**Windows Forms**

**Элементы управления**

**Последнее обновление: 31.10.2015**

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. 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

**Кнопка**

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

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

|  |  |
| --- | --- |
| 1  2  3  4 | private void button1\_Click(object sender, EventArgs e)  {      MessageBox.Show("Hello World");  } |

**Оформление кнопки**

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

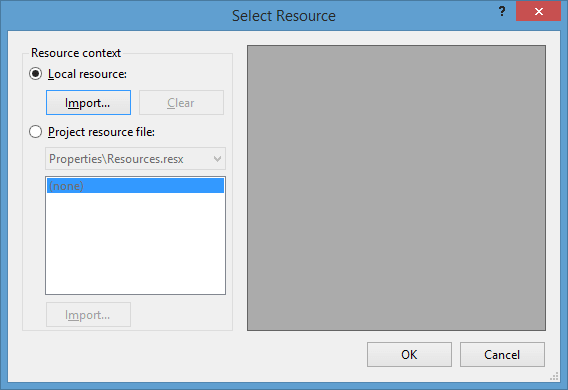
* **Flat** - Кнопка имеет плоский вид
* **Popup** - Кнопка приобретает объемный вид при наведении на нее указателя, в иных случаях она имеет плоский вид
* **Standard** - Кнопка имеет объемный вид (используется по умолчанию)
* **System** - Вид кнопки зависит от операционной системы

**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 кнопке. Для этого надо использовать свойство**TextImageRelation**. Оно приобретает следующие значения:

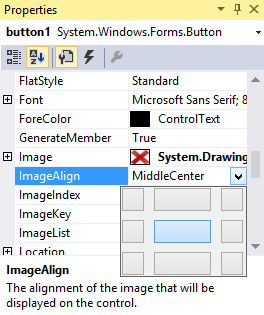
* **Overlay**: текст накладывается на изображение
* **ImageAboveText**: изображение располагается над текстом
* **TextAboveImage**: текст располагается над изображением
* **ImageBeforeText**: изображение располагается перед текстом
* **TextBeforeImage**: текст располагается перед изображением

Например, установим для кнопки изображение. Для этого выберем кнопку и в окне Свойств нажмем на поле Image (не путать с BackgroundImage). Нам откроется диалоговое окно установи изображения:



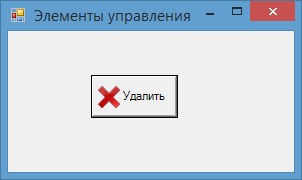
В этом окне выберем опцию Local Resource и нажмем на кнопку Import, после чего нам откроется диалоговое окно для выбора файла изображения.

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. Оставим здесь значение по умолчанию - MiddleCenter, то есть позиционирование по центру.

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



**Клавиши быстрого доступа**

При работе с формами при использовании клавиатуры очень удобно пользоваться клавишами быстрого доступа. При нажатии на клавиатуре комбинации клавиш At+некоторый символ, будет вызываться определенная кнопка. Например, зададим для некоторой кнопки свойство Text равное &Аватар. Первый знак - амперсанд - определяет ту букву, которая будет подчеркнута. В данном случае надпись будет выглядеть как Аватар. И теперь чтобы вызвать событие Click, нам достаточно нажать на комбинацию клавиш Alt+А.

**Кнопки по умолчанию**

Форма, на которой размещаются все элементы управления, имеет свойства, позволяющие назначать кнопку по умолчанию и кнопку отмены.

Так, свойство формы AcceptButton позволяет назначать кнопку по умолчанию, которая будет срабатывать по нажатию на клавишу Enter.

Аналогично работает свойство формы CancelButton, которое назначает кнопку отмены. Назначив такую кнопку, мы можем вызвать ее нажатие, нажав на клавишу Esc.

**Метки и ссылки**

**Последнее обновление: 31.10.2015**

**Label**

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

**LinkLabel**

Особый тип меток представляют элементы LinkLabel, которые предназначены для вывода ссылок, которые аналогичны ссылкам, размещенным на стандартных веб-станиц.

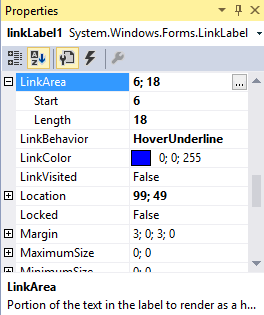
Также, как и с обычными ссылками на веб-страницах, мы можем по отношению к данному элементу определить три цвета:

* Свойство **ActiveLinkColor** задает цвет ссылки при нажатии
* Свойство **LinkColor** задает цвет ссылки до нажатия, по которой еще не было переходов
* Свойство **VisitedLinkColor** задает цвет ссылки, по которой уже были переходы

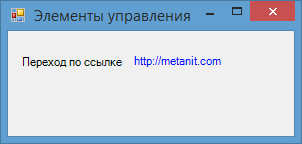
Кроме цвета ссылки для данного элемента мы можем задать свойство **LinkBehavior**, которое управляет поведением ссылки. Это свойство принимает четыре возможных значения:

* **SystemDefault**: для ссылки устанавливаются системные настройки
* **AlwaysUnderline**: ссылка всегда подчеркивается
* **HoverUnderline**: ссылка подчеркивается только при наведении на нее курсора мыши
* **NeverUnderline**: ссылка никогда не подчеркивается

По умолчанию весь текст на данном элементе считается ссылкой. Однако с помощью свойства **LinkArea** мы можем изменить область ссылки. Например, мы не хотим включать в ссылку первые шесть символов. Для этого задаем подсвойство Start:



Чтобы выполнить переход по ссылке по нажатию на нее, надо дополнительно написать код. Данный код должен обрабатывать событие LinkClicked, которое есть у элемента LinkLabel. Например, пусть у нас на форме есть элемент ссылки называется linkLabel1 и который содержит некоторую ссылку:



Чтобы перейти по ссылке, зададим обработчик LinkClicked:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();          // задаем обработчик события          linkLabel1.LinkClicked += linkLabel1\_LinkClicked;      }        private void linkLabel1\_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)      {          System.Diagnostics.Process.Start("[http://metanit.com](http://metanit.com/)");      }  } |

Метод System.Diagnostics.Process.Start() откроет данную ссылку в веб-браузере, который установлен в системе браузером по умолчанию.

**Текстовое поле TextBox**

**Последнее обновление: 31.10.2015**

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, все избыточные символы, которые выходят за границы поля, будут переноситься на новую строку.

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

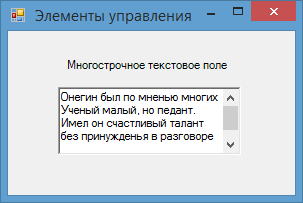
* **None**: без прокруток (по умолчанию)
* **Horizontal**: создает горизонтальную прокрутку при длине строки, превышающей ширину текстового поля
* **Vertical**: создает вертикальную прокрутку, если строки не помещаются в текстовом поле
* **Both**: создает вертикальную и горизонтальную прокрутку

**Автозаполнение текстового поля**

Элемент TextBox обладает достаточными возможностями для создания автозаполняемого поля. Для этого нам надо привязать свойство **AutoCompleteCustomSource** элемента TextBox к некоторой коллекции, из которой берутся данные для заполнения поля.

Итак, добавим на форму текстовое поле и пропишем в код события загрузки следующие строки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();          AutoCompleteStringCollection source = new AutoCompleteStringCollection()          {              "Кузнецов",              "Иванов",              "Петров",              "Кустов"          };          textBox1.AutoCompleteCustomSource = source;          textBox1.AutoCompleteMode = AutoCompleteMode.SuggestAppend;          textBox1.AutoCompleteSource = AutoCompleteSource.CustomSource;      }  } |



Режим автодополнения, представленный свойством **AutoCompleteMode**, имеет несколько возможных значений:

* **None**: отсутствие автодополнения
* **Suggest**: предлагает варианты для ввода, но не дополняет
* **Append**: дополняет введенное значение до строки из списка, но не предлагает варианты для выбора
* **SuggestAppend**: одновременно и предлагает варианты для автодополнения, и дополняет введенное пользователем значение

**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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            textBox1.TextChanged += textBox1\_TextChanged;      }        private void textBox1\_TextChanged(object sender, EventArgs e)      {          label1.Text = textBox1.Text;      }  } |

## 4.7.png

## Элемент MaskedTextBox

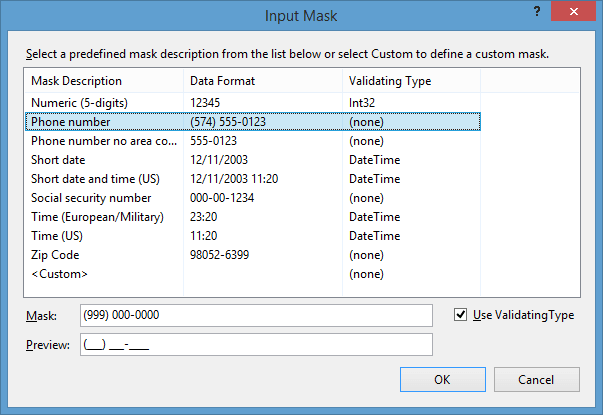
**Последнее обновление: 31.10.2015**

Элемент MaskedTextBox по сути представляет обычное текстовое поле. Однако данные 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:

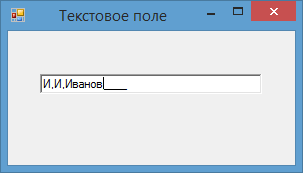
* 0: Позволяет вводить только цифры
* 9: Позволяет вводить цифры и пробелы
* #: Позволяет вводить цифры, пробелы и знаки '+' и '-'
* L: Позволяет вводить только буквенные символы
* ?: Позволяет вводить дополнительные необязательные буквенные символы
* A: Позволяет вводить буквенные и цифровые символы
* .: 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(Porperties), 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:



Данный элемент также представляет нам ряд свойств, которые можно использовать для управления вводом. Так, свойство**BeepOnError** при установке значения true подает звуковой сигнал при введении некорректного символа.

Свойство **HidePromptOnLeave** при установке в true при потери текстовым полем фокуса скрывает, указанные в PromptChar

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 asci-символы, то есть символы из диапазона A-Z и a-z.

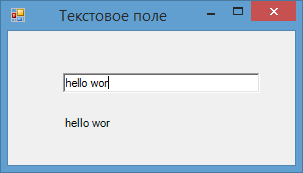
**Элементы Radiobutton и CheckBox**

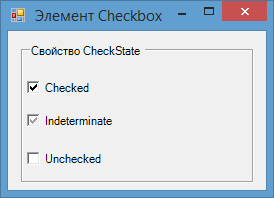
**Последнее обновление: 31.10.2015**

**CheckBox**

Элемент CheckBox или флажок предназначен для установки одного из двух значений: отмечен или не отмечен. Чтобы отметить флажок, надо установить у его свойства **Checked** значение true.

Кроме свойства Checked у элемента CheckBox имеется свойство **CheckState**, которое позволяет задать для флажка одно из трех состояний - Checked (отмечен), Indeterminate (флажок не определен - отмечен, но находится в неактивном состоянии) и Unchecked (не отмечен)





Также следует отметить свойство AutoCheck - если оно имеет значение false, то мы не можем изменять состояние флажка. По умолчанию оно имеет значение true.

При изменении состояния флажка он генерирует событие **CheckedChanged**. Обрабатывая это событие, мы можем получать измененный флажок и производить определенные действия:

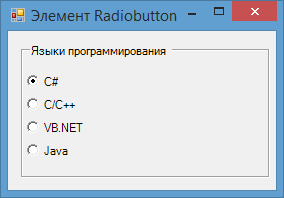
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | private void checkBox\_CheckedChanged(object sender, EventArgs e)  {      CheckBox checkBox = (CheckBox)sender; // приводим отправителя к элементу типа CheckBox      if (checkBox.Checked == true)      {          MessageBox.Show("Флажок " + checkBox.Text + "  теперь отмечен");      }      else      {          MessageBox.Show("Флажок " + checkBox.Text + "  теперь не отмечен");      }  } |

**Radiobutton**

На элемент CheckBox похож элемент 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:



Похожим образом 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:

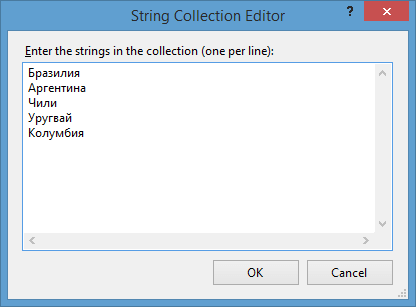
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | private void radioButton\_CheckedChanged(object sender, EventArgs e)  {      // приводим отправителя к элементу типа RadioButton      RadioButton radioButton = (RadioButton)sender;      if (radioButton.Checked)      {          MessageBox.Show("Вы выбрали " + radioButton.Text);      }  } |

## ListBox

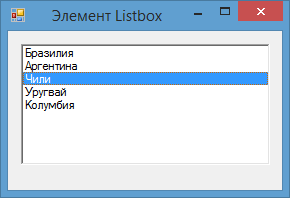
**Последнее обновление: 31.10.2015**

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 двойного щелчка на свойство 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:



### 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:

|  |  |
| --- | --- |
| 1  2 | 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:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1 | listBox1.Items.Remove("Чили"); |

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

|  |  |
| --- | --- |
| 1 | listBox1.Items.RemoveAt(1); |

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

|  |  |
| --- | --- |
| 1 | listBox1.Items.Clear(); |

#### Доступ к элементам списка

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

|  |  |
| --- | --- |
| 1 | string firstElement = listBox1.Items[0]; |

Метод Count позволяет определить количество элементов в списке:

|  |  |
| --- | --- |
| 1 | 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 програмно, 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:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | 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);      }  } |

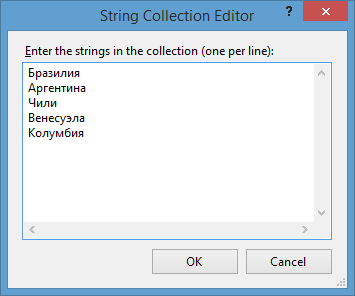
В данном случае по выбору элемента списка будет отображаться сообщение с выделенным элементом.

**Элемент ComboBox**

**Последнее обновление: 31.10.2015**

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:



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

Добавление элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 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:

|  |  |
| --- | --- |
| 1 | comboBox1.Items[0] = "Парагвай"; |

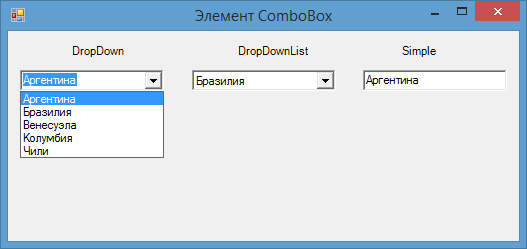
**Настройка оформления ComboBox**

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
* **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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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.

**Привязка данных в ListBox и ComboBox**

**Последнее обновление: 31.10.2015**

Кроме прямого добавления элементов в коллекцию Items компонентов ListBox и ComboBox мы также можем использовать механизм привязки данных.

Привязка данных в ListBox и ComboBox реализуется с помощью следующих свойств:

* **DataSource**: источник данных - какой-нибудь массив или коллекция объектов
* **DisplayMember**: свойство объекта, которое будет использоваться для отображения в ListBox / ComboBox
* **ValueMember**: свойство объекта, которое будет использоваться в качестве его значения

Рассмотрим пример.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            List<Phone> phones = new List<Phone>          {              new Phone { Id=11, Name="Samsung Galaxy Ace 2", Year=2012},              new Phone { Id=12, Name="Samsung Galaxy S4", Year=2013},              new Phone { Id=13, Name="iPhone 6", Year=2014},              new Phone { Id=14, Name="Microsoft Lumia 435", Year=2015},              new Phone { Id=15, Name="Xiaomi Mi 5", Year=2015}          };            listBox1.DataSource = phones;          listBox1.DisplayMember = "Name";          listBox1.ValueMember = "Id";            listBox1.SelectedIndexChanged += listBox1\_SelectedIndexChanged;      }        void listBox1\_SelectedIndexChanged(object sender, EventArgs e)      {          // получаем id выделенного объекта          int id = (int)listBox1.SelectedValue;            // получаем весь выделенный объект          Phone phone = (Phone)listBox1.SelectedItem;          MessageBox.Show(id.ToString() + ". " + phone.Name);      }  }  class Phone  {      public int Id { get; set; }      public string Name { get; set; }      public int Year { get; set; }  } |

Итак, на форме у нас есть список ListBox с именем listBox1. В коде имеется класс Phone с тремя свойствами, объекты которого мы хотим выводить в список. В отличие от предыдущих тем эта задача сложнее, так как раньше мы выводили обычные строки, тут же у нас сложные объекты.

Для вывода используем механизм привязки. Сначала устанавливаем список телефонов в качестве источника данных:

|  |  |
| --- | --- |
| 1 | listBox1.DataSource = phones; |

Затем устанавливаем в качестве отображаемого свойства свойство Name класса Phone, а в качестве свойства значения - свойство Id:

|  |  |
| --- | --- |
| 1  2 | listBox1.DisplayMember = "Name";  listBox1.ValueMember = "Id"; |

Значение отображаемого свойства мы затем увидим в списке. Оно будет представлять каждый отдельный объект Phone.

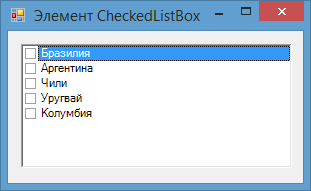
С помощью же свойства значения, которым является свойство Id, мы можем упростить работу с источником данных. В данном случае оно не играет большой роли. Но если бы мы использовали в качестве источника данных некоторый набор объектов из базы данных, то с помощью id нам было проще удалять, обновлять и взаимодействовать с базой данных.

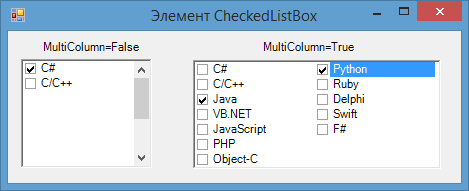
И теперь если мы выделим какой-то объект, то свойство **SelectedItem** элементы ListBox будет содержать объект Phone, у которого мы можем получить значения свойств:

|  |  |
| --- | --- |
| 1  2 | Phone phone = (Phone)listBox1.SelectedItem;  string name = phone.Name; |

А выделенное значение, то есть значение свойства Id выделенного телефона, будет находиться в свойстве **SelectedValue**.

И если мы запустим приложение, то увидим все отображаемые телефоны:

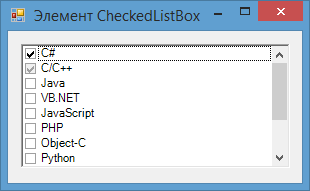




Все то же самое характерно и для элемента ComboBox. Пусть кроме ListBoxa на форме есть ComboBox:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            List<Phone> phones = new List<Phone>          {              new Phone { Id=11, Name="Samsung Galaxy Ace 2", Year=2012},              new Phone { Id=12, Name="Samsung Galaxy S4", Year=2013},              new Phone { Id=13, Name="iPhone 6", Year=2014},              new Phone { Id=14, Name="Microsoft Lumia 435", Year=2015},              new Phone { Id=15, Name="Xiaomi Mi 5", Year=2015}          };            comboBox1.DataSource = phones;          comboBox1.DisplayMember = "Name";          comboBox1.ValueMember = "Id";            comboBox1.SelectedIndexChanged += comboBox1\_SelectedIndexChanged;            listBox1.DisplayMember = "Name";          listBox1.ValueMember = "Id";      }        void comboBox1\_SelectedIndexChanged(object sender, EventArgs e)      {          Phone phone = (Phone)comboBox1.SelectedItem;          listBox1.Items.Add(phone);      }  }    class Phone  {      public int Id { get; set; }      public string Name { get; set; }      public int Year { get; set; }  } |

Здесь также для комбобокса устанавливается привязка, а также отображаемое свойство и свойство значения. Кроме того, здесь обрабатывается событие выбора элемента в комбобоксе так, чтобы выбранный элемент попадал в ListBox.



В отличие от ListBoxa ComboBox имеет три свойства для обработки выделенного объекта:

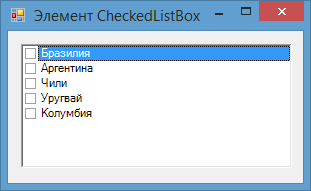
* SelectedItem: выбранный элемент
* SelectedValue: значение свойства значения, в данном случае свойство Id
* SelectedText: значение свойства отображение, в данном случае свойство Name класса Phone

**Элемент CheckedListBox**

**Последнее обновление: 31.10.2015**

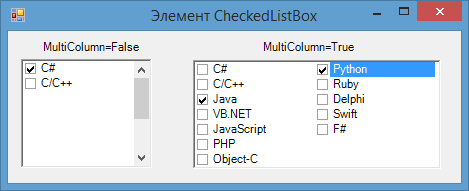
Элемент CheckedListBox представляет симбиоз компонентов ListBox и CheckBox. Для каждого элемента такого списка определено специальное поле CheckBox, которое можно отметить.

Все элементы задаются в CheckedListBox задаются в свойстве **Items**. Также, как и для элементов ListBox и ComboBox, мы можем задать набор элементов. По умолчанию для каждого добавляемого нового элемента флажок не отмечен:



Чтобы поставить отметку в checkBox рядом с элементом в списке, нам надо сначала выделить элемент и дополнительным щелчком уже установить флажок. Однако это не всегда удобно, и с помощью свойства **CheckOnClick** и установке для него значения true мы можем определить сразу выбор элемента и установку для него флажка в один клик.

Другое свойство **MultiColumn** при значении true позволяет сделать многоколоночный список, если элементы не помещаются по длине:

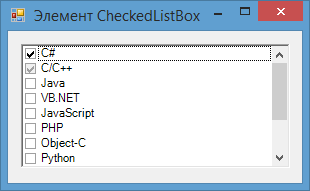


Выделенный элемент мы также можем получить с помощью свойства **SelectedItem**, а его индекс - с помощью свойства**SelectedIndex**. Но это верно только, если для свойства **SelectionMode** установлено значение One, что подразумевает выделение только одного элемента.

При установке для свойства SelectionMode значений MultiSmple и MultiExtended можно выбрать сразу несколько элементов, и тогда все выбранные элементы будут доступны в свойстве **SelectedItems**, а их индексы - в свойстве**SelectedIndeces**.

И поскольку мы можем поставить отметку не для всех выбранных элементов, то чтобы отдельно получить отмеченные элементы, у CheckedListBox имеются свойства **CheckedItems** и **CheckedIndices**.

Для добавления и удаления элементов в CheckedListBox определены все те же методы, что и в LstBox:



* Add(item): добавляет один элемент
* AddRange(array): добавляет в список массив элементов
* Insert(index, item): добавляет элемент по определенному индексу
* Remove(item): удаляет элемент
* RemoveAt(index): удаляет элемент по определенному индексу
* Clear(): полностью очищает список

**SetItemChecked и SetItemCheckState**

К особенностям элемента можно отнести методы **SetItemChecked** и **SetItemCheckState**. Метод SetItemChecked позволяет установить или сбросить отметку на одном из элементов. А метод SetItemCheckState позволяет установить флажок в одно из трех состояний: Checked (отмечено), Unchecked (неотмечено) и Indeterminate (промежуточное состояние):

|  |  |
| --- | --- |
| 1  2 | checkedListBox1.SetItemChecked(0, true);  checkedListBox1.SetItemCheckState(1, CheckState.Indeterminate); |

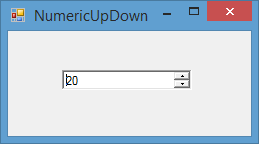
**Элементы NumericUpDown и DomainUpDown**

**Последнее обновление: 31.10.2015**

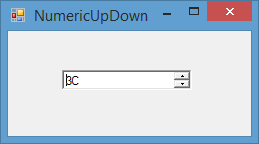
**NumericUpDown**

Элемент NumericUpDown представляет 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.



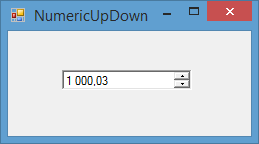
Даже если мы в коде установим обычное десятичное значение:

|  |  |
| --- | --- |
| 1 | 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:



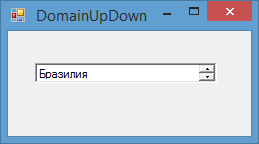
При этом надо учитывать, что если мы устанавливаем значение для свойства Value в окне свойств, то там в качестве разделителя цеой и дробной части используется запятая. Если же мы устанавливаем данное свойство в коде, тогда в качестве разделителя используется точка.

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | 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);      }  } |

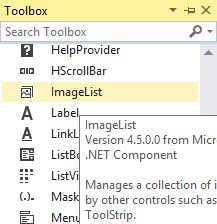
Для обработки изменения текста здесь таке, как и для элемента TextBox, можно использовать событие TextChanged, в обработчике которого мы выводим выбранный текст в сообщение.

**ImageList**

**Последнее обновление: 31.10.2015**

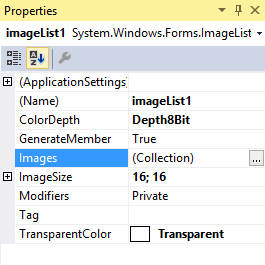
ImageList не является визуальным элементом управления, однако он представляет компонент, который используется элементами управления. Он определяет набор изображений, который могут использовать такие элементы, как ListView или TreeView.

Чтобы его добавить в проект, его также можно перенести на форму с Панели Инструментов:



Так как компонент не является визуальным элементом, то мы увидим его под формой.

Ключевым свойством ImageList является свойство **Images**, которое задает коллекцию изображений.



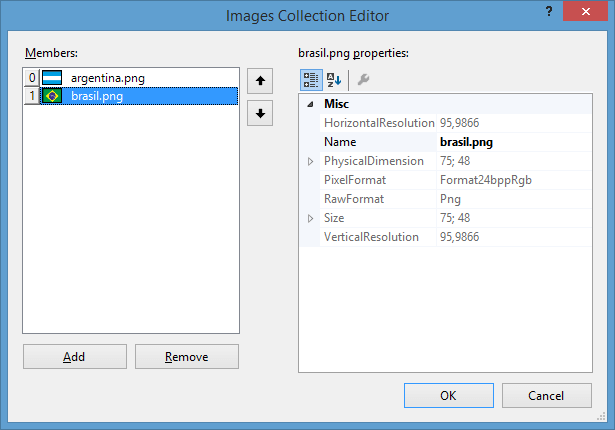
При выборе данного свойства нам откроется окно редактора изображений, в котором мы можем добавить новое изображение или удалить имеющееся.

Чтобы установить размер изображений для данного ImageList можно использовать его свойство **ImageSize**. По умолчанию ширина и высота имеют значение 16 пикселей, но мы можем установить любое другое, но не больше 256 пикселей.

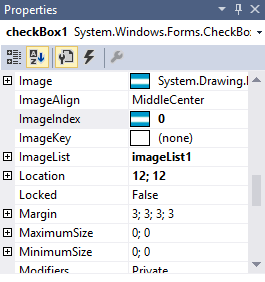
Также можно добавлять/удалять изображения из списка программно:

|  |  |
| --- | --- |
| 1  2 | imageList1.Images.Add(Image.FromFile(@"C:\Users\Eugene\Pictures\uruguay.png"));  imageList1.Images.RemoveAt(0); // удаляем первое изображение |

Чтобы делать разобраться, как использовать ImageList, добавим в него три изображения и поместим на форму три чекбокса. У каждого чекбокса уберем тест и установим свойство ImageList и укажем в свойстве ImageIndex индекс изобраения из imageList1:



И получим форму наподобие следующей:



Так как компонент не является визуальным элементом, то мы увидим его под формой.

Ключевым свойством ImageList является свойство **Images**, которое задает коллекцию изображений.

При выборе данного свойства нам окроется окно редактора изображений, в котором мы можем добавить новое изображение или удалить имеющееся.

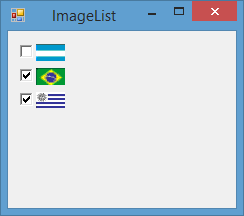
Чтобы установить размер изображений для данного ImageList можно использовать его свойство **ImageSize**. По умолчанию ширина и высота имеют значение 16 пикселей, но мы можем установить любое другое, но не больше 256 пикселей.

Также можно добавлять/удалять изображения из списка программно:

|  |  |
| --- | --- |
| 1  2 | imageList1.Images.Add(Image.FromFile(@"C:\Users\Eugene\Pictures\uruguay.png"));  imageList1.Images.RemoveAt(0); // удаляем первое изображение |

Чтобы делать разобраться, как использовать ImageList, добавим в него три изображения и поместим на форму три чекбокса. У каждого чекбокса уберем тест и установим свойство ImageList и укажем в свойстве ImageIndex индекс изобраения из imageList1:

И получим форму наподобие следующей:



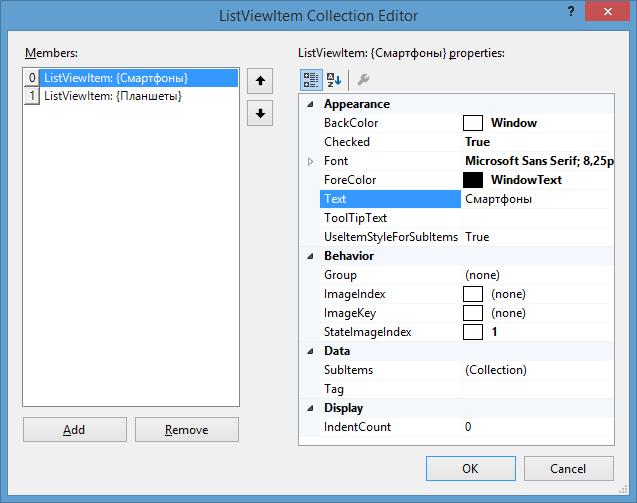
**ListView**

**Последнее обновление: 31.10.2015**

Элемент ListView представляет список, но с более расширенными возможностями, чем ListBox. В ListView можно отображать сложные данные в различных столбцах, можно задавать данным изображения и пиктограммы.

**ListViewItem**

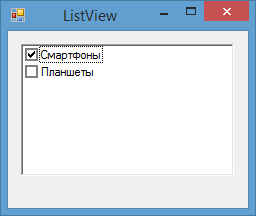
Все элементы, как и в других списковых визуальных компонентах, задаются с помощью свойства **Items**. Но в отличие от ListBox или ComboBox, если мы через панель Свойств откроем окно редактирования элементов ListView:



Каждый отдельный элемент в ListView представляет объект **ListViewItem**. В окне редактирования элементов мы также можем добавлять и удалять элементы списка. Но кроме того, здесь также мы можем выполнить дополнительную настройку элементов с помощью следующих свойств:

* **BackColor**: фоновый цвет элемента
* **Checked**: если равно true, то данный элемент будет отмечен
* **Font**: шрифт элемента
* **ForeColor**: цвет шрифта
* **Text**: текст элемента
* **ToolTipText**: текст всплывающей подсказки, устанавливаемой для элемента
* **UseItemStyleForSubItems**: если равно true, то стиль элемента будет также использоваться и для всех его подэлементов
* **Group**: задает фоновый цвет элемента
* **ImageIndex**: получает или задает индекс изображения, выводимого для данного элемента
* **ImageKey**: получает или задает индекс изображения для данного элемента
* **StateImageIndex**: получает или задает индекс изображения состояния (например установленного или снятого флажка, указывающего состояние элемента)
* **SubItems**: коллекция подэлементов для данного элемента ListViewItem
* **Tag**: тег элемента
* **IdentCount**: устанавливает отступ от границ ListViewItem до используемого им изображения

Это только те свойства, которые мы можем задать в окне редактирования элементов ListView. Но потом все добавляемые элементы мы сможем увидеть в ListView:



Чтобы добавить к элементам в ListView флажки, кроме задания свойства Checked у каждого отдельного элемента ListViewItem, надо также у свойства **CheckBoxes** у самого объекта ListView установить значение true.

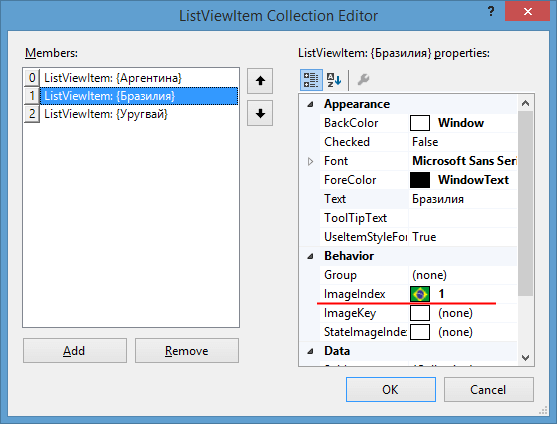
**Изображения элементов**

Для добавления элементам изображений у ListView есть несколько свойств:

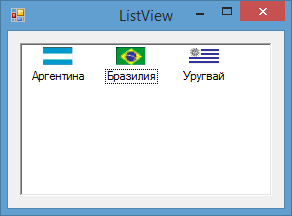
* **LargeImageList**: задает список ImageList, изображения которого будут использоваться для крупных значков
* **SmallImageList**: задает список ImageList, изображения которого будут использоваться для мелких значков
* **StateImageList**: задает список ImageList, изображения которого будут использоваться для разных состояний

Пусть у нас есть некоторый ImageList с изображениями. Зададим этот ImageList для свойств LargeImageList и SmallImageList.

Тогда при добавлении новых элементов мы можем указать индекс изображение из ImageList, которое будет использоваться элементом:



Тогда в приложении вместе с текстыми метками элементов можно будет увидеть и изображения:

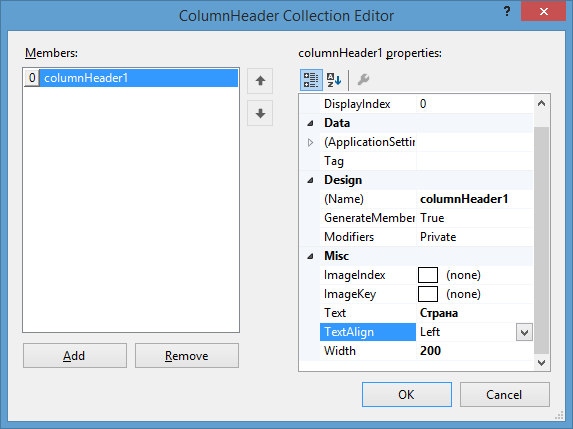


**Типы отображений**

С помощью свойства **View** у элемента ListView можно задать тип отображения, который принимает следующие значения:

* **Details**: отображение в виде таблицы
* **LargeIcon**: набор крупных значков (применяется по умолчанию)
* **List**: список
* **SmallIcon**: набор мелких значков
* **Tile**: плитка

При отображении в виде таблицы также надо задать набор столбцов в свойстве Columns у ListView:



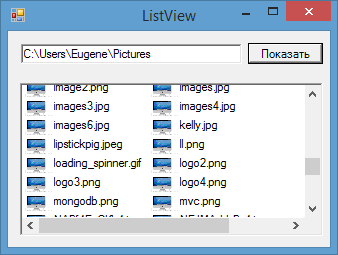
В данном случае я указал один столбец, у которого заголовок будет "Страна". Если у элементов ListViewItem были бы подэлементы, то можно было бы также задать и столбцы для подэлементов.

Кроме рассмотренных выше свойств ListView надо еще отметить некоторые. Свойство **MultiSelect** при установке в true позволяет выделять несколько строк в ListView одновременно.

Свойство **Sorting** позволяет задать режим сортировки в ListView. По умолчанию оно имеет значение None, но также можно установить сортировку по возрастанию (значение Ascending) или сортировку по убыванию (значение Descending)

**ListView. Практика**

Выполним небольшую практическую задачу: выберем все названия файлов из какой-нибудь папки в ListView.



Во-первых добавим на форму элементы TextBox (для ввода названия папки, файлы которой надо получить), Button (для запуска получения) и ListView.

Чтобы все файлы имели какое-нибудь изображение, добавим на форму ImageList с именем imageList1 и поместим в него какую-нибудь картинку.

У ListView для свойства View установим значение SmallIcon.

Все остальное сделаем в коде формы:

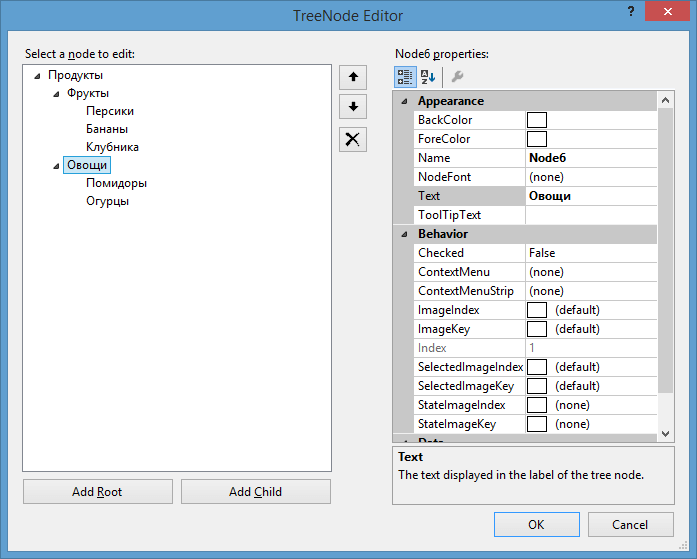
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | using System;  using System.ComponentModel;  using System.IO;  using System.Windows.Forms;    namespace HelloApp  {      public partial class Form1 : Form      {          public Form1()          {              InitializeComponent();                listView1.SmallImageList = imageList1;          }            private void button1\_Click(object sender, EventArgs e)          {              string path = textBox1.Text;              // получаем все файлы  string[] files = Directory.GetFiles(path);                    // перебор полученных файлов              foreach(string file in files)              {                  ListViewItem lvi = new ListViewItem();                  // установка названия файла                  lvi.Text = file.Remove(0, file.LastIndexOf('\\') + 1);                  lvi.ImageIndex = 0; // установка картинки для файла                  // добавляем элемент в ListView                  listView1.Items.Add(lvi);              }          }      }  } |

**TreeView**

**Последнее обновление: 31.10.2015**

TreeView представляет визуальный элемент в виде дерева. Дерево содержит узлы, которые представляют объекты **TreeNode**. Узлы могут содержать другие подузлы и могут находиться как скрытом, так и в раскрытом состоянии. Все узлы содержатся в свойстве **Nodes**.

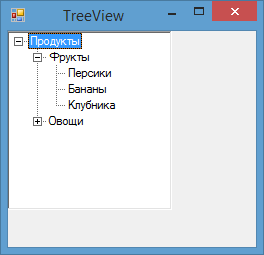
Если мы нажем в панели Свойств на свойство Nodes, то нам откроется окно редактирования узлов TreeView:



В этом окне мы можем добавить новые узлы, создать для них подузлы, удалить уже имеющиеся, настроить свойства узлов. Рассмотрим некоторые свойства, которые мы здесь может установить:

* **BackColor**: фоновый цвет узла
* **Checked**: если равно true, то данный узел будет отмечен флажком
* **NodeFont**: шрифт узла
* **ForeColor**: цвет шрифта
* **Text**: текст узла
* **ImageIndex**: получает или задает индекс изображения, выводимого для данного узла
* **ImageKey**: получает или задает индекс изображения для данного узла
* **SelectedImageKey**: получает или задает индекс изображения для данного узла в выбранном состоянии
* **SelectedImageIndex**: получает или задает индекс изображения, выводимого для данного узла в выбранном состоянии
* **StateImageIndex**: получает или задает индекс изображения состояния (например установленного или снятого флажка, указывающего состояние элемента)
* **Tag**: тег узла

И затем все добавленные узлы мы сможем увидеть в приложении на форме:



Кроме данных свойств, управляющих визуализацией, элемент TreeNode имеет еще ряд важных свойств, которые мы можем использовать к коде:

* **FirstNode**: первый дочерний узел
* **LastNode**: последний дочерний узел
* **NextNode**: возвращает следующий сестринский узел по отношению к текущему
* **NextVisibleNode**: возвращает следующий видимый узел по отношению к текущему
* **PrevNode**: возвращает предыдущий сестринский узел по отношению к текущему
* **PrevVisibleNode**: возвращает предыдущий видимый узел по отношению к текущему
* **Nodes**: возвращает коллекцию дочерних узлов
* **Parent**: возвращает родительский узел для текущего узла
* **TreeView**: возвращает объект TreeView, в котором определен текущий узел

**Программное управление узлами**

Рассмотрим программное добавление и удаление узлов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | TreeNode tovarNode = new TreeNode("Товары");  // Добавляем новый дочерний узел к tovarNode  tovarNode.Nodes.Add(new TreeNode("Смартфоны"));  // Добавляем tovarNode вместе с дочерними узлами в TreeView  treeView1.Nodes.Add(tovarNode);  // Добавляем второй очерний узел к первому узлу в TreeView  treeView1.Nodes[0].Nodes.Add(new TreeNode("Планшеты"));  // удаление у первого узла второго дочернего подузла  treeView1.Nodes[0].Nodes.RemoveAt(1);  // Удаление узла tovarNode и всех его дочерних узлов  treeView1.Nodes.Remove(tovarNode); |

**Скрытие и раскрытие узлов**

Для раскрытия узлов к объекту TreeNode применяется метод **Expand()**, а для скрытия - метод **Collapse()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // раскрытие узла  tovarNode.Expand();  // раскрытие не только узла, но и всех его дочерних подузлов  tovarNode.ExpandAll();  // скрытие узла  tovarNode.Collapse(); |

**Добавление чекбоксов**

Чтобы добавить чекбоксы к узлам дерева, надо у TreeView установить свойство CheckBoxes = true:

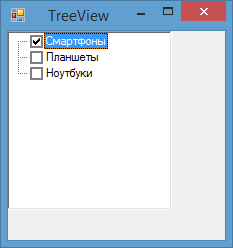
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | treeView1.CheckBoxes = true;  TreeNode smartNode = new TreeNode("Смартфоны");  smartNode.Checked = true;  treeView1.Nodes.Add(smartNode);    treeView1.Nodes.Add(new TreeNode("Планшеты"));  treeView1.Nodes.Add(new TreeNode("Ноутбуки")); |

**Добавление изображений**

Для добавления изображений нам нужен компонент ImageList, в котором имеется несколько картинок. Добавим эти картинки к узлам:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // установка источника изображений  treeView1.ImageList = imageList1;    TreeNode argentinaNode = new TreeNode { Text = "Аргентина", ImageIndex=0, SelectedImageIndex=0 };  treeView1.Nodes.Add(argentinaNode);    TreeNode braziliaNode = new TreeNode { Text = "Бразилия", ImageIndex = 1, SelectedImageIndex=1 };  treeView1.Nodes.Add(braziliaNode);    TreeNode chilieNode = new TreeNode { Text = "Чили", ImageIndex = 2, SelectedImageIndex=2 };  treeView1.Nodes.Add(chilieNode);    TreeNode columbiaNode = new TreeNode { Text = "Колумбия", ImageIndex = 3, SelectedImageIndex=3 };  treeView1.Nodes.Add(columbiaNode);  using System;  using System.Collections.Generic;  using System.ComponentModel;  using System.IO;  using System.Windows.Forms;    namespace HelloApp  {      public partial class Form1 : Form      {          public Form1()          {              InitializeComponent();                treeView1.BeforeSelect += treeView1\_BeforeSelect;              treeView1.BeforeExpand += treeView1\_BeforeExpand;              // заполняем дерево дисками              FillDriveNodes();          }          // событие перед раскрытием узла          void treeView1\_BeforeExpand(object sender, TreeViewCancelEventArgs e)          {              e.Node.Nodes.Clear();              string[] dirs;              try              {                  if (Directory.Exists(e.Node.FullPath))                  {                      dirs = Directory.GetDirectories(e.Node.FullPath);                      if (dirs.Length != 0)                      {                          for (int i = 0; i < dirs.Length; i++)                          {                              TreeNode dirNode = new TreeNode(new DirectoryInfo(dirs[i]).Name);                              FillTreeNode(dirNode, dirs[i]);                              e.Node.Nodes.Add(dirNode);                          }                      }                  }              }              catch (Exception ex) { }          }          // событие перед выделением узла          void treeView1\_BeforeSelect(object sender, TreeViewCancelEventArgs e)          {              e.Node.Nodes.Clear();              string[] dirs;              try              {                  if(Directory.Exists(e.Node.FullPath))                  {                      dirs = Directory.GetDirectories(e.Node.FullPath);                      if (dirs.Length!= 0)                      {                          for(int i=0; i<dirs.Length;i++)                          {                              TreeNode dirNode = new TreeNode(new DirectoryInfo(dirs[i]).Name);                              FillTreeNode(dirNode, dirs[i]);                              e.Node.Nodes.Add(dirNode);                          }                      }                  }              }              catch (Exception ex) { }          }            // получаем все диски на компьютере          private void FillDriveNodes()          {              try              {                  foreach(DriveInfo drive in DriveInfo.GetDrives())                  {                      TreeNode driveNode = new TreeNode { Text = drive.Name };                      FillTreeNode(driveNode, drive.Name);                      treeView1.Nodes.Add(driveNode);                  }              }              catch (Exception ex) { }          }          // получаем дочерние узлы для определенного узла          private void FillTreeNode(TreeNode driveNode, string path)          {              try              {                  string[] dirs = Directory.GetDirectories(path);                  foreach (string dir in dirs)                  {                      TreeNode dirNode = new TreeNode();                      dirNode.Text = dir.Remove(0, dir.LastIndexOf("\\") + 1);                      driveNode.Nodes.Add(dirNode);                  }              }              catch (Exception ex) { }          }      }  } |

При установке изображений надо учитывать, что если мы не установим свойство SelectedImageIndex для каждого узла, то в качестве картинки для выделенного узла по умолчанию будет использоваться первое изображение из ImageList.

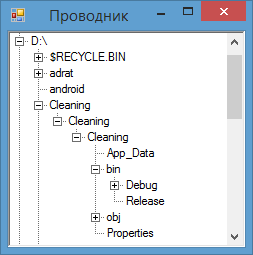


**TreeView. Практический пример**

Выполним небольшую задачу с TreeView. А именно попробуем сделать примитивный интерфейс на подобие проводника. Для этого добавим на форму элемент TreeView. А в файле кода формы пропишим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | using System;  using System.Collections.Generic;  using System.ComponentModel;  using System.IO;  using System.Windows.Forms;    namespace HelloApp  {      public partial class Form1 : Form      {          public Form1()          {              InitializeComponent();                treeView1.BeforeSelect += treeView1\_BeforeSelect;              treeView1.BeforeExpand += treeView1\_BeforeExpand;              // заполняем дерево дисками              FillDriveNodes();          }          // событие перед раскрытием узла          void treeView1\_BeforeExpand(object sender, TreeViewCancelEventArgs e)          {              e.Node.Nodes.Clear();              string[] dirs;              try              {                  if (Directory.Exists(e.Node.FullPath))                  {                      dirs = Directory.GetDirectories(e.Node.FullPath);                      if (dirs.Length != 0)                      {                          for (int i = 0; i < dirs.Length; i++)                          {                              TreeNode dirNode = new TreeNode(new DirectoryInfo(dirs[i]).Name);                              FillTreeNode(dirNode, dirs[i]);                              e.Node.Nodes.Add(dirNode);                          }                      }                  }              }              catch (Exception ex) { }          }          // событие перед выделением узла          void treeView1\_BeforeSelect(object sender, TreeViewCancelEventArgs e)          {              e.Node.Nodes.Clear();              string[] dirs;              try              {                  if(Directory.Exists(e.Node.FullPath))                  {                      dirs = Directory.GetDirectories(e.Node.FullPath);                      if (dirs.Length!= 0)                      {                          for(int i=0; i<dirs.Length;i++)                          {                              TreeNode dirNode = new TreeNode(new DirectoryInfo(dirs[i]).Name);                              FillTreeNode(dirNode, dirs[i]);                              e.Node.Nodes.Add(dirNode);                          }                      }                  }              }              catch (Exception ex) { }          }            // получаем все диски на компьютере          private void FillDriveNodes()          {              try              {                  foreach(DriveInfo drive in DriveInfo.GetDrives())                  {                      TreeNode driveNode = new TreeNode { Text = drive.Name };                      FillTreeNode(driveNode, drive.Name);                      treeView1.Nodes.Add(driveNode);                  }              }              catch (Exception ex) { }          }          // получаем дочерние узлы для определенного узла          private void FillTreeNode(TreeNode driveNode, string path)          {              try              {                  string[] dirs = Directory.GetDirectories(path);                  foreach (string dir in dirs)                  {                      TreeNode dirNode = new TreeNode();                      dirNode.Text = dir.Remove(0, dir.LastIndexOf("\\") + 1);                      driveNode.Nodes.Add(dirNode);                  }              }              catch (Exception ex) { }          }      }  } |

TreeView имеет ряд событий, которые позволяют нам управлять деревом. Наиболее важные из них:



* BeforeSelect / AfterSelect: срабатывает перед / после выбора узла дерева
* BeforeExpand / AfterExpand: срабатывает перед / после раскрытия узла дерева
* BeforeCollapse / AfterCollapse: срабатывает перед / после скрытия узла дерева

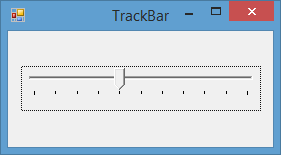
В вышеприведенном коде мы заблаговременно перед раскрытием или выбором наполняем выбранный узел дочерними подузлами, благодаря чему у нас появляется видимость, что узлы заполнены.

**TrackBar, Timer и ProgressBar**

**Последнее обновление: 31.10.2015**

**TrackBar**

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



Некоторые важные свойства TrackBar:

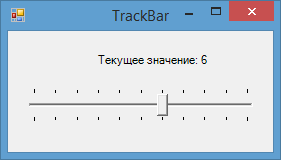
* **Orientation**: задает ориентацию ползунка - расположение по горизонтали или по вертикали
* **TickStyle**: задает расположение делений на ползунке
* **TickFrequency**: задает частоту делений на ползунке
* **Minimum**: минимальное возможное значение на ползунке (по умолчанию 0)
* **Maximum**: максимальное 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)

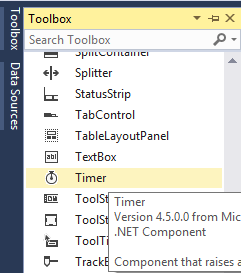
К наиболее важным событиям элемента следует отнести событие **Scroll**, которое позволяет обработать перемещение ползунка от одного деления к другому. Что может быть полезно, если нам надо, например, устанавливать соответствующую громкость звука в зависимости от значения ползунка, либо какике-нибудь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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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);      }  } |



**Timer**

Timer является компонентом для запуска действий, повторяющихся через определенный промежуток времени. Хотя он не является визуальным элементом, но его аткже можно перетащить с Панели Инструментов на форму:



Наиболее важные свойства и методы таймера:

* Свойство **Enabled**: при значении true указывает, что таймер будет запускаться вместе с запуском формы
* Свойство **Interval**: указывает интервал в миллисекундах, через который будет срабатывать обработчик события Tick, которое есть у таймера
* Метод **Start()**: запускает таймер
* Метод **Stop()**: останавливает таймер

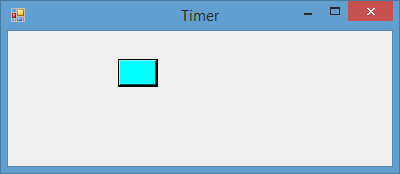
Для примера определим простую форму, на которую добавим кнопку и таймер. В файле кода формы определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | public partial class Form1 : Form  {      int koef = 1;      public Form1()      {          InitializeComponent();            this.Width = 400;          button1.Width = 40;          button1.Left = 40;          button1.Text = "";          button1.BackColor = Color.Aqua;            timer1.Interval = 500; // 500 миллисекунд          timer1.Enabled = true;          button1.Click += button1\_Click;          timer1.Tick += timer1\_Tick;      }      // обработчик события Tick таймера      void timer1\_Tick(object sender, EventArgs e)      {          if (button1.Left == (this.Width-button1.Width-10))          {              koef=-1;          }          else if (button1.Left == 0)          {              koef = 1;          }          button1.Left += 10 \*koef;      }      // обработчик нажатия на кнопку      void button1\_Click(object sender, EventArgs e)      {          if(timer1.Enabled==true)          {              timer1.Stop();          }          else          {              timer1.Start();          }      }  } |

Здесь в конструкторе формы устанавливаются начальные значения для таймера, кнопки и формы.

Через каждый интервал таймера будет срабатывать обработчик timer1\_Tick, в котором изменяется положение кнопки по горизонтали с помощью свойства button1.Left. А с помощью дополнительной переменной koef можно управлять направлением движения.

Кроме того, с помощью обраотчика нажатия кнопки button1\_Click можно либо остановить таймер (и вместе с ним движение кнопки), либо опять его запустить.



**Индикатор прогресса ProgressBar**

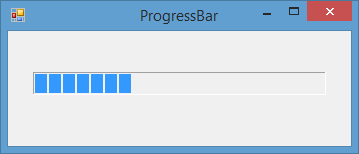
Элемент ProgressBar служит для того, чтобы дать пользователю информацию о ходе выполнения какой-либо задачи.

Наиболее важые свойства ProgressBar:

* **Minimum**: минимальное возможное значение
* **Maximum**: максимальное возможное значение
* **Value**: текущее значение элемента
* **Step**: шаг, на который изменится значение Value при вызове метода PerformStep

Для имитации работы прогрессбара поместим на форму таймер и в коде формы определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            timer1.Interval = 500; // 500 миллисекунд          timer1.Enabled = true;          timer1.Tick += timer1\_Tick;      }      // обработчик события Tick таймера      void timer1\_Tick(object sender, EventArgs e)      {          progressBar1.PerformStep();      }  } |



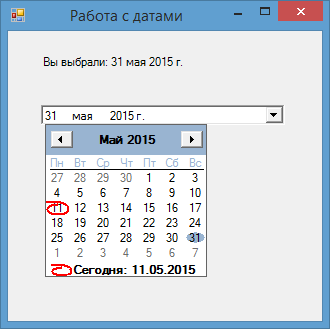
**DateTimePicker и MonthCalendar**

**Последнее обновление: 31.10.2015**

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

**DateTimePicker**

DateTimePicker представляет раскрывающийся по нажатию календарь, в котором можно выбрать дату. собой элемент, который с помощью перемещения ползунка позволяет вводить числовые значения.



Наиболее важные свойства DateTimePicker:

* **Format**: определяет формат отображения даты в элементе управления. Может принимать следующие значения:

Custom: формат задается разработчиком

Long: полная дата

Short: дата в сокращенном формате

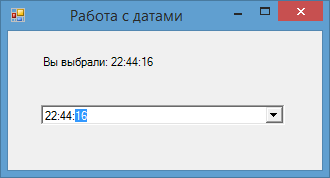
Time: формат для работы с временем

* **CustomFormat**: задает формат отображения даты, если для свойства Format установлено значение Custom
* **MinDate**: минимальная дата, которую можно выбрать
* **MaxDate**: наибольшая дата, которую можно выбрать
* **Value**: определеяте текущее выбранное значение в DateTimePicker
* **Text**: представляет тот текст, который отображается в элементе

При выборе даты элемент генерирует событие ValueChanged. Например, обработаем данное собатие и присвоим выбранное значение тексту метки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();          dateTimePicker1.Format = DateTimePickerFormat.Time;          dateTimePicker1.ValueChanged+=dateTimePicker1\_ValueChanged;      }        private void dateTimePicker1\_ValueChanged(object sender, EventArgs e)      {          label1.Text = String.Format("Вы выбрали: {0}", dateTimePicker1.Value.ToLongTimeString());      }  } |

Свойство Value хранит объект DateTime, поэтому с ним можно работать как и с любой другой датой. В данном случае выбранная дата преобразуется в строку времени.

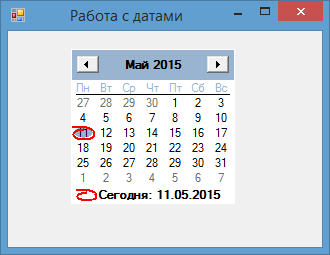


В вышеприведенном случае значение dateTimePicker1.Value.ToLongTimeString() аналогично тому тексту, который отображается в элементе. И мы могли бы написать так:

|  |  |
| --- | --- |
| 1 | label1.Text = String.Format("Вы выбрали: {0}", dateTimePicker1.Text); |

**MonthCalendar**

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



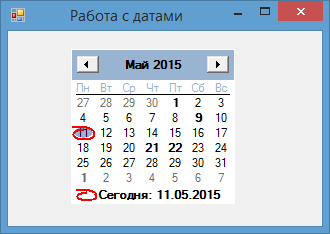
Рассмотрим некоторые основные свойства элемента.

Свойства выделения дат:

* **AnnuallyBoldedDates**: содержит набор дат, которые будут отмечены жирным в календаре для каждого года
* **BoldedDates**: содержит набор дат, которые будут отмечены жирным (только для текущего года)
* **MonthlyBoldedDates**: содержит набор дат, которые будут отмечены жирным для каждого месяца

Добавление выделенных дат делается с помощью определенных методов (как и удаление):

|  |  |
| --- | --- |
| 1  2  3  4 | monthCalendar1.AddBoldedDate(new DateTime(2015, 05, 21));  monthCalendar1.AddBoldedDate(new DateTime(2015, 05, 22));  monthCalendar1.AddAnnuallyBoldedDate(new DateTime(2015, 05, 9));  monthCalendar1.AddMonthlyBoldedDate(new DateTime(2015, 05, 1)); |



Для снятия выделения можно использовать аналоги этих методов:

|  |  |
| --- | --- |
| 1  2  3  4 | monthCalendar1.RemoveBoldedDate(new DateTime(2015, 05, 21));  monthCalendar1.RemoveBoldedDate(new DateTime(2015, 05, 22));  monthCalendar1.RemoveAnnuallyBoldedDate(new DateTime(2015, 05, 9));  monthCalendar1.RemoveMonthlyBoldedDate(new DateTime(2015, 05, 1)); |

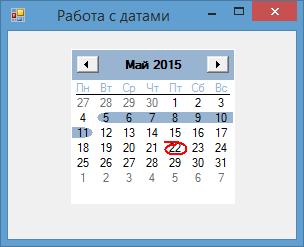
Свойства для определения дат в календаре:

* **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
* **SelectionRange**: определяет диапазон выделенных дат
* **SelectionEnd**: задает конечную дату выделения
* **SelectionStart**: определяет начальную дату выделения
* **ShowToday**: при значении true отображает внизу календаря текущую дату
* **ShowTodayCircle**: при значении true текущая дата будет обведена кружочком
* **TodayDate**: определяет текущую дату. По умолчанию используется системная дата на компьютере, но с помощью данного свойства мы можем ее изменить

Например, при установке свойств:

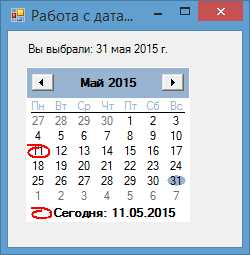
|  |  |
| --- | --- |
| 1  2  3  4  5 | monthCalendar1.TodayDate= new DateTime(2015, 05, 22);  monthCalendar1.ShowTodayCircle = true;  monthCalendar1.ShowToday = false;  monthCalendar1.SelectionStart = new DateTime(2015, 05, 1);  monthCalendar1.SelectionEnd = new DateTime(2015, 05, 11); |

будет следующее отображение календаря:



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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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());      }  } |



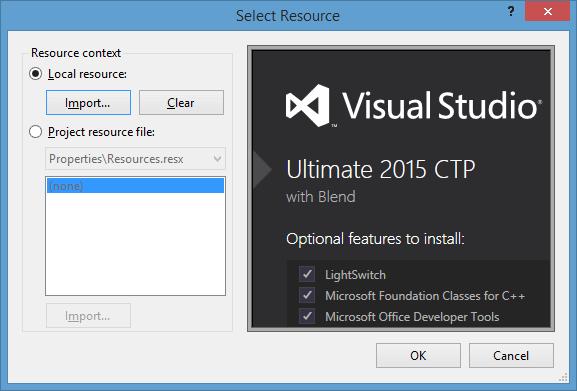
## Элемент PictureBox

**Последнее обновление: 31.10.2015**

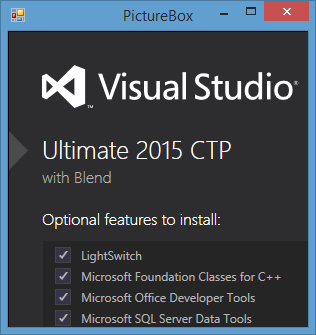
PictureBox предназначен для показа изображений. Он позволяет отобразить файлы в формате bmp, jpg, gif, а также метафайлы ищображений и иконки. Для установки изображения в PictureBox можно использовать ряд свойств:

* **Image**: устанавливает объект типа Image
* **ImageLocation**: устанавливает путь к изображению на диске или в интернете
* **InitialImage**: некоторое начальное изображение, которое будет отображаться во время загрузки главного изображения, которое хранится в свойстве Image
* **ErrorImage**: изображение, которое отображается, если основное изображение не удалось загрузить в PictureBox

Чтобы установить изображение в Visual Studio, надо в панели Свойств PictureBox выбрать свойство Image. В этом случае нам откроется окно импорта изображения в проект, где мы собственно и сможем выбрать нужное изображение на компьютере и установить его для PictureBox:



И затем мы сможем увидеть данное изображение в PictureBox:



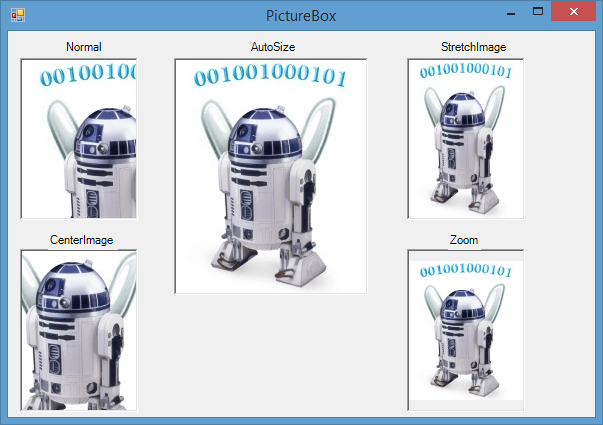
Либо можно загрузить изображение в коде:

|  |  |
| --- | --- |
| 1 | pictureBox1.Image = Image.FromFile("C:\Users\Eugene\Pictures\12.jpg"); |

**Размер изображения**

Для установки изображения в PictureBox используется свойство **SizeMode**, которое принимает следующие значения:

* Normal: изображение позиционируется в левом верхнем углу PictureBox, и размер изображения не изменяется. Если PictureBox больше размеров изображения, то по справа и снизу появляются пустоты, если меньше - то изображение обрезается
* StretchImage: изображение растягивается или сжимается таким обраом, чобы вместиться по всей ширине и высоте элемента PictureBox
* AutoSize: элемент PictureBox автоматически растягивается, подстраиваясь под размеры изображения
* CenterImage: если PictureBox меньше изображения, то изображение обрезается по краям и выводится только его центральная часть. Если же PictureBox больше изображения, то оно позиционируется по центру.
* Zoom: изоражение подстраивается под размеры PictureBox, сохраняя при этом пропорции



**Элемент WebBrowser**

**Последнее обновление: 31.10.2015**

WebBrowser предоставляет функции интернет-браузера, позволяя загружать и отображать контент из сети интернет. В то же время важно понимать, что данный элемент не является полноценным веб-браузером, и возможности по его настройки и изменению довольно ограничены.

Рассмотрим основные его свойства:

* **AllowWebBrowserDrop**: при установке для данного свойства значения true можно будет с помощью мыши переносить документы в веб-браузер и открывать их.
* **CanGoBack**: определяет, может ли веб-браузер переходить назад по истории просмотров
* **CanGoForward**: определяет, может ли веб-браузер переходить вперед
* **Document**: возвращает открытый в веб-браузере документ
* **DocumentText**: возвращает текстовое содержание документа
* **DocumentTitle**: возвращает заголовок документа
* **DocumentType**: возвращает тип документа
* **IsOffline**: возвращает true, если отсутствует подключение к интернету
* **ScriptErrorsSuppressed**: указывает, будут ли отображаться ошибки javascript в диалоговом окне
* **ScrollBarsEnabled**: определяет, будет ли использоваться прокрутка
* **URL**: возвращает или устанавливает URL документа в веб-браузере

Кроме того, WebBrowser содержит ряд методов, которые позволяют осуществлять навигацию между документами:

* **GoBack()**: осуществляет переход к предыдущей странице в истории навигации (если таковая имеется)
* **GoForward()**: осуществляет переход к следующей странице в истории навигации
* **GoHome()**: осуществляет переход к домашней странице веб-браузера
* **GoSearch()**: осуществляет переход к странице поиска
* **Navigate**: осуществляет переход к определенному адресу в сети интернет

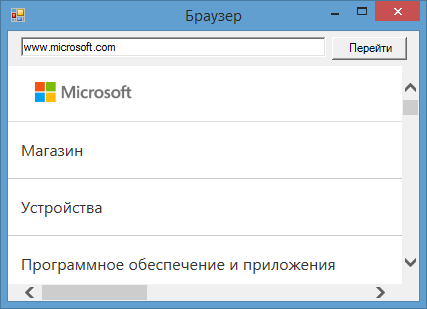
Таким образом, чтобы перейти к определенному документу, надо исользовать метод Navigate:

|  |  |
| --- | --- |
| 1  2  3  4 | // перейти к адресу в интернете  webBrowser1.Navigate("[http://google.com](http://google.com/)");  // открыть документ на диске  webBrowser1.Navigate("[C://Images//24.png](file:///C:\Images\24.png)"); |

Создадим небольшй веб-браузер. Для этого поместим на форму элементы WebBrowser, TextBox (в него будем вводить адрес) и Button. И в файле формы пропишем следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();          // установка начального адреса          webBrowser1.Url=new Uri("[http://google.com](http://google.com/)");          button1.Click+=button1\_Click;      }        private void button1\_Click(object sender, EventArgs e)      {          webBrowser1.Navigate(textBox1.Text);      }  } |

И по нажатию кнопки произойдет переход к адресу, введенному в текстовое поле:



**Элемент NotifyIcon**

**Последнее обновление: 31.10.2015**

Элемент NotifyIcon позволяет задать значок, который будет отображаться при запуске приложения в панели задач.

Рассмотрим основные его свойства:

* **BallonTipIcon**: иконка, которая будет использоваться на всплывающей подсказке. Это свойство может иметь следующие значения: None, Info, Warning, Error.
* **BalloonTipText**: текст, отображаемый во всплывающей подсказке
* **BalloonTipTitle**: загаловок всплывающей подсказки
* **ContextMenuStrip**: устанавливает контекстное меню для объекта NotifyIcon
* **Icon**: задает значок, который будет отображаться в системном трее
* **Text**: устанавливает текст всплывающей подсказки, которая появляется при нахождении указателя мыши над значком
* **Visible**: устанавливает видимость значка в системном трее

Чтобы добавить на форму NotifyIcon, перенесем данный элемент на форму с панели инструментов. После этого добавленный компонент NotifyIcon отобразится внизу дизайнера формы.

Затем зададим у NotifyIcon для свойства Icon какую-нибудь иконку в формате *.ico*. И также установим для свойства Visibleзначение true.

Далее также зададим у NotifyIcon для свойства Text какой-нибудь текст, например, "Показать форму". Этот текст отобразится при прохождении указателя мыши над значком NotifyIcon в системном трее.

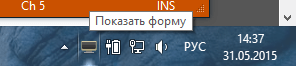
Чтобы можно было открыть форму по клику на значок в трее, надо обработать событие Click у NotifyIcon. Поэтому в коде формы определим обработчик для этого события:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            this.ShowInTaskbar = false;          notifyIcon1.Click += notifyIcon1\_Click;      }        void notifyIcon1\_Click(object sender, EventArgs e)      {          this.WindowState = FormWindowState.Normal;      }  } |

В обработчике просто переводим форму из минимизированного состояния в обычное.

И кроме того, чтобы форма не отображалась на панели задач, у нее задаем свойство **ShowInTaskbar = false**.

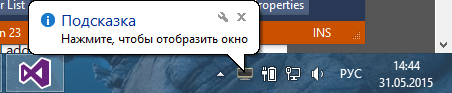
В итоге после запуска приложения в трее будет отображаться значок NotifyIcon, нажав на который при свернутой форме, мы ее можем заново открыть.



Теперь используем всплывающую подсказку. Для этого изменим конструктор формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            this.ShowInTaskbar = false;          notifyIcon1.Click += notifyIcon1\_Click;            // задаем иконку всплывающей подсказки          notifyIcon1.BalloonTipIcon = ToolTipIcon.Info;          // задаем текст подсказки          notifyIcon1.BalloonTipText = "Нажмите, чтобы отобразить окно";          // устанавливаем зголовк          notifyIcon1.BalloonTipTitle = "Подсказка";          // отображаем подсказку 12 секунд          notifyIcon1.ShowBalloonTip(12);      }        void notifyIcon1\_Click(object sender, EventArgs e)      {          this.WindowState = FormWindowState.Normal;      }  } |

И при запуске отобразится всплывающая подсказка:



**Окно сообщения MessageBox**

**Последнее обновление: 31.10.2015**

Как правило, для вывода сообщений применяется элемент MessageBox. Однако кроме сообственно 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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**:

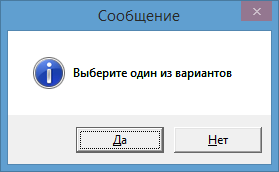
* Button1: первая кнопка из тех, которые задаются перечислением MessageBoxButtons
* Button2: вторая кнопка
* Button3: третья кнопка

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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 и No
* No: нажата кнопка No

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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;        this.TopMost = true;  } |

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

**OpenFileDialog и SaveFileDialog**

**Последнее обновление: 31.10.2015**

Окна открытия и сохранения файла представлены классами **OpenFileDialog** и **SaveFileDialog**. Они имеют во многом схожую функциональность, поэтому рассмотрим их вместе.

OpenFileDialog и SaveFileDialog имеют ряд общих свойств, среди которых можно выделить следующие:

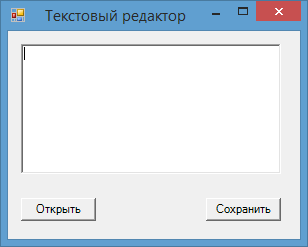
* DefaultExt: устанавливает расширение файла, которое добавляется по умолчанию, если пользователь ввел имя файла без расширения
* AddExtension: при значении true добавляет к имени файла расширение при его отсуствии. Расширение берется из свойства DefaultExt или Filter
* CheckFileExists: если имеет значение true, то проверяет существование файла с указанным именем
* CheckPathExists: если имеет значение true, то проверяет существование пути к файлу с указанным именем
* FileName: возвращает полное имя файла, выбранного в диалоговом окне
* Filter: задает фильтр файлов, благодаря чему в диалоговом окне можно отфильтровать файлы по расширению. Фильтр задается в следующем формате *Название\_файлов|\*.расширение*. Например, Текстовые файлы(\*.txt)|\*.txt. Можно задать сразу несколько фильтров, для этого они разделяются вертикальной линией |. Например, Bitmap files (\*.bmp)|\*.bmp|Image files (\*.jpg)|\*.jpg
* InitialDirectory: устанавливает каталог, который отображается при первом вызове окна
* Title: заголовок диалогового окна

Отдельно у класса SaveFileDialog можно еще выделить пару свойств:

* CreatePrompt: при значении true в случае, если указан не существующий файл, то будет отображаться сообщение о его создании
* OverwritePrompt: при значении true в случае, если указан существующий файл, то будет отображаться сообщение о том, что файл будет перезаписан

Чтобы отобразить диалоговое окно, надо вызвать метод ShowDialog().

Рассмотрим оба диалоговых окна на примере. Добавим на форму текстовое поле textBox1 и две кнопки button1 и button2. Также перетащим с панели инструментов компоненты OpenFileDialog и SaveFileDialog. После добавления они отобразятся внизу дизайнера формы. В итоге форма будет выглядеть примерно так:



Теперь изменим код формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            button1.Click += button1\_Click;          button2.Click += button2\_Click;          openFileDialog1.Filter = "Text files(\*.txt)|\*.txt|All files(\*.\*)|\*.\*";          saveFileDialog1.Filter = "Text files(\*.txt)|\*.txt|All files(\*.\*)|\*.\*";      }      // сохранение файла      void button2\_Click(object sender, EventArgs e)      {          if (saveFileDialog1.ShowDialog() == DialogResult.Cancel)              return;          // получаем выбранный файл          string filename = saveFileDialog1.FileName;          // сохраняем текст в файл          System.IO.File.WriteAllText(filename, textBox1.Text);          MessageBox.Show("Файл сохранен");      }      // открытие файла      void button1\_Click(object sender, EventArgs e)      {          if (openFileDialog1.ShowDialog() == DialogResult.Cancel)              return;          // получаем выбранный файл          string filename = openFileDialog1.FileName;          // читаем файл в строку          string fileText = System.IO.File.ReadAllText(filename);          textBox1.Text = fileText;          MessageBox.Show("Файл открыт");      }  } |

По нажатию на первую кнопку будет открываться окно открытия файла. После выбора файла он будет считываться, а его текст будет отображаться в текстовом поле. Клик на вторую кнопку отобразит окно для сохранения файла, в котором надо установить его название. И после этого произойдет сохранение текста из текстового поля в файл.

**FontDialog и ColorDialog**

**FontDialog**

Для выбора шифта и его параметров используется **FontDialog**. Для его использования перенесем компонент с Панели инструментов на форму. И пусть на форме имеется кнопка button1. Тогда в коде формы пропишем следующее:

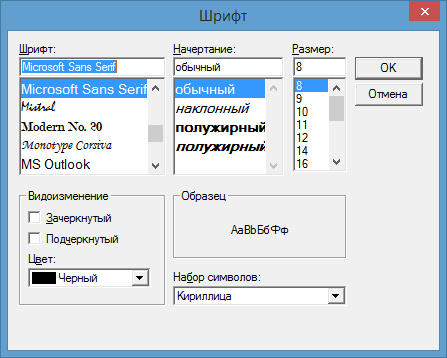
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            button1.Click += button1\_Click;          // добавляем возможность выбора цвета шрифта          fontDialog1.ShowColor = true;      }        void button1\_Click(object sender, EventArgs e)      {          if (fontDialog1.ShowDialog() == DialogResult.Cancel)              return;          // установка шрифта          button1.Font = fontDialog1.Font;          // установка цвета шрифта          button1.ForeColor = fontDialog1.Color;      }  } |

FontDialog имеет ряд свойств, среди которых стоит отметить следующие:

* ShowColor: при значении true позволяет выбирать цвет шрифта
* Font: выбранный в диалоговом окне шрифт
* Color: выбранный в диалоговом окне цвет шрифта

Для отображения диалогового окна используется метод ShowDialog().

И если мы запустим приложение и нажмем на кнопку, то нам отобразится диалоговое окно, где мы можем задать все параметры шрифта. И после выбоа установленные настройки будут применены к шрифту кнопки:



**ColorDialog**

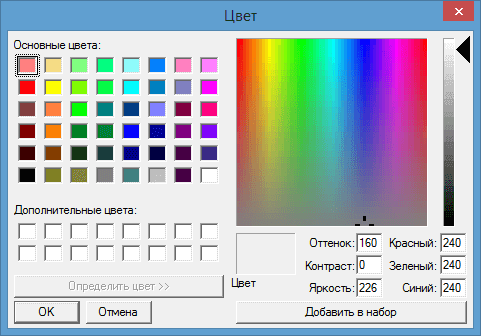
ColorDialog позволяет выбрать настройки цвета. Также перенесем его с Панели инструментов на форму. И изменим код формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            button1.Click += button1\_Click;          // расширенное окно для выбора цвета          colorDialog1.FullOpen = true;          // установка начального цвета для colorDialog          colorDialog1.Color = this.BackColor;      }        void button1\_Click(object sender, EventArgs e)      {          if (colorDialog1.ShowDialog() == DialogResult.Cancel)              return;          // установка цвета формы          this.BackColor = colorDialog1.Color;      }  } |

Среди свойств ColorDialog следует отметить следующие:

* FullOpen: при значении true отображается диалоговое окно с расширенными настройками для выбора цвета
* SolidColorOnly: при значении true позволяет выбирать только между отнотонные оттенки цветов
* Color: выбранный в диалоговом окне цвет

И при нажатии кнопку нам отобразится диалоговое окно, в котором можно установить цвет формы:



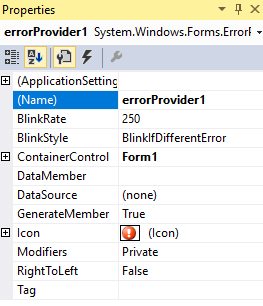
**ErrorProvider**

**Последнее обновление: 09.11.2016**

ErrorProvider не является полноценным визуальным компонентом, тем не менее он позволяет настраивать визуальное отображение ошибок при вводе пользователя. Этот элемент применяется преимущестенно 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 оторазится под формой, а в окне свойств мы также сможем управлять его свойствами:



Среди его свойств можно выделить следующие:

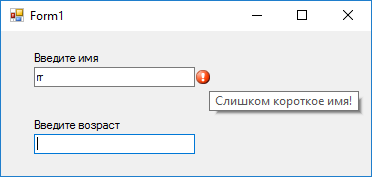
* **BlinkRate**: задает частоту мигания значка ошибки
* **BlinkStyle**: задает, когда значок ошибки будет мигать
* **Icon**: устанавливает сам значок ошибки. По умолчанию это красный кружок с восклицательным знаком, но можно установить любую другую иконку.

В коде формы приложения пропишем следующее:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | using System;  using System.ComponentModel;  using System.Windows.Forms;    namespace HelloApp  {      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();              }          }      }  } |

Здесь для обоих текстовых полей задано событие Validating, которое срабатывает при вводе пользователя и призвано управлять валидацией вода. Это событие имеется и у других элементов управления, не только у текстовых полей. В обработчике события Validating мы смотрим на введенный текст, и если он не удовлетворяет условиям, то с помощью методаerrorProvider1.SetError() для определенного элемента добавляем ошибку. Если все условия соблюдены, то, наоборот, удаляем все ошибки с помощью метода errorProvider1.Clear().

Запустим приложение и при некорректном вводе мы увидим ошибку:



Здесь есть небольшая деталь - валидация элемента будет происходить, когда мы завершим ввод и перейдем на другой элемент. Если же нам надо валидировать элемент по мере ввода, то тогда мы можем обрабатывать событие TextChanged у тех же текстовых полей.

<https://metanit.com/sharp/windowsforms/5.1.php>

**Меню и панели инструментов**

**Панель инструментов ToolStrip**

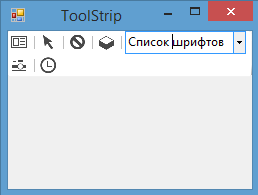
Элемент ToolStrip представляет панель инструментов. Каждый отдельный элемент на этой панели является объектом**ToolStripItem**.

Ключевые свойства компонента ToolStrip связаны с его позиционированием на форме:

* Dock: прикрепляет панель инструментов к одной из сторон формы
* LayoutStyle: задает ориентацию панели на форме (горизонтальная, вертикальная, табличная)
* ShowItemToolTips: указывает, будут ли отображаться всплывающие подсказки для отдельных элементов панели инструментов
* Stretch: позволяет растянуть панель по всей длине контейнера

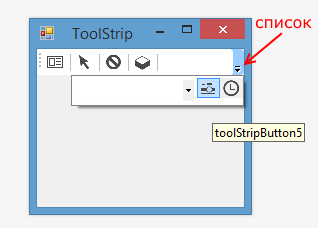
В зависимости от значения свойства LayoutStyle панель инструментов может располагаться по горизонтали, или в табличном виде:

* HorizontalStackWithOverflow: расположение по горизонтали с переполнением - если длина панели превышает длину контейнера, то новые элементы, выходящие за границы контейнера, не отображаются, то есть панель переполняется элементами
* StackWithOverflow: элементы располагаются автоматически с переполнением
* VerticalStackWithOverflow: элементы располагаются вертикально с переполнением
* Flow: элементы располагаются автоматически, но без переполнения - если длина панели меньше длины контейнера, то выходящие за границы элементы переносятся, а панель инструментов растягивается, чтобы вместить все элементы



* Table: элементы позиционируются в виде таблицы

Если LayoutStyle имеет значения HorizontalStackWithOverflow / VerticalStackWithOverflow, то с помощью свойства**CanOverflow** мы можем задать поведение при переполнении. Так, если это свойство равно true (значение по умолчанию), то для элементов, не попадающих в границы ToolStrip, создается выпадающий список:

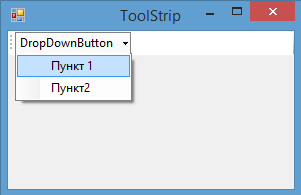


При значении false подобный выпадающий список не создается.

**Типы элементов панели и их добавление**

Панель ToolStrip может содержать объекты следующих классов

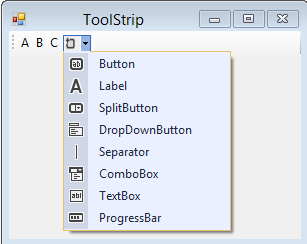
* **ToolStripLabel**: текстовая метка на панели инструментов, представляет функциональность элементов Label и LinkLabel
* **ToolStripButton**: аналогичен элементу Button.Также имеет событие Click, с помощью которого можно обработать нажатие пользователя на кнопку
* **ToolStripSeparator**: визуальный разделитель между другими элементами на панели инструментов
* **ToolStripToolStripComboBox**: подобен стандартному элементу ComboBox
* **ToolStripTextBox**: аналогичен текстовому полю TextBox
* **ToolStripProgressBar**: индикатор прогресса, как и элемент ProgressBar
* **ToolStripDropDownButton**: представляет кнопку, по нажатию на которую открывается выпадающее меню



К каждому элементу выпадающего меню дополнительно можно прикрепить обработчик нажатия и обработать клик по этим пунктам меню

* **ToolStripSplitButton**: объединяет функциональность ToolStripDropDownButton и ToolStripButton

Добавить новые элементы можно в режиме дизайнера:



Также можно добавлять новые элементы программно в коде. Их расположение на панели инструментов будет соответствовать порядку добавления. Все элементы хранятся в ToolStrip в свойстве Items. Мы можем добавить в него любой объект класса ToolStripItem (то есть любой из выше перечисленных классов, так как они наследуются от ToolStripItem):

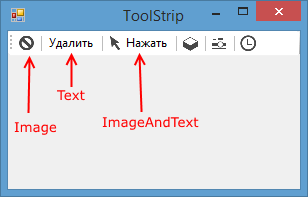
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            ToolStripButton clearBtn = new ToolStripButton();          clearBtn.Text = "Clear";          // устанавливаем обработчик нажатия          clearBtn.Click += btn\_Click;          toolStrip1.Items.Add(clearBtn);      }        void btn\_Click(object sender, EventArgs e)      {          MessageBox.Show("Производится удаление");      }  } |

Кроме того, здесь задается обработчик, позволяющий обрабатывать нажатия по кнопки на панели инструментов.

Элементы ToolStripButton, ToolStripDropDownButton и ToolStripSplitButton могут отображать как текст, так и изобажения, оибо сразу и то, и другое. Для управления размещением изображений в этих элементах имеются следующие свойства:

* DisplayStyle: определяет, будет ли отображаться на элементе текст, или изображение, или и то и другое.
* Image: указывает на само изображение
* ImageAlign: устанавливает выравнивание изображения относительно элемента
* ImageScaling: указывает, будет ли изображение растягиваться, чтобы заполнить все пространство элемента
* ImageTransparentColor: указывает, будет ли цвет изображения прозрачным

Чтобы указать разместить изображение на кнопке, у свойства DisplayStyle надо установить значение Image. Если мы хотим, чтобы кнопка отображала только текст, то надо указать значение Text, либо можно комбинировать два значения с помощью другого значения ImageAndText:



Все эти значения хранятся в перечислении **ToolStripItemDisplayStyle**. Также можно установить свойства в коде c#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ToolStripButton clearBtn = new ToolStripButton();  clearBtn.Text = "Поиск";  clearBtn.DisplayStyle = ToolStripItemDisplayStyle.ImageAndText;  clearBtn.Image = Image.FromFile(@"D:\Icons\0023\search32.png");  // добавляем на панель инструментов  toolStrip1.Items.Add(clearBtn); |

<https://metanit.com/sharp/windowsforms/5.1.php>

<https://metanit.com/sharp/windowsforms/5.2.php>

**Создание меню MenuStrip**

**Последнее обновление: 31.10.2015**

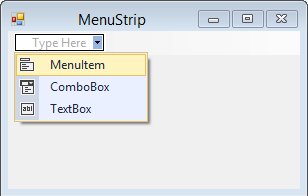
Для создания меню в Windows Forms применяется элемент **MenuStrip**. Данный класс унаследован от ToolStrip и поэтому наследует его функциональность.

Наиболее важные свойства компонента MenuStrip:

* Dock: прикрепляет меню к одной из сторон формы
* LayoutStyle: задает ориентацию панели меню на форме. Может также, как и с ToolStrip, принимать следующие значения
  + HorizontalStackWithOverflow: расположение по горизонтали с переполнением - если длина меню превышает длину контейнера, то новые элементы, выходящие за границы контейнера, не отображаются, то есть панель переполняется элементами
  + StackWithOverflow: элементы располагаются автоматически с переполнением
  + VerticalStackWithOverflow: элементы располагаются вертикально с переполнением
  + Flow: элементы размещаются автоматически, но без переполнения - если длина панели меню меньше длины контейнера, то выходящие за границы элементы переносятся
  + Table: элементы позиционируются в виде таблицы
* ShowItemToolTips: указывает, будут ли отображаться всплывающие подсказки для отдельных элементов меню
* Stretch: позволяет растянуть панель по всей длине контейнера
* TextDirection: задает направление текста в пунктах меню

MenuStrip выступает своего рода контейнером для отдельных пунктов меню, которые представлены объектом**ToolStripMenuItem**.

Добавить новые элементы в меню можно в режиме дизайнера:



Для добавления доступно три вида элементов: MenuItem (объект ToolStripMenuItem), ComboBox и TextBox. Таким образом, в меню мы можем использовать выпадающие списки и текстовые поля, однако, как правило, эти элементы применяются в основном на панели инструментов. Меню же обычно содержит набор объектов ToolStripMenuItem.

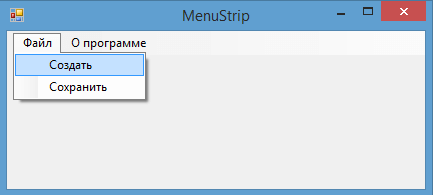
Также мы можем добавить пункты меню в коде C#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            ToolStripMenuItem fileItem = new ToolStripMenuItem("Файл");            fileItem.DropDownItems.Add("Создать");          fileItem.DropDownItems.Add(new ToolStripMenuItem("Сохранить"));            menuStrip1.Items.Add(fileItem);            ToolStripMenuItem aboutItem = new ToolStripMenuItem("О программе");          aboutItem.Click += aboutItem\_Click;          menuStrip1.Items.Add(aboutItem);      }        void aboutItem\_Click(object sender, EventArgs e)      {          MessageBox.Show("О программе");      }  } |

ToolStripMenuItem в конструкторе принимает текстовую метку, которая будет использоваться в качестве текста меню. Каждый подобный объект имеет коллекцию **DropDownItems**, которая хранит дочерние объекты ToolStripMenuItem. То есть один элемент ToolStripMenuItem может содержать набор других объектов ToolStripMenuItem. И таким образом, образуется иерархическое меню или структура в виде дерева.

Если передать при добавление строку текста, то для нее неявным образом будет создан объект ToolStripMenuItem:fileItem.DropDownItems.Add("Создать")

Назначив обработчики для события Click, мы можем обработать нажатия на пункты меню: aboutItem.Click += aboutItem\_Click



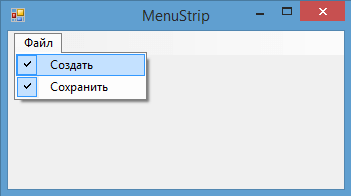
**Отметки пунктов меню**

Свойство CheckOnClick при значении true позволяет на клику отметить пункт меню. А с помощью свойства Checked можно установить, будет ли пункт меню отмечен при запуске программы.

Еще одно свойство CheckState возвращает состояние пункта меню - отмечен он или нет. Оно может принимать три значения:Checked (отмечен), Unchecked (неотмечен) и Indeterminate (в неопределенном состоянии)

Например, создадим ряд отмеченных пунктов меню и обработаем событие установки / снятия отметки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            ToolStripMenuItem fileItem = new ToolStripMenuItem("Файл");            ToolStripMenuItem newItem = new ToolStripMenuItem("Создать") { Checked = true, CheckOnClick = true };          fileItem.DropDownItems.Add(newItem);              ToolStripMenuItem saveItem = new ToolStripMenuItem("Сохранить") { Checked = true, CheckOnClick = true };          saveItem.CheckedChanged += menuItem\_CheckedChanged;            fileItem.DropDownItems.Add(saveItem);            menuStrip1.Items.Add(fileItem);      }        void menuItem\_CheckedChanged(object sender, EventArgs e)      {          ToolStripMenuItem menuItem = sender as ToolStripMenuItem;          if (menuItem.CheckState == CheckState.Checked)              MessageBox.Show("Отмечен");          else if (menuItem.CheckState == CheckState.Unchecked)              MessageBox.Show("Отметка снята");      }  } |



**Клавиши быстрого доступа**

Если нам надо быстро обратиться к какому-то пункту меню, то мы можем использовать клавиши быстрого доступа. Для задания клавиш быстрого доступа используется свойство **ShortcutKeys**:

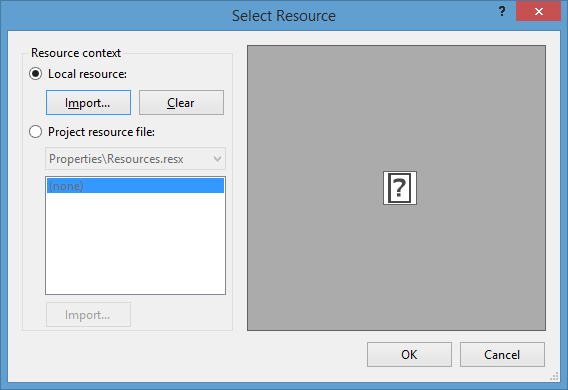
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();            ToolStripMenuItem fileItem = new ToolStripMenuItem("Файл");            ToolStripMenuItem saveItem = new ToolStripMenuItem("Сохранить") { Checked = true, CheckOnClick = true };          saveItem.Click+=saveItem\_Click;          saveItem.ShortcutKeys = Keys.Control | Keys.P;            fileItem.DropDownItems.Add(saveItem);          menuStrip1.Items.Add(fileItem);      }        void saveItem\_Click(object sender, EventArgs e)      {          MessageBox.Show("Сохранение");      }  } |

Клавиши задаются с помощью перечисления Keys. В данном случае по нажатию на комбинацию клавиш Ctrl + P, будет срабатывать нажатие на пункт меню "Сохранить".

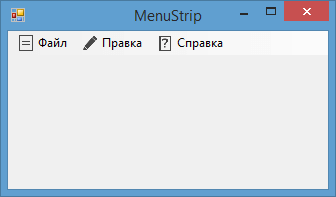
С помощью изображений мы можем разнообразить внешний вид пунктов меню. Для этого мы можем использовать следующие свойства:

* DisplayStyle: определяет, будет ли отображаться на элементе текст, или изображение, или и то и другое.
* Image: указывает на само изображение
* ImageAlign: устанавливает выравнивание изображения относительно элемента
* ImageScaling: указывает, будет ли изображение растягиваться, чтобы заполнить все пространство элемента
* ImageTransparentColor: указывает, будет ли цвет изображения прозрачным

Если изображение для пункта меню устанавливает в режиме дизайнера, то нам надо выбрать в окне свойство пункт Image, после чего откроется окно для импорта ресурса изображения в проект



Чтобы указать, как разместить изображение, у свойства DisplayStyle надо установить значение Image. Если мы хотим, чтобы кнопка отображала только текст, то надо указать значение Text, либо можно комбинировать два значения с помощью другого значения ImageAndText. По умолчанию изображение размещается слева от текста:



Также можно установить изображение динамически в коде:

|  |  |
| --- | --- |
| 1 | fileToolStripMenuItem.Image = Image.FromFile(@"D:\Icons\0023\block32.png"); |

**Строка состояния StatusStrip**

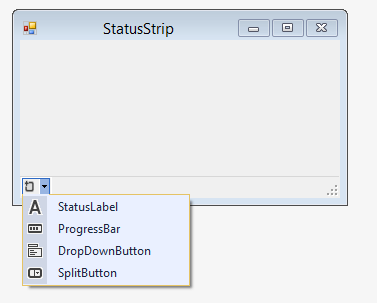
**Последнее обновление: 31.10.2015**

StatusStrip представляет строку состояния, во многом аналогичную панели инструментов ToolStrip. Строка состояния предназначена для отображения текущей информации о состоянии работы приложения.

При добавлении на форму StatusStrip автоматически размещается в нижней части окна приложения (как и в большинстве приложений). Однако при необходимости мы сможем его иначе позиционировать, управляя свойством Dock, которое может принимать следующие значения:

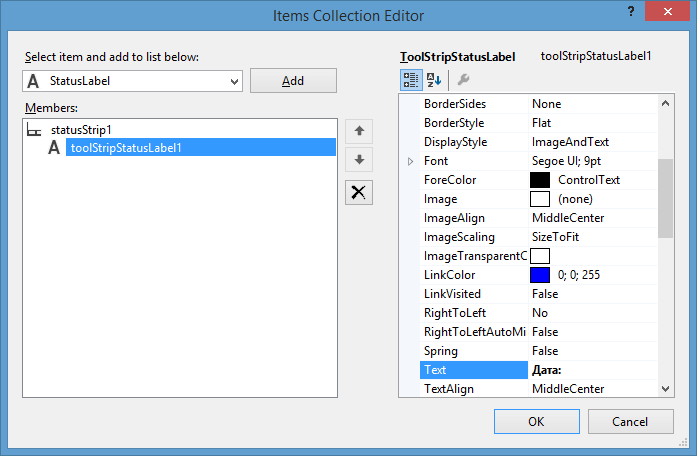
* Bottom: размещение внизу (значение по умолчанию)
* Top: прикрепляет статусную строку к верхней части формы
* Fill: растягивает на всю форму
* Left: размещение в левой части формы
* Right: размещение в правой части формы
* None: произвольное положение

StatusStrip может содержать различные элементы. В режиме дизайнера мы можем добавить следующие типы элементов:



* StatusLabel: метка для вывода текстовой информации. Представляет объект ToolStripLabel
* ProgressBar: индикатор прогресса. Представляет объект ToolStripProgressBar
* DropDownButton: кнопка с выпадающим списком по клику. Представляет объект ToolStripDropDownButton
* SplitButton: еще одна кнопка, во многом аналогичная DropDownButton. Представляет объект ToolStripSplitButton

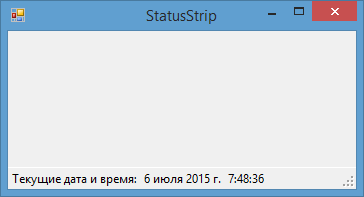
Либо можно обратиться на панели свойств к свойству Items компонента StatusStrip и открывшемся окне добавить и настроить все элементы:



Также мы можем добавить элементы программно. Создадим небольшую программу. Определим следующий код формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public partial class Form1 : Form  {      ToolStripLabel dateLabel;      ToolStripLabel timeLabel;      ToolStripLabel infoLabel;      Timer timer;      public Form1()      {          InitializeComponent();            infoLabel = new ToolStripLabel();          infoLabel.Text = "Текущие дата и время:";          dateLabel = new ToolStripLabel();          timeLabel = new ToolStripLabel();            statusStrip1.Items.Add(infoLabel);          statusStrip1.Items.Add(dateLabel);          statusStrip1.Items.Add(timeLabel);            timer = new Timer() { Interval = 1000 };          timer.Tick += timer\_Tick;          timer.Start();      }        void timer\_Tick(object sender, EventArgs e)      {          dateLabel.Text = DateTime.Now.ToLongDateString();          timeLabel.Text = DateTime.Now.ToLongTimeString();      }  } |

Здесь создаются три метки на строке состояния и таймер. После создания формы таймер запускается, и срабатывает его событие Tick, в обработчике которого устанавливаем текст меток.

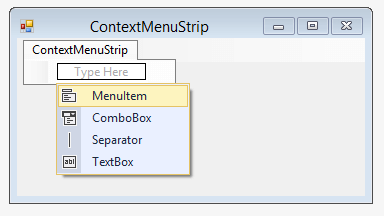


**Контекстное меню ContextMenuStrip**

**Последнее обновление: 31.10.2015**

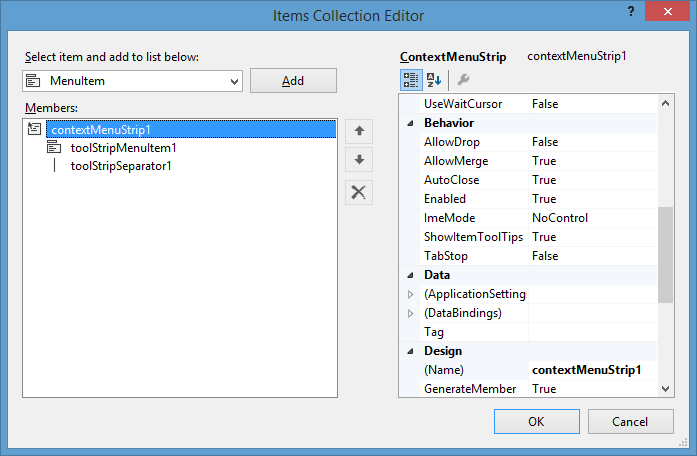
ContextMenuStrip представляет контекстное меню. Данный компонент во многом аналогичен элементы MenuStrip за тем исключением, что контекстное меню не может использоваться само по себе, оно обязательно применяется к какому-нибудь другому элементу, например, текстовому полю.

Новые элементы в контекстное меню можно добавить в режиме дизайнера:



При этом мы можем добавить все те же элементы, что и в MenuStrip. Но, как правило, использует ToolStripMenuItem, либо элемент ToolStripSeparator, представляющий горизонтальную полоску разделитель между другими пунктами меню.

Либо на панели свойств можно обратиться к свойству Items компонента ContextMenuStrip и в открывшемся окне добавить и настроить все элементы меню:



Теперь создадим небольшую программу. Добавим на форму элементы ContextMenuStrip и TextBox, которые будут иметь названия contextMenuStrip1 и textBox1 соответственно. Затем изменим код формы следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | public partial class Form1 : Form  {      string buffer;      public Form1()      {          InitializeComponent();            textBox1.Multiline = true;          textBox1.Dock = DockStyle.Fill;            // создаем элементы меню          ToolStripMenuItem copyMenuItem = new ToolStripMenuItem("Копировать");          ToolStripMenuItem pasteMenuItem = new ToolStripMenuItem("Вставить");          // добавляем элементы в меню          contextMenuStrip1.Items.AddRange(new[] { copyMenuItem, pasteMenuItem });          // ассоциируем контекстное меню с текстовым полем          textBox1.ContextMenuStrip = contextMenuStrip1;          // устанавливаем обработчики событий для меню          copyMenuItem.Click += copyMenuItem\_Click;          pasteMenuItem.Click += pasteMenuItem\_Click;      }      // вставка текста      void pasteMenuItem\_Click(object sender, EventArgs e)      {          textBox1.Paste(buffer);      }      // копирование текста      void copyMenuItem\_Click(object sender, EventArgs e)      {          // если выделен текст в текстовом поле, то копируем его в буфер          buffer = textBox1.SelectedText;      }  } |

В данном случае выполнена простейшая реализация функциональности copy-paste. В меню добавляется два элемента. А у текстового поля устанавливается многострочность, и оно растягивается по ширине контейнера.

У многих компонентов есть свойство ContextMenuStrip, которое позволяет ассоциировать контекстное меню с данным элементом. В случае с TextBox ассоциация происходит следующим образом: textBox1.ContextMenuStrip = contextMenuStrip1. И по нажатию на текстовое поле правой кнопкой мыши мы сможем вызвать ассоциированное контекстное меню.

С помощью обработчиков нажатия пунктов меню устанавливаются действия по копированию и вставке строк.

