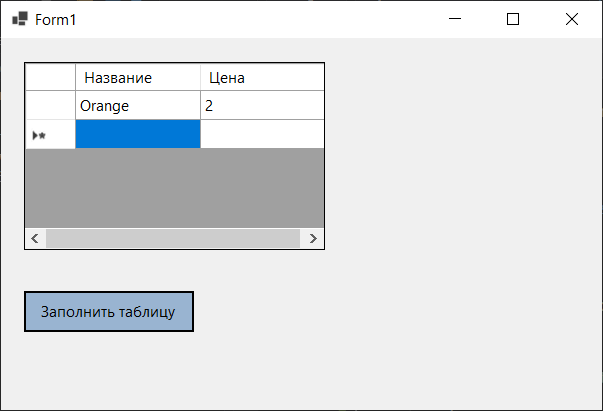
}

}

}

Свойство Rows элемента dataGridView может быть использовано для ручного заполнения элемента управления, как в данном случае.

После нажатия на кнопку:



**Управление одной формой из другой**

Иногда бывает нужно обратиться к элементам какой-то формы из другой в процессе выполнения программы. Например, есть форма Form1, из нее мы открываем другую Form2.

**Открытие формы Form2 при нажатии на кнопку в форме Form1**

namespace WindowsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

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

{

Form2 frm = new Form2();

frm.ShowDialog();

}

}

}

**Доступ ко всей родительской форме**

Чтобы осуществить это, нужно внести изменения в нескольких файлах, но зато при этом получим доступ ко всем элементам родительской формы и не нужно передавать ссылку на каждый элемент.

**Шаг 1.** В файле Program.cs создаем публичную переменную f1.

namespace WindowsApplication1

{

    static class Program

    {

        public static Form1 f1; // переменная, которая будет содержать ссылку на форму Form1

        [STAThread]

        static void Main()

        {

            Application.EnableVisualStyles();

            Application.SetCompatibleTextRenderingDefault(false);

            Application.Run(new Form1());

        }

    }

}

**Шаг 2**. Открываем Form1.Designer.cs и в нем у элементов, к которым нужно будет обратиться из другой формы, меняем private на public. Например, сделаем доступной для изменений кнопку button1 на форме Form1.

public System.Windows.Forms.Button button1; // заменили private на public

**Шаг 3**. При создании формы Form1 присваиваем переменной f1 ссылку на эту форму

namespace WindowsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

Program.f1 = this; // теперь f1 будет ссылкой на форму Form1

InitializeComponent();

}

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

{

Form2 frm = new Form2();

frm.ShowDialog();

}

}

}

**Шаг 4**. Теперь из абсолютно любой формы или из любого класса можно обратиться к элементу button1 находящемуся на Form1 так: Program.f1.button1. Например, пусть кнопка в Form2 поменяет текст кнопки на Form1:

namespace WindowsApplication1

{

public partial class Form2 : Form

{

public Form2()

{

InitializeComponent();

}

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

{

Program.f1.button1.Text = "test"; // Меняем текст на кнопке формы Form1

}

}

}

**Ответы на вопросы**

***1. Какое основное назначение технологии Windows Forms, ее особенности,***

***преимущества и недостатки?***

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

Windows Forms позволяет разрабатывать интеллектуальные клиенты. *Интеллектуальный клиент* — это приложение с полнофункциональным графическим интерфейсом, простое в развертывании и обновлении, способное работать при наличии или отсутствии подключения к Интернету и использующее более безопасный доступ к ресурсам на локальном компьютере по сравнению с традиционными приложениями Windows.

**Windows Forms** — это технология интеллектуальных клиентов для NET Framework. Она представляет собой набор управляемых библиотек, упрощающих выполнение стандартных задач, таких как чтение из файловой системы и запись в нее. С помощью такой среды разработки, как Visual Studio, можно создавать интеллектуальные клиентские приложения Windows Forms, которые отображают информацию, запрашивают ввод от пользователей и обмениваются данными с удаленными компьютерами по сети.

В Windows Forms **форма** — это визуальная поверхность, на которой выводится информация для пользователя. Обычно приложение Windows Forms строится путем помещения элементов управления на форму и написания кода для реагирования на действия пользователя, такие как щелчки мыши или нажатия клавиш. **Элемент управления** — это отдельный элемент пользовательского интерфейса, предназначенный для отображения или ввода данных.

При выполнении пользователем какого-либо действия с формой или одним из ее элементов управления создается событие. Приложение реагирует на эти события с помощью кода и обрабатывает события при их возникновении.

Наряду с ручным способом конструирования форм оболочка Visual Studio (и ряд других современных IDE) содержит и визуальный редактор диалоговых окон (Designer - **конструктор форм**), который обеспечивает условия комфортного программирования на этапе разработки графического интерфейса пользователя (GUI) оконных приложений. Конструктор форм очень облегчает эту работу, сопровождая действия программиста генерацией соответствующего кода в исходный текст приложения.

***2. Зачем используется класс Form? Назовите основные методы, свойства и***

***события данного класса.***

Пространство имен System.Windows.Forms содержит сотни типов, которые позволяют создавать высокоразвитый GUI оконных приложений. Их можно разделить на следующие большие категории:

▪ Базовая инфраструктура (классы Form, Application)

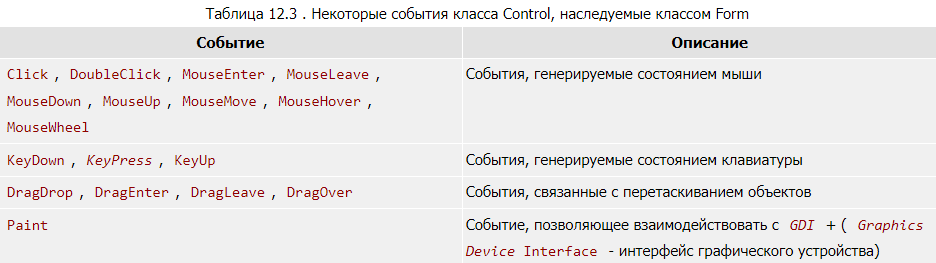
▪ Элементы управления, позволяющие пользователю взаимодействовать с приложением в интерактивном режиме (производные от класса Control )

▪ Невизуальные компоненты, дополняющие элементы управления своей функциональностью (Timer, ToolTip)

▪ Диалоговые окна общего вида (CommonDialog, ColorDialog, FolderBrowserDialog, FontDialog, OpenFileDialog, SaveFileDialog, PageSetupDialog, PrintDialog, PrintPreviewDialog)

**Функциональные возможности класса Form, наследуемые от класса Control**

Класс **Control** определяет общее поведение элементов управления и самой формы, так как является их базовым классом. Он содержит множество свойств, методов и событий, позволяющих реагировать на изменение состояния мыши, клавиатуры, действия выделения и перетаскивания объектов, другие действия.



**Класс Form**

Класс Form представляет окно, которое составляет пользовательский интерфейс приложения.

Форма — это представление любого окна, отображаемого в вашем приложении. Класс Form можно использовать для создания стандартных, инструментальных, окон без полей и плавающих окон. Вы также можете использовать класс Form для создания модальных окон, таких как диалоговое окно.

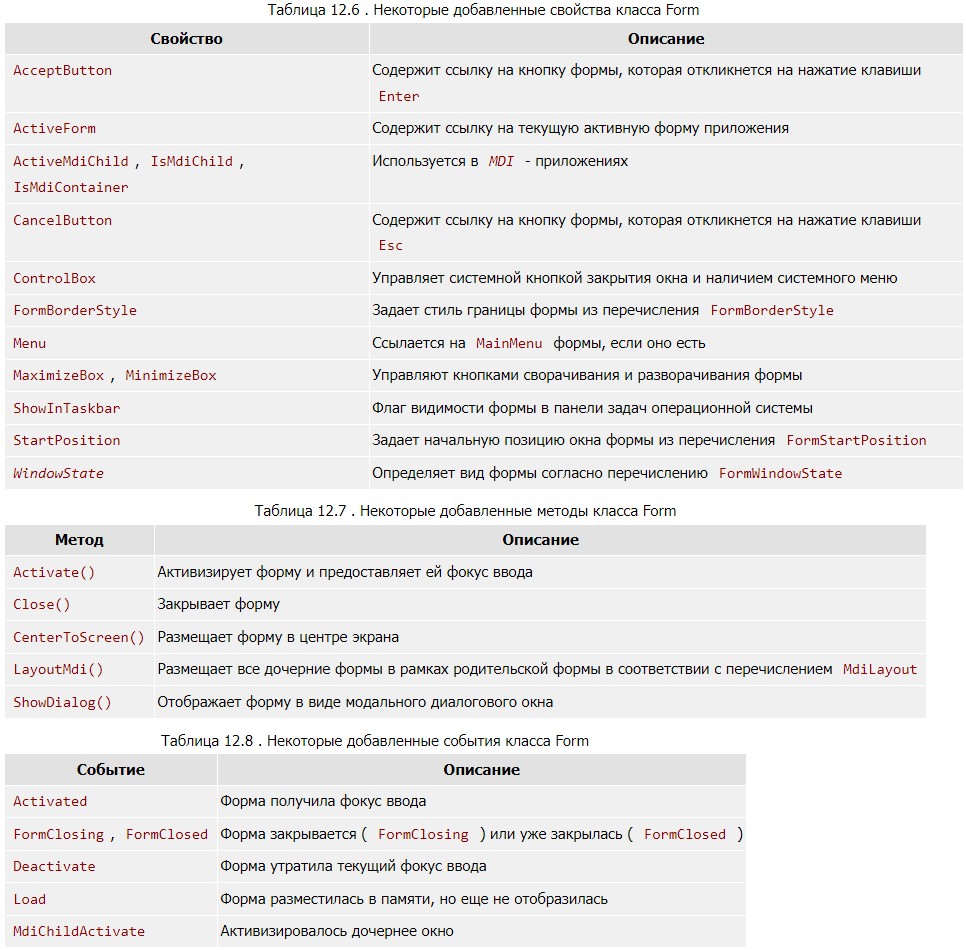
Используя свойства, доступные в классе Form , вы можете определить внешний вид, размер, цвет и функции управления окном создаваемого окна или диалогового окна. Свойство Text позволяет указать заголовок окна в строке заголовка. Свойства Size и DesktopLocation позволяют определить размер и положение окна при его отображении. Вы можете использовать свойство цвета ForeColor , чтобы изменить цвет переднего плана по умолчанию для всех элементов управления, размещенных в форме. FormBorderStyle , MinimizeBox и MaximizeBox \_свойства позволяют вам контролировать, можно ли свернуть, развернуть или изменить размер формы во время выполнения.

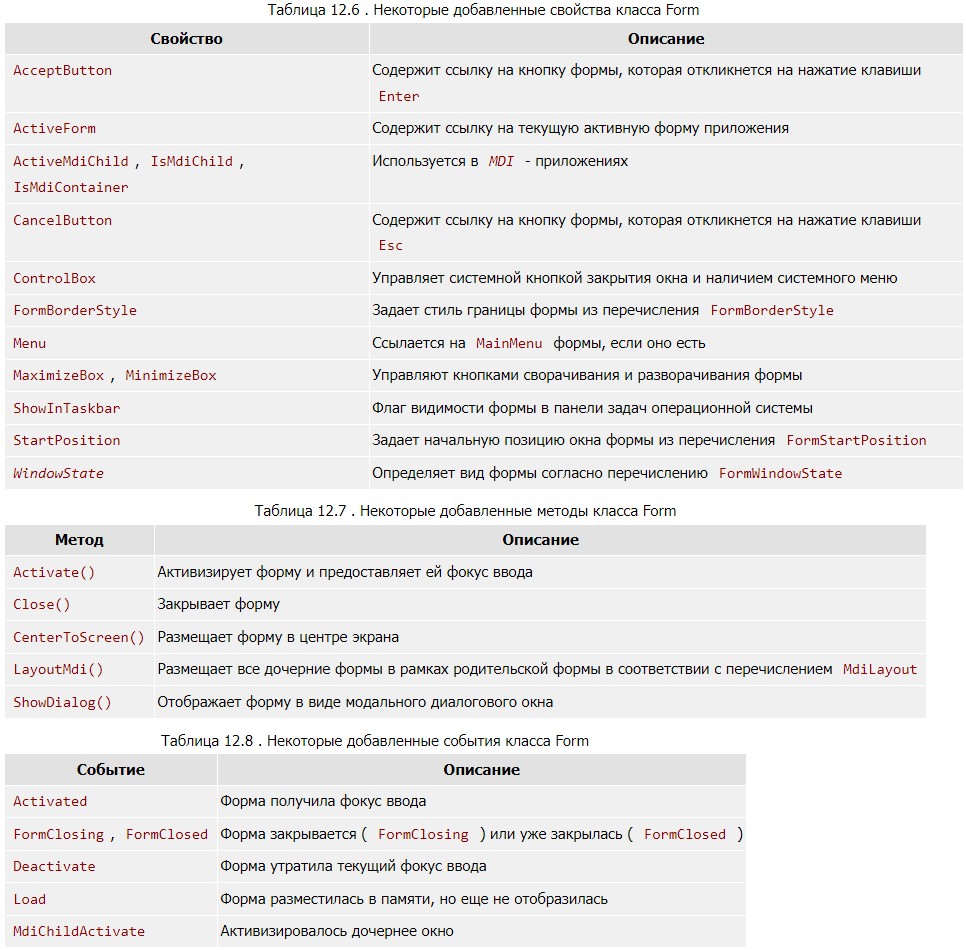
В дополнение к свойствам вы можете использовать методы класса для управления формой. Например, вы можете использовать метод ShowDialog для отображения формы в виде модального диалогового окна. Вы можете использовать метод SetDesktopLocation для размещения формы на рабочем столе.

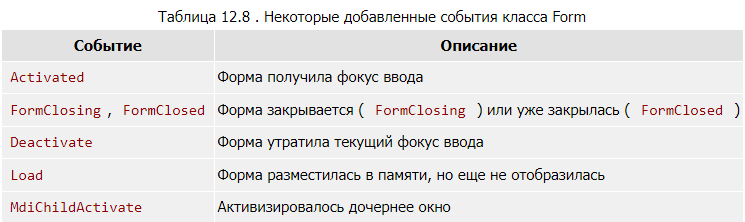
События класса Form позволяют реагировать на действия, выполняемые над формой. Вы можете использовать событие Activated для выполнения таких операций, как обновление данных, отображаемых в элементах управления формы, когда форма активирована.

**Собственные функциональные возможности класса Form**

Класс Form добавляет к унаследованным свои свойства, методы и события. Рассмотрим некоторые из них.

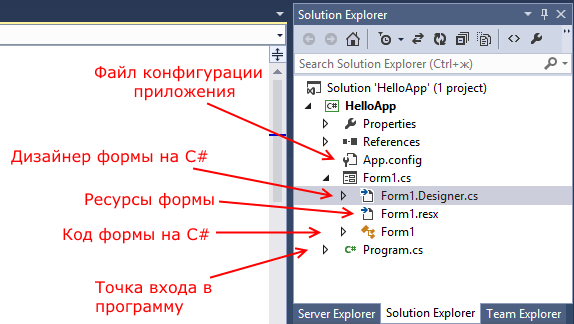






**3. Поясните структуру проекта и назначение всех файлов?**

Если мы запустим приложение, то нам отобразится одна пустая форма. Однако даже такой простой проект с пустой формой имеет несколько компонентов:



Несмотря на то, что мы видим только форму, но стартовой точкой входа в графическое приложение является класс Program, расположенный в файле **Program.cs**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Laba2

{

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.SetHighDpiMode(HighDpiMode.SystemAware);

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

}

Сначала программой запускается данный класс, затем с помощью выражения Application.Run(new Form1()) он запускает форму Form1. Если вдруг мы захотим изменить стартовую форму в приложении на какую-нибудь другую, то нам надо изменить в этом выражении Form1 на соответствующий класс формы.

Сама форма сложна по содержанию. Она делится на ряд компонентов. Так, в структуре проекта есть файл **Form1.Designer.cs**, который выглядит примерно так:

namespace Laba2

{

partial class Form1

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

this.SuspendLayout();

//

// Form1

//

this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 20F);

this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

this.ClientSize = new System.Drawing.Size(405, 450);

this.Name = "Form1";

this.Text = "Form1";

this.ResumeLayout(false);

}

#endregion

}

}

Здесь объявляется частичный класс формы Form1, которая имеет два метода: Dispose(), который выполняет роль деструктора объекта, и InitializeComponent(), который устанавливает начальные значения свойств формы.

При добавлении элементов управления, например, кнопок, их описание также добавляется в этот файл.

Но на практике мы редко будем сталкиваться с этим классом, так как они выполняет в основном дизайнерские функции - установка свойств объектов, установка переменных.

Еще один файл - **Form1.resx** - хранит ресурсы формы. Как правило, ресурсы используются для создания однообразных форм сразу для нескольких языковых культур.

И более важный файл - **Form1.cs**, который в структуре проекта называется просто Form1, содержит код или программную логику формы:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Laba2

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

}

}

По умолчанию здесь есть только конструктор формы, в котором просто вызывается метод InitializeComponent(), объявленный в файле дизайнера Form1.Designer.cs. Именно с этим файлом мы и будем больше работать.

***4. Зачем нужен атрибут STAThreadAttribute?***

Этот атрибут указывает исполняющей среде, что приложение использует однопоточную COM-модель (Component Object Model «объектная модель компонентов»). Этот атрибут не обязателен, т.е. можно смело его выбросить, если не предполагается использовать .NET-приложение как COM-сервер.

***5. Как в вашем проекте используются события и делегаты?***

***6. Объясните схему работы цепочек делегатов.***

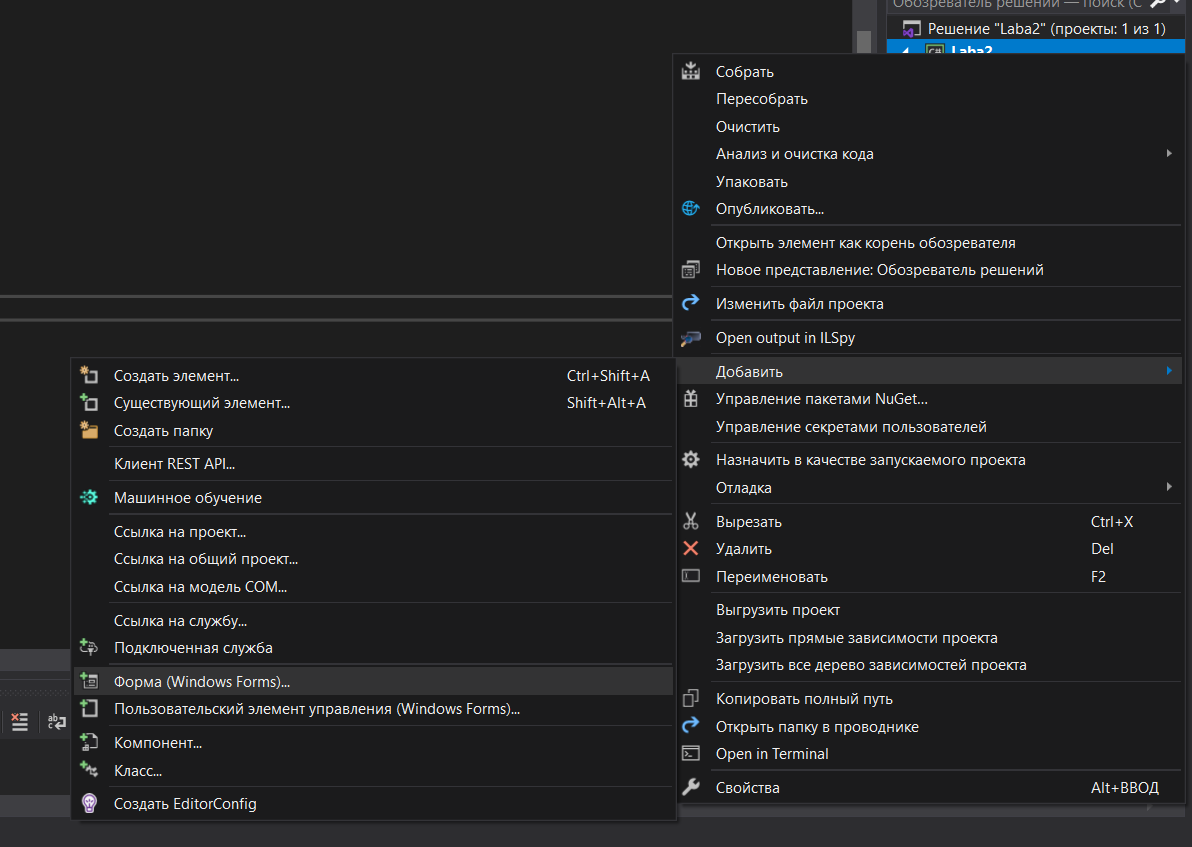
***7. Объясните механизм подписки и отмены подписки на события.***

***8. Как создать вторую форму и передать туда данные? Есть ли другие***

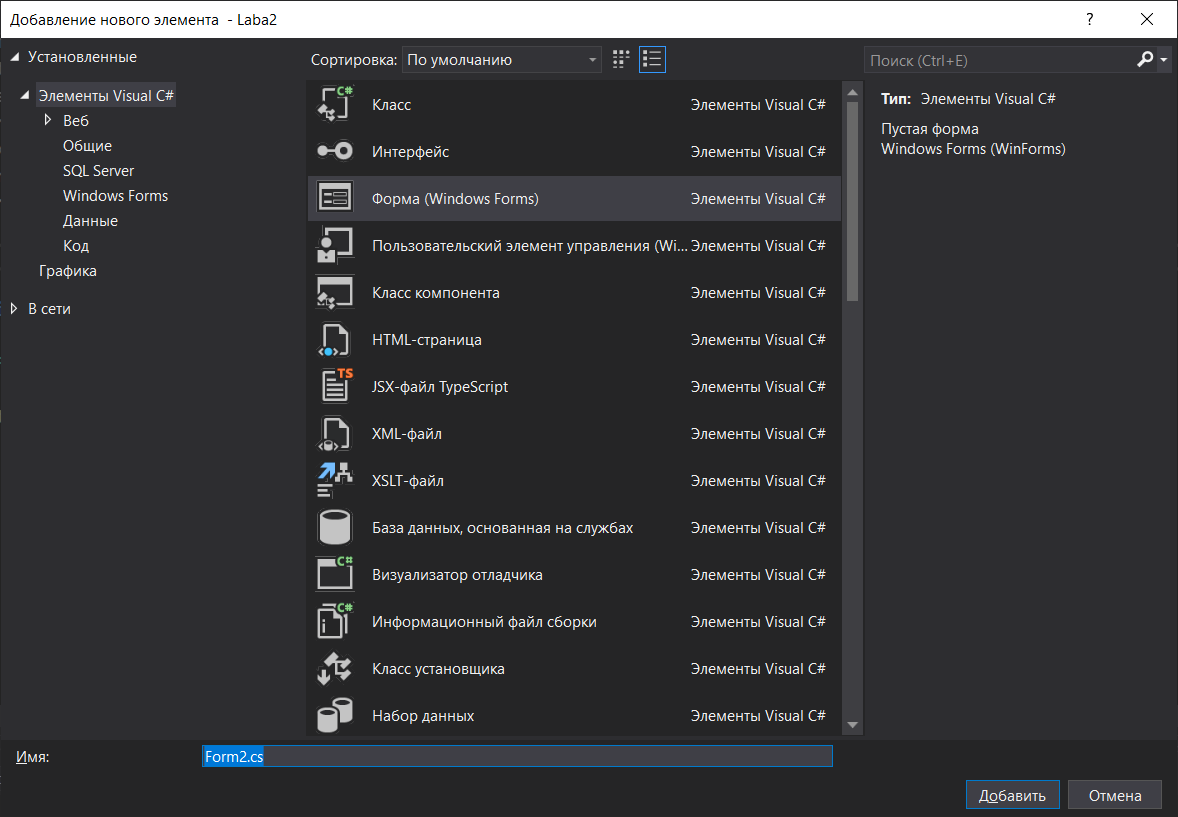
***способы?***

**Добавление форм. Взаимодействие между формами**

Чтобы добавить еще одну форму в проект, нажмем на имя проекта в окне Solution Explorer (Обозреватель решений) правой кнопкой мыши и выберем Add (Добавить)->Windows Forms (Форма)...



Дадим новой форме какое-нибудь имя, например, Form2.cs:



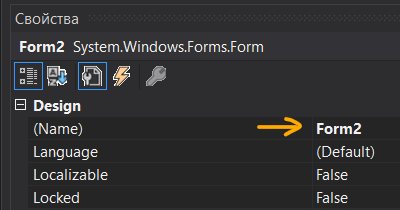
Итак, у нас в проект была добавлена вторая форма. Теперь попробуем осуществить взаимодействие между двумя формами. Допустим, первая форма по нажатию на кнопку будет вызывать вторую форму. Во-первых, добавим на первую форму Form1 кнопку и двойным щелчком по кнопке перейдем в файл кода. Итак, мы попадем в обработчик события нажатия кнопки, который создается по умолчанию после двойного щелчка по кнопке:

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

{

}

Теперь добавим в него код вызова второй формы. У нас вторая форма называется Form2.



Поэтому сначала мы создаем объект данного класса, а потом для его отображения на экране вызываем метод Show:

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

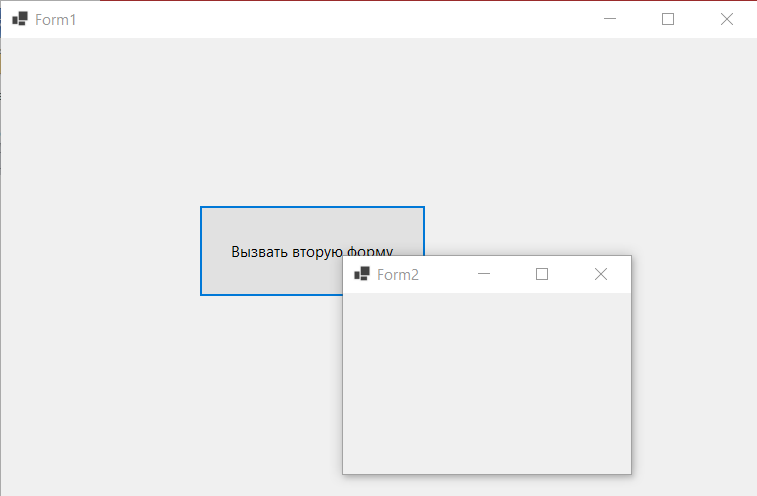
{

Form2 newForm = new Form2();

newForm.Show();

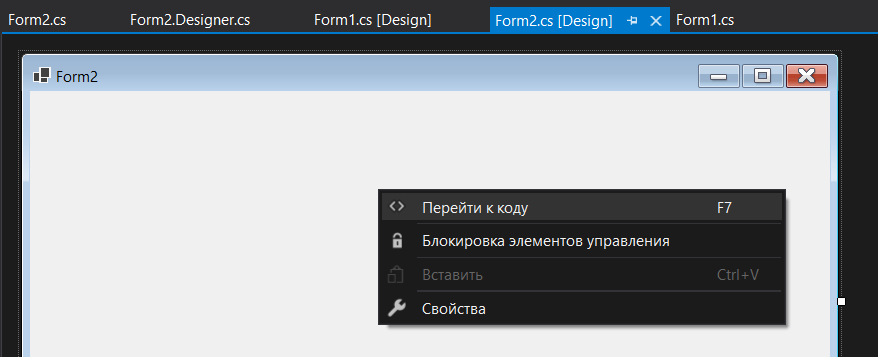
}

Результат (при нажатии на кнопку первой формы вызывается вторая):

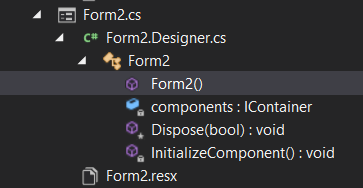
******

Теперь сделаем наоборот - чтобы вторая форма воздействовала на первую. 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.

Итак, перейдем ко второй форме и перейдем к ее коду - нажмем правой кнопкой мыши на форму и выберем View Code (Перейти к коду).



Или в обозревателе решений будем разворачивать файлы проекта, пока не найдем нужный:



Пока он пустой и содержит только конструктор. Поскольку C# поддерживает перегрузку методов, то мы можем создать несколько методов и конструкторов с разными параметрами и в зависимости от ситуации вызывать один из них. Итак, изменим файл кода второй формы на следующий:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

namespace Laba2

{

public partial class Form2 : Form

{

public Form2()

{

InitializeComponent();

}

public Form2(Form1 f)

{

InitializeComponent();

f.BackColor = Color.Yellow;

}

}

}

Мы только добавили здесь новый конструктор public Form2(Form1 f), в котором мы получаем первую форму и устанавливаем ее фон в желтый цвет. Теперь перейдем к коду первой формы, где мы вызывали вторую форму и изменим его на следующий:

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

{

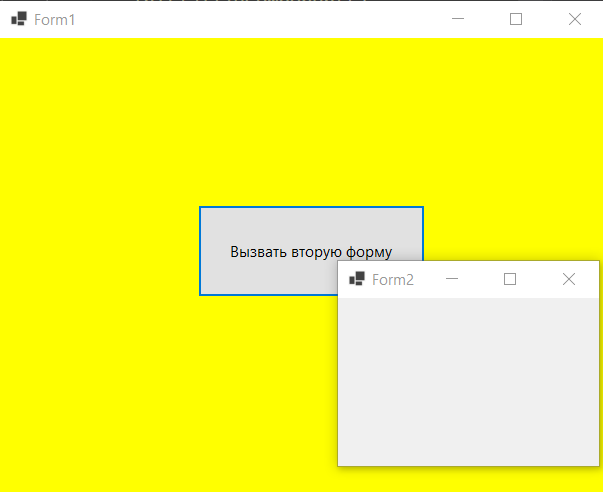
Form2 newForm = new Form2(this);

newForm.Show();

}

Поскольку в данном случае ключевое слово this представляет ссылку на текущий объект - объект Form1, то при создании второй формы она будет получать ее (ссылку) и через нее управлять первой формой.

Теперь после нажатия на кнопку у нас будет создана вторая форма, которая сразу изменит цвет первой формы.



Мы можем также создавать объекты и текущей формы:

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

{

Form1 newForm1 = new Form1();

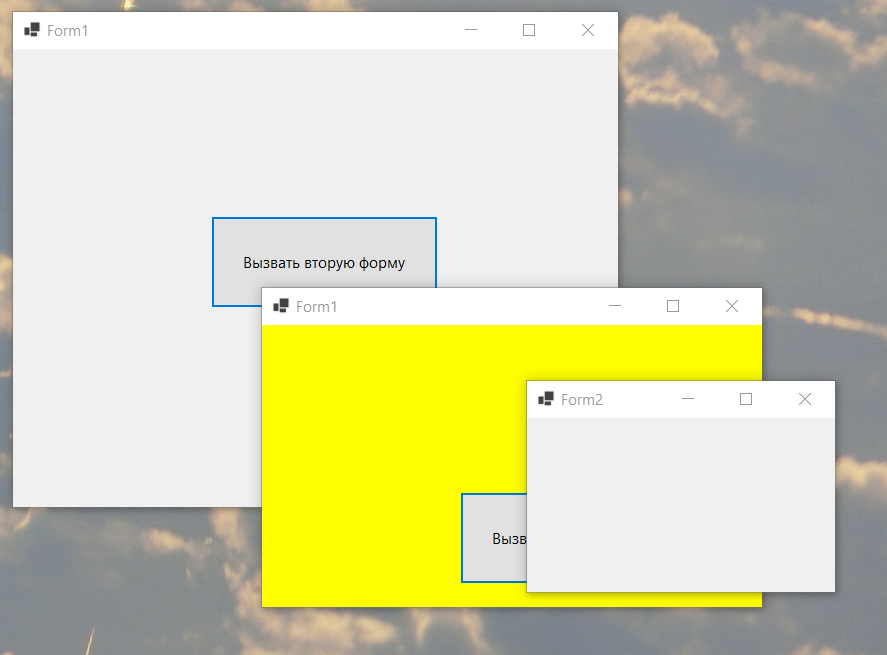
newForm1.Show();

Form2 newForm2 = new Form2(newForm1);

newForm2.Show();

}

Первая форма порождает тоже форму номер один, а так же форму два, код которой меняет цвет новосозданной формы номер один:



При работе с несколькими формами надо учитывать, что одна из них является главной - которая запускается первой в файле Program.cs. Если у нас одновременно открыта куча форм, то при закрытии главной закрывается все приложение и вместе с ним все остальные формы.

***9. Как во время выполнения приложения добавить/удалить элемент***

***управления?***

