Часть 1. Визуальное программирование в C#.

Лабораторная работа №1.

Тема: *Работа с компонентами ListBox, ComboBox, RadioButton, CheckBox, GroupBox, Panel.*

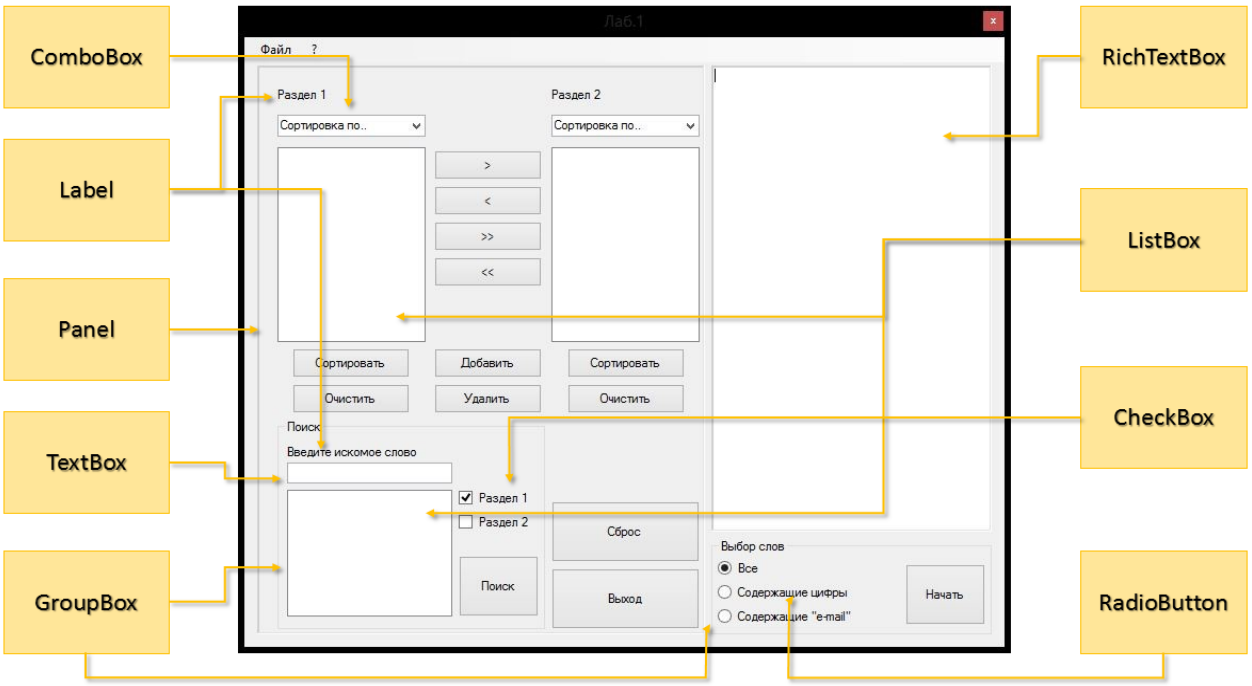


Рисунок 1.

Требования к работе:

1) Создать визуальную часть, используя необходимые компоненты. Все нужные компоненты указаны на рисунке 1. Их расположение и общий вид формы может выбираться самостоятельно.

2) Приложение должно позволить пользователю открыть текстовый файл, который считается в RichTextBox. Далее, пользователь может выбрать критерий по которому он хочет отобрать слова: «Все», «Содеражащие цифры», «Содержащие ‘e-mail’. После нажатия на кнопку «Начать», текст в RichTextBox’е разбивается на слова, которые в свою очередь, заносятся в ListBox (Раздел 1), по заданному критерию. Между двумя разделами имеется панель, в которой находятся 4 кнопки, посредством которых можно переносить отдельные выбранные слова, либо всю коллекцию из одного ListBox’а в другой, а также кнопки «Добавить» и «Удалить», которые соответственно реализуют добавление/удаление элементов из разделов.

Также каждый раздел можно очистить, либо отсортировать четырьмя способами: по длине (возр.), по длине (убыв.), по алфавиту (возр.) и соответственно по алфавиту (убыв.). В нижнем правом углу находится блок, отвечающий за поиск строк в разделах. Должна быть также реализована возможность сохранения содержимого из Раздела 2 в текстовый файл.

3) Реализовать Сортировку разделов, любым известным алгоритмом сортировки самостоятельно.

Задание 1. Создание визуальной части приложения.

1) Перенесите на форму все необходимые элементы из Панели элементов, чтобы сделать форму, показанную на рисунке 1.

2) Создайте меню.

Порядок действий:

1. Расположите все компоненты, как показано на рисунке 1, либо в произвольном порядке.

2. У компонента Panel установите значение свойства BorderStyle (Fixed3D – выпуклая, утопленная). У компонентов RadioButton, CheckBox и ComboBox установите начальные значения свойств Checked, Checked и Text, как показано на рисунке 1, соответственно. Измените свойство SelectionMode у ListBox'ов на MultiExtended. Также добавьте в свойство Items обоих компонентов ComboBox, четыре строки:

Алфавиту (по возрастанию)

Алфавиту (по убыванию)

Длине слова (по возрастанию)

Длине слова (по убыванию)

3. Создание меню.

а) Перенесите на форму компонент MenuStrip.

б) Создайте меню по типу, показанному на рисунке 2. Для добавления пунктов/подпунктов просто, нажимайте на квадратные области в месте, где установлено меню, и вводите необходимый текст. Установите «горячие клавиши»: нажмите на необходимый элемент меню, например «Открыть»,

далее перейдите в свойства компонентов и в свойстве «ShortcutKeys» установите необходимые сочетания.

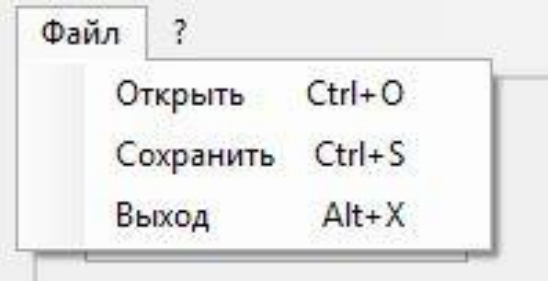


Рисунок 2.

Задание 2. Программирование элементов.

1) Меню.

2) Обработчики нажатий на кнопки.

Порядок действий:

1. Меню.

а) Реализуйте открытие текстового файла. Для того, чтобы открыть и записать текстовый файл в RichTextBox, необходимо изначально в обработчике события «Click» элемента «Открыть» создать объект класса OpenFileDialog.

```
OpenFileDialog OpenDlg = new OpenFileDialog();
```

Далее, если в диалоговом окне, пользователь нажмёт на кнопку «ОК», то нужно считать выбранный файл в RichTextBox. Для этого мы создаём объект класса StreamReader, параметрами которого будут являться Имя выбранного файла и стандартная кодировка. Считывание производится с помощью метода ReadToEnd(), который считывает текстовый файл от начала до конца в необходимое местоположение.

```
if (OpenDlg.ShowDialog() == DialogResult.OK)
{
       StreamReader Reader = new StreamReader(OpenDlg.FileName, Encoding.Default);
       richTextBox1.Text = Reader.ReadToEnd();
       Reader.Close();
}

OpenDlg.Dispose();
```

б) Сохранение файла производится аналогично, только теперь мы создаём объект класса StreamWriter, и из ListBox’а построчно заносим в файл наши слова:

```
if (SaveDlg.ShowDialog() == DialogResult.OK
{
	StreamWriter Writer = new StreamWriter(SaveDlg.FileName);

	for (int i = 0; i < listBox2.Items.Count; i++)
	{
		Writer.WriteLine((string)listBox2.Items[i]);
	}

	Writer.Close();
}

SaveDlg.Dispose();
```

в) Выход из приложения: `Application.Exit();`

г) Информационное сообщение: `MessageBox.Show("Информация о приложении и разработчике");`

2. Обработчики нажатий на кнопки.

а) Кнопка «Начать». Как было уже сказано выше, при нажатии на кнопку начать, нам необходимо разбить считанный текст на слова, т.е. убрать пробелы, знаки табуляции и переходы на новую строку. Для удобства мы заносим текст в массив строк, из которого, в свою очередь, отбираем необходимые строки по 3-ём критериям. По первому критерию мы заносим абсолютно все строки в список, делается это с помощью метода `listBox1.Items.Add`, параметром для которого соответственно является строка, которую мы хотим добавить. Для того, чтобы реализовать следующие два критерия нам необходимо добавить ещё одно пространство имён `using System.Text.RegularExpressions`. В нём имеется класс Regex, с помощью одного из методов которого мы и будем искать вложенные подстроки: `(Regex.IsMatch(Str, @"\d"))`. Т.е., как очевидно, метод IsMatch, ищет в строке Str некоторые подстроки, и если в какой либо строке находится подстрока, удовлетворяющая условиям поиска, то он возвращает true. Пример работы кнопки «Начать» показан ниже.

```
listBox1.Items.Clear();
listBox2.Items.Clear();

listBox1.BeginUpdate();

string[] Strings = richTextBox1.Text.Split(new char[] { '\n', '\t', ' ' },
StringSplitOptions.RemoveEmptyEntries);

foreach (string s in Strings)
{
	string Str = s.Trim();

	if (Str == String.Empty) continue;
	if (radioButton1.Checked) listBox1.Items.Add(Str);
	if (radioButton2.Checked)
```

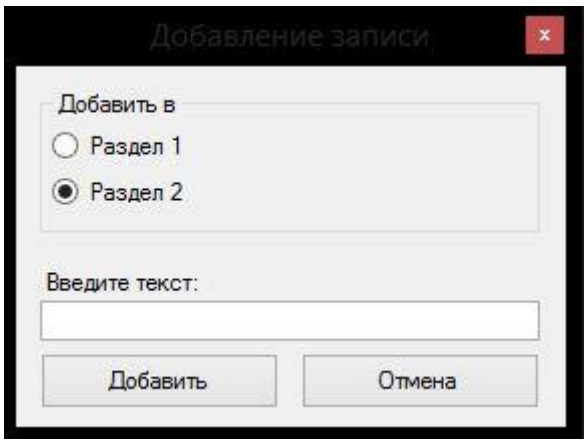
```
        {
            if (Regex.IsMatch(Str, @"\d")) listBox1.Items.Add(Str);
        }
        if (radioButton3.Checked)
        {
            if (Regex.IsMatch(Str, @"\w+@\w+\.\w+")) listBox1.Items.Add(Str);
        }
}

listBox1.EndUpdate();
```

б) Кнопки «Выход», «Сброс», «Очистить». Запрограммируйте кнопку «Выход» соответственно для закрытия приложения. «Сброс» означает, что наша форма должна вернуться к первоначальному виду, т.е. нужно очистить ListBox'ы (метод ListBox.Items.Clear()), поле Text у RichTextBox'а и TextBox'а, установить изначальное состояние свойства Checked у RadioButton и CheckBox.

в) «Поиск». Для поиска подстрок в строках, можно воспользоваться методом Contains класса string. Пример работы функции поиска:

```
            listBox3.Items.Clear();

            string Find = textBox1.Text;

            if (checkBox1.Checked)
            {
                foreach (string String in listBox1.Items)
                {
                    if (String.Contains(Find)) listBox3.Items.Add(String);
                }
            }

            if (checkBox2.Checked)
            {
                foreach (string String in listBox2.Items)
                {
                    if (String.Contains(Find)) listBox3.Items.Add(String);
                }
            }
```

г) Кнопки «Добавить» и «Удалить». При нажатии на кнопку добавить у нас должна открыться новая модульная форма (модульная, значит, что пока она открыта, мы не можем взаимодействовать с основной формой). Чтобы создать новую форму, нужно в Обозревателе решений нажать правой кнопкой на наш проект и: Добавить –> Создать элемент.., где в открывшемся окне выбираем «Форма Windows Forms». Данная форма должна выглядеть примерно так:



Для того, чтобы связать две формы нужно в обработчике для кнопки «Добавить» (которая находится в ГЛАВНОЙ ФОРМЕ) вписать такой код:

```
Form2 AddRec = new Form2();

AddRec.Owner = this;
AddRec.ShowDialog();
```

В первой строке, мы создали объект AddRec, т.е. создали форму. Далее, нам необходимо указать, что «родителем» этой новой формы является наша главная форма (свойство Owner). Затем мы соответственно открываем созданную форму, используя метод ShowDialog, который собственно и делает эту форму модульной.

Теперь, в форме 2, создайте обработчик нажатия на кнопку «Добавить» и пропишите такой код:

```
Form1 Main = this.Owner as Form1;

if (textBox1.Text != "")
{
    if (this.radioButton1.Checked == true)
    Main.listBox1.Items.Add(this.textBox1.Text);
    else Main.listBox2.Items.Add(this.textBox1.Text);

    this.Close();
}
```

В первой строке, мы создаём объект Main класса Form1(наша основная форма) и указываем, что Main, и есть «родитель» формы 2. Наконец, мы связали наши формы, но подобная связка позволит им взаимодействовать только при условии, что мы «откроем» необходимые поля. Для этого, опять же, в Обозревателе решений откройте файл Form1.Designer.cs, найдите блок кода, где были созданы ListBox’ы, и вместо private, присвойте им уровень доступа public. Теперь мы можем из второй формы обращаться к спискам первой. Создайте обработчик для нажатия кнопки Отмена, которая закрывает Форму 2.

Кнопка «Удалить» позволит нам удалить выбранные строки из разделов. Целесообразно создать отдельную функцию, например DeleteSelectedStrings, в которую и передавать нужный ListBox, в зависимости от выбранного для удаления раздела:

```
for (int i = ListBox.Items.Count - 1; i >= 0; i--)
{
      if (ListBox.GetSelected(i)) ListBox.Items.RemoveAt(i);
}
```

д) Кнопки переноса строк.

Для переноса строк из одного раздела в другой необходимо воспользоваться методом ListBox.Items.Add(строка из другого лист бокса), для обращения к отдельным строкам: ListBox.Items.[..]. Не забудьте после переноса строки удалять строки из исходного раздела, методы Remove() и RemoveAt(), в первый посылается в параметры объект, а во второй индекс нужной строки соответственно. Для обращения к выделенным строкам: ListBox.SelectedItems. Пример кода для переноса всей коллекции из первого списка во второй:

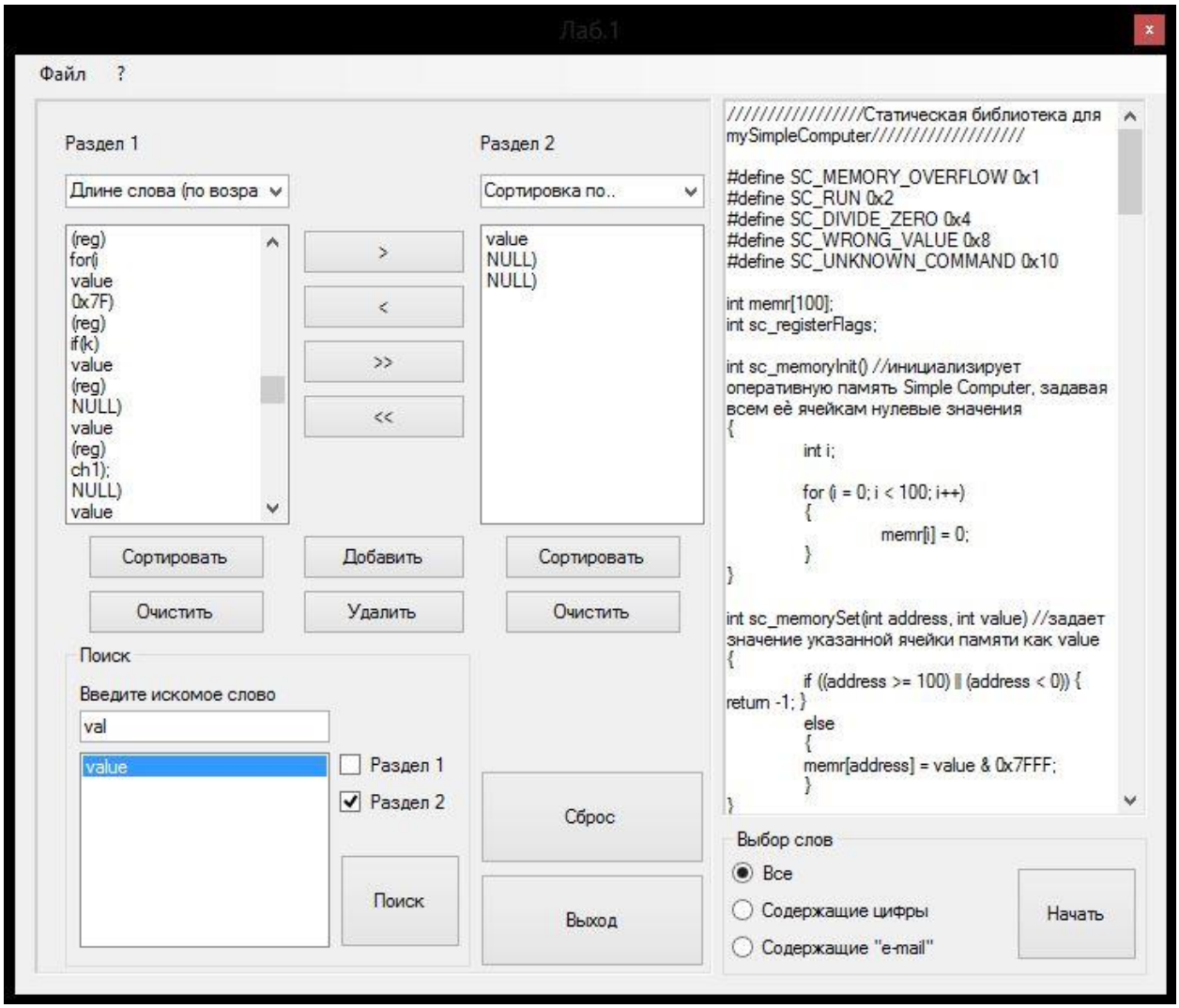
```
ListBoxTo.Items.AddRange(ListBoxFrom.Items);
ListBoxFrom.Items.Clear();
```

Перенос выделенных строк:

```
ListBoxTo.BeginUpdate();

foreach (object Item in ListBoxFrom.SelectedItems)
{
      ListBoxTo.Items.Add(Item);
}

ListBoxTo.EndUpdate();
```

е) «Сортировать». Напишите самостоятельно функции для сортировки разделов с помощью любых вам известных алгоритмов сортировки. *Подсказка: у ListBox'а есть свойство Sort, которое располагает элементы списка в алфавитном порядке. Для реализации сортировок «по убыванию» можно перенести все элементы списка в некий массив, у которого есть метод Reverse(). Для сортировки по длине, можно написать отдельную функцию, которая будет сортировать элементы списка в неком массиве, затем очищать список и заносить в него уже отсортированные элементы.*

Пример готовой программы:



Лабораторная работа №2.

Тема: Создание графического редактора, позволяющего:

- Создавать, редактировать, загружать, сохранять изображения;
- Рисовать с помощью мыши (при нажатии левой кнопки мыши и её перемещении отображается кривая движения указателя мыши. При нажатии правой кнопки мыши появляется стирательная резинка);
- Задавать цвет, толщину и стиль линии;
- Пользоваться историей изменений в обе стороны – undo и redo.

Компоненты: *MenuStrip, ToolStrip, Panel, ColorDialog, OpenFileDialog, SaveFileDialog, PictureBox, ImageList, TrackBar, ComboBox*.

Теоритические сведения:

Компонент MenuStrip. Для быстрого вызова команд можно использовать так называемые быстрые клавиши. Для этого надо установить свойство ShowShortCutKeys, выбрав значение True. Также установить свойство ShortCutKeys, выбрав значение из списка (или набрать). При этом нужно следить, чтобы быстрые клавиши не повторялись во избежание коллизий.

Можно использовать любые готовые иконки либо создать их самостоятельно. Для этого в свойствах необходимо найти Image и дважды нажать на значение свойства, появится окно «Выбор ресурса». В окне выберете контекст ресурса (Локальный или Файл ресурсов проекта). Локальный – если вы хотите установить собственную иконку, Файл ресурсов проекта – если вас устраивают стандартные иконки (windows theme).

Компонент ToolStrip. Представляет собой специальный контейнер для создания панелей инструментов. Может управлять любыми вставленными в него дочерними элементами: группировать, выравнивать по размерам, располагать элементы в несколько рядов.

Специально для ToolStripPanel разработан компонент ToolStripButton (кнопка панели инструментов, отсутствует в палитре компонентов). Для добавления в панель компонента ToolStripButton надо: щелкнуть правой кнопкой мыши на ToolStripPanel и выбрать Button|Label|SplitButton|DropDownButton|Separator|ComboBox|TextBox|ProgressBar.

На кнопки можно поместить изображения. Для этого надо установить свойство Image.

Компонент PictureBox. Служит для размещения на форме одного из трех поддерживаемых типов изображений: растрового изображения, значка и метафайла.

Растровое изображение – это произвольные графические изображения в файлах со стандартным расширением .bmp. Значки (иконки) – небольшие растровые изображения, снабженные специальными средствами, регулирующими их прозрачность. Расширение файлов значков, обычно, .ico . (Метафайл – это изображение, построенное на графическом устройстве с помощью специальных команд, которые сохраняются в файле с расширением .wmf .)

Компонент TrackBar – ползунок. Для плавного изменения числовой величины (во многом схож с компонентом ScrollBar).

Некоторые свойства:
- Maximum – определяет максимальное значение диапазона изменения;
- Minimum – определяет минимальное значение диапазона изменения;
- Value: integer – определяет текущее положение ползунка;
- Orientation – ориентация компонента (горизонтальная, вертикальная);

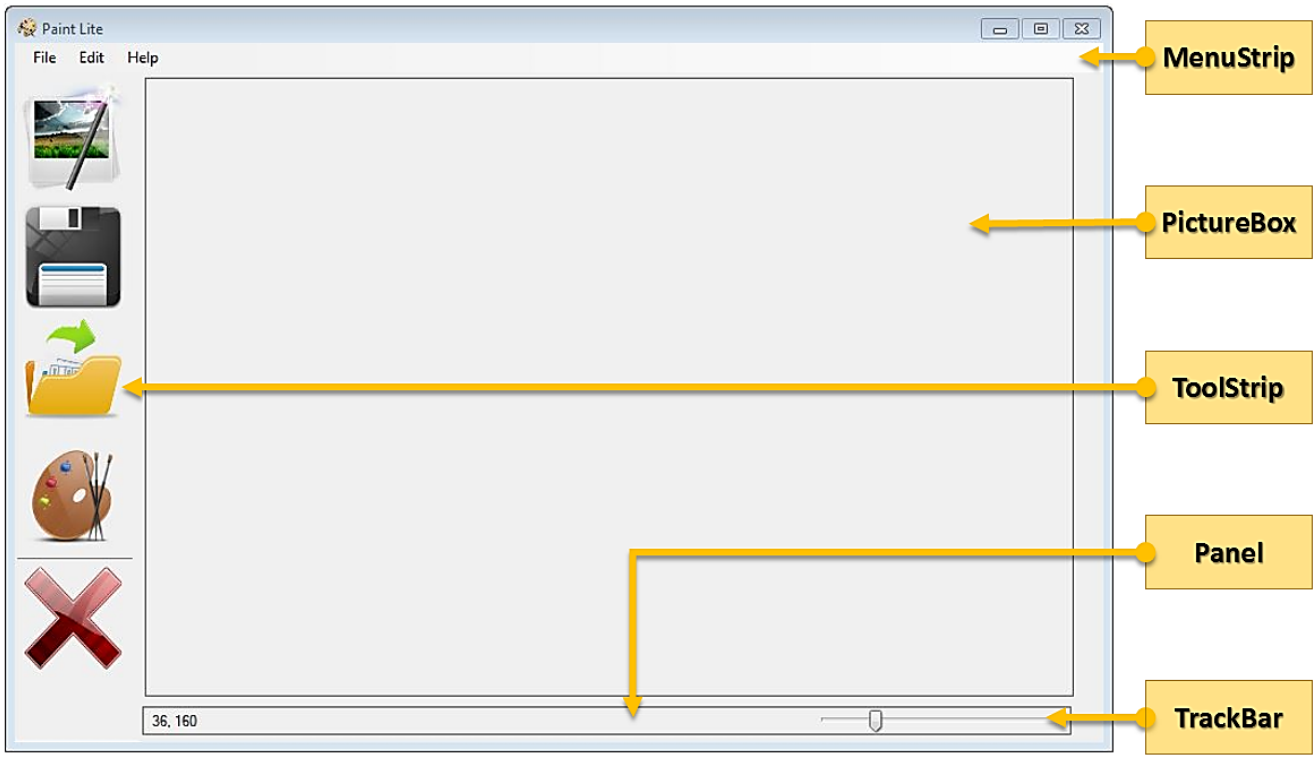
События мыши. Для выполнения какого-либо действия с помощью щелчка левой кнопки мыши для большинства случаев достаточно запрограммировать обработчик событий OnClick, для реакции на двойной щелчок используется событие OnDbClick. Для более совершенного управления мышью лучше использовать обработчики следующих событий:

- OnMouseDown. Вызывается при нажатии любой кнопки мыши.
- OnMouseMove. Вызывается при перемещении мыши.
- OnMouseUp. Вызывается при отпускании какой-нибудь кнопки мыши.

Процедуры обработки этих событий получают следующие параметры:

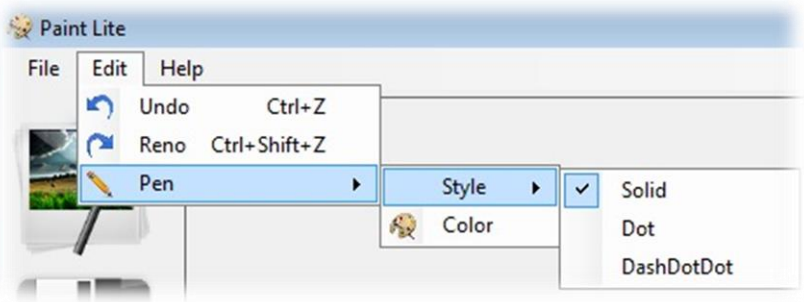
- Sender. Представляет объект, который получил это событие (на каком объекте щелкнули мышью).
- Button. Имеет одно из трех значений: MouseButtons.Right, MouseButtons.Left, MouseButtons.Middle. Используется для определения того, какую кнопку мыши нажал пользователь.
- X, Y. Координаты указателя мыши в пикселях относительно клиентной области окна с координатами (0,0) в верхнем левом углу.

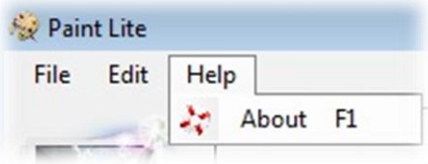
Задание 1. Создание формы.



Создание визуальной части.

1. Для начала создаем главное меню следующего вида:





Не забываем назначить все горячие клавиши и выставить по умолчанию свойство *Checked = true* у *Pen->Style->Solid*, как показано на скриншоте.

2. Затем создаем форму *ToolStrip* и помещаем туда продублированные команды New, Open, Save, Color и Exit, либо другие пункты на собственное усмотрение.

3. Добавляем управление толщиной пера и вывод текущих координат. Для этого помещаем в удобное место (например под *PictureBox*) *Panel*, *Label* и *TrackBar*, как показано на скриншоте. *Label* будем использовать один общий для двух координат, чтобы не было их смещения, как могло бы быть при использовании двух независимых *Label* отдельно для разделения координат *X* и *Y*.

Создание невизуальной части.

Условимся на том, что PictureBox будет называться picDrawingSurface.

Для начала необходимо запрограммировать работу всех пунктов меню у File и Help. В пункте меню Help можно указать версию программы, имя разработчика и возможности программы. Это творческое задание.

Создание нового файла. Ограничиться только одним перемещением PictureBox на форму для рисования в нем не получится. Ведь там рисовать будет нельзя, даже если очень захочется. Ведь свойство *Image* у *PictureBox* не инициализируется от одного добавления последнего на форму и при попытке что-либо чиркнуть в этом поле в откомпилированной программе – появится ошибка *System.ArgumentNullException* или подобная ей. Поэтому пункт меню New и будет сводиться к следующим строчкам кода:

```
Bitmap pic = new Bitmap(750, 500);
picDrawingSurface.Image = pic;
```

Размеры *new Bitmap* определяем из размеров *PictureBox*, помещенного в форму. Теперь в таком поле появится возможность рисовать.

Фича #1. Для того, чтобы пользователь по ошибке не начал рисовать в неинициализированной области *PictureBox* (ведь при запуске программы у нас еще ничего не готово) создадим своеобразную «защиту»: при попытке рисовать на неинициализированном *PictureBox* будет всплывать сообщение с предупреждением «Сначала создайте новый файл!» и дальнейшее предотвращение попытки рисования. Для этого нужно зайти в события PictureBox и выбрать событие *MouseDown*. Дважды щелкаем по нему и в открывшимся редакторе кода прописываем следующие строчки:

```
            if (picDrawingSurface.Image == null)
            {
                MessageBox.Show("Сначала создайте новый файл!");
                return;
            }
```

Если поле для рисования *Image* не инициализировано (равно *null*), то выводим предупреждающее пользователя сообщение и выходим из функции. Таким образом будет предотвращена попытка рисования в неинициализированной области и программа не вылетит с ошибкой.

Для возможности сохранить изображение будем использовать так же, как и в предыдущей лабораторной работы *SaveFileDialog*. Здесь так же создаем объект *SaveDlg* и прописываем ему параметры:

```
            SaveFileDialog SaveDlg = new SaveFileDialog();
            SaveDlg.Filter = "JPEG Image|*.jpg|Bitmap Image|*.bmp|GIF Image|*.gif|PNG
Image|*.png";
            SaveDlg.Title = "Save an Image File";
            SaveDlg.FilterIndex = 4;    //По умолчанию будет выбрано последнее расширение
*.png

            SaveDlg.ShowDialog();
```

Теперь нужно прописать код для сохранения картинки в различных расширениях. Для этого используем *switch*.

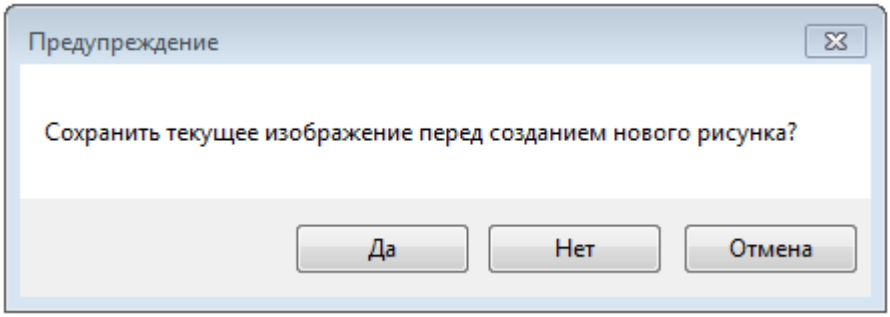
```
if (SaveDlg.FileName != "")     //Если введено не пустое имя
            {
                System.IO.FileStream fs =
                   (System.IO.FileStream)SaveDlg.OpenFile();

                switch (SaveDlg.FilterIndex)
                {
                    case 1:
                        this.picDrawingSurface.Image.Save(fs, ImageFormat.Jpeg);
                        break;

                    case 2:
                        this.picDrawingSurface.Image.Save(fs, ImageFormat.Bmp);
                        break;

                    case 3:
                        this.picDrawingSurface.Image.Save(fs, ImageFormat.Gif);
                        break;

                    case 4:
                        this.picDrawingSurface.Image.Save(fs, ImageFormat.Png);
                        break;
                }

                fs.Close();
            }
```

Фича #2

Если вдруг пользователь захотел создать еще один новый файл, при условии, что уже было что-то нарисовано в *PictureBox*, можно будет ему предложить сохранить текущее изображение, чтобы оно не потерялось. Ведь создание нового файла ведет к удалению предыдущего. Для этого потребуется следующий кусок кода:

```
if (picDrawingSurface.Image != null)
            {
                var result = MessageBox.Show("Сохранить текущее изображение перед
созданием нового рисунка?", "Предупреждение", MessageBoxButtons.YesNoCancel);

                switch (result)
                {
                    case DialogResult.No: break;
                    case DialogResult.Yes: saveMenu_Click(sender, e); break;
                    case DialogResult.Cancel: return;
                }
            }
```

Здесь, как и в Фиче #1, мы используем условный оператор для того, чтобы узнать, инициализирована ли форма *PictureBox* или нет. Если она уже была инициализирована, значит нужно предложить пользователю сохранить изображение. Данный *MessageBox* будет иметь 3 кнопки: *Yes*, *No* или *Cancel*. В зависимости от выбора пользователя будем решать, как поступить: отменить создание нового файла (*Cancel*), сохранить данное изображение (*Yes*) или не сохранять (*No*). Данный *MessageBox* будет выглядеть следующим образом:



Вставить данный кусок кода нужно непосредственно в метод, ответственный за пункт меню New.

Теперь перейдем непосредственно к Open. По тому же принципу, как и в Save, создаем объект класса *OpenFileDialog* и прописываем ему ряд параметров:

```
OpenFileDialog OP = new OpenFileDialog();
            OP.Filter = "JPEG Image|*.jpg|Bitmap Image|*.bmp|GIF Image|*.gif|PNG
Image|*.png";
            OP.Title = "Open an Image File";
            OP.FilterIndex = 1; //По умолчанию будет выбрано первое расширение *.jpg
```

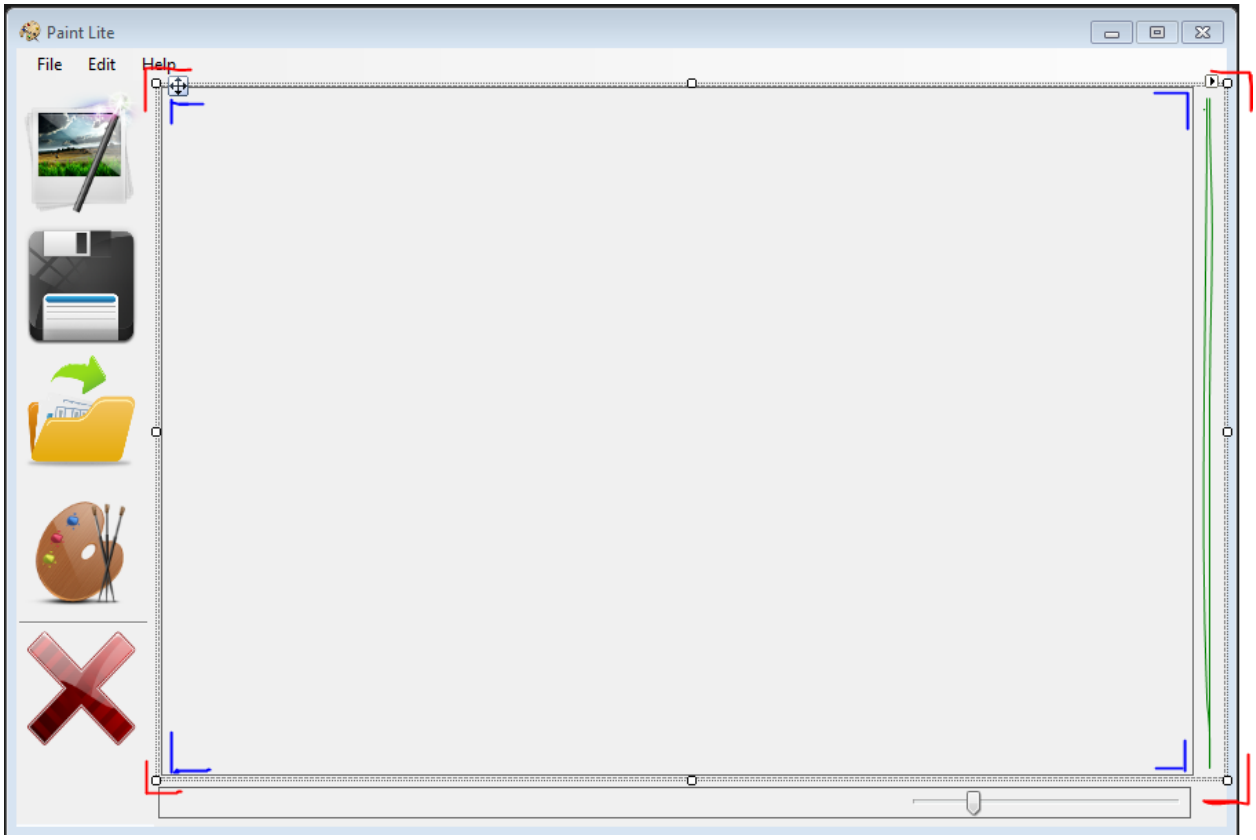
И, когда пользователь укажет нужный путь к картинке, ее нужно будет загрузить в *PictureBox*:

```
if (OP.ShowDialog() != DialogResult.Cancel)
                picDrawingSurface.Load(OP.FileName);

            picDrawingSurface.AutoSize = true;
```

Фича #3.

Если пользователь захочет загрузить картинку, размеры которой превышают размеры самого окна *PictureBox*, то возникнет одна проблема. Во-первых,

загруженная картинка не сожмется до размеров окна *PictureBox*, а откроется в полном размере, но видно будет лишь та часть, которая влезет в окно. Во-вторых, у пользователя не будет возможности перемещать данную картинку во всех направлениях для ее просмотра и изменения. Для решения этой проблемы можно воспользоваться следующей хитростью: нужно поместить под *PictureBox Panel*. С помощью *Panel* у пользователя появится возможность двигать картинку во всех направлениях в том случае, если ее размер будет превышать размеры окна *PictureBox*.



Здесь красными уголками выделен элемент *Panel*, синими – *PictureBox*, а зеленой двойной линией показан отступ, который нужно оставить для вертикальной полосы прокрутки, который появится, как только длинна загруженной картинки станет больше длинны *PictureBox*. Для горизонтальной полосы прокрутки место оставлять не нужно, она появится в нужном месте сама.

5. Самостоятельно запрограммируйте весь *ToolStrip*. Для этого достаточно продублировать все методы из *MenuStrip*, которые вы продублировали на свой выбор.

6. Переходим непосредственно к рисованию. Для начала пропишем ряд глобальных переменных:

```
        bool drawing;
        GraphicsPath currentPath;
        Point oldLocation;
```

```
        Pen currentPen;
```

И пропишем пару строк в конструкторе формы:

```
public Form1()
        {
            InitializeComponent();
            drawing = false;                    //Переменная, ответственная за рисование
            currentPen = new Pen(Color.Black);  //Инициализация пера с черным цветом
        }
```

Так же для рисования нам потребуется добавить еще 2 события помимо *MouseDown – MouseUp* и *MouseMove*.

MouseDown отвечает за нажатую кнопку мыши, *MouseUp* – за отпущенную, а *MouseMove* соответственно за перемещение мыши. Алгоритм будет следующий: при нажатии клавиши мы активируем режим рисования (для этого и нужна переменная типа *bool*, названная *drawing*, которая в момент нажатия мыши будет принимать значение *true*, а в момент отпускания *false*). Тогда, при нажатии мыши мы активируем режим рисования, а в момент отпускания деактивируем. Дополним код *MouseDown*:

```
        private void picDrawingSurface_MouseDown(object sender, MouseEventArgs e)
        {
            if (picDrawingSurface.Image == null)
            {
                MessageBox.Show("Сначала создайте новый файл!");
                return;
            }
            if (e.Button == MouseButtons.Left)
            {
                drawing = true;
                oldLocation = e.Location;
                currentPath = new GraphicsPath();
            }
        }
```

и пропишем код для *MouseUp*:

```
        private void picDrawingSurface_MouseUp(object sender, MouseEventArgs e)
        {
            drawing = false;
            try
            {
                currentPath.Dispose();
            }
            catch { };
        }
```

При каждом нажатии мыши мы будем выделять память для *currentPath* под *GraphicsPath* (этот класс представляет последовательность соединенных линий и кривых), ну, а чтобы не было переполнения памяти, в *MouseUp* будем удалять объект *currentPath*.

Само рисование будет происходить в *MouseMove*. Пропишем это событие:

```
        private void picDrawingSurface_MouseMove(object sender, MouseEventArgs e)
```

```
        {
            if (drawing)
            {
                Graphics g = Graphics.FromImage(picDrawingSurface.Image);
                currentPath.AddLine(oldLocation, e.Location);
                g.DrawPath(currentPen, currentPath);
                oldLocation = e.Location;
                g.Dispose();
                picDrawingSurface.Invalidate();
            }
        }
```

Таким образом, только в случае, если *drawing* будет равное *true* (клавиша мыши нажата), то будет происходить рисование. С помощью данного кода за нажатой мышкой будет рисоваться линия. Теперь можно протестировать программу.

7. Ластик по нажатию правой кнопки мыши. Для самостоятельного решения.

Алгоритм будет следующий. Сначала нужно создать новую глобальную переменную `Color historyColor`. В нее нужно будет сохранять текущий цвет пера при нажатии на правую клавишу мыши в событии *MouseDown*. Далее, в том же самом событии для правой клавиши мыши нужно будет менять цвет пера с текущего на белый в переменной *currentPen*. А в событии *MouseUp* нужно возвращать цвет пера, который был до использования ластика с помощью переменной *historyColor*.

8. Привязываем *TrackBar* для толщины пера и выводим текущие координаты.

Для начала пропишем код для вывода текущих координат. Как уже было сказано выше, мы будем использовать лишь один *Label*, который расположили в *Panel* под *PictureBox*. Код будем прописывать в событии *MouseMove*, что достаточно очевидно. Итак, дополним *MouseMove* следующей строчкой:

```
label_XY.Text = e.X.ToString() + ", " + e.Y.ToString();
```

Здесь *label_XY* – это наш *Label*, (у вас по умолчанию он будет называться label1). Но для информативности в пределах данной работы он был переименован. *Label* приминает значение *string*. Параметр `MouseEventArgs e` передаст координаты *X* и *Y*, если к нему обратиться как *e.X* и *e.Y*, соответственно. По умолчанию они (координаты) имеют целочисленный тип, поэтому к ним мы добавляем через точку метод *ToString()*. Операцией сложения мы добавляем разделение двух координат через запятую. Таким образом при движении мыши будут постоянно обновляться координаты и выводиться в виде *X, Y*, где под *X* и *Y* будут подставлены текущие координаты положения мыши в поле *PictureBox*. Эту строчку нужно вставить в самом начале метода *picDrawingSurface_MouseMove* перед условным

оператором, т.к. координаты нас будут интересовать и без нажатой кнопки мыши.

Теперь перейдем к подключению *TrackBar*. У него есть 3 интересующих нас свойства: *Maximum*, *Minimum* и *Value*. В данной работе, в *Maximum* задано значение 20, в *Minimum* 1, а в *Value* 5. Значения заданы в конструкторе форм, хотя их можно прописать в ручную в конструкторе класса *Form1*. При движении ползунка *TrackBar* будет меняться значение *Value* от *Minimum* до *Maximum*. Именно *Value* нас и будет интересовать. Дополним код конструктора Form1:

```
public Form1()
        {
            InitializeComponent();

            drawing = false;                //Переменная, ответственная за рисование
            currentPen = new Pen(Color.Black);      //Инициализация пера с черным цветом
            currentPen.Width = trackBarPen.Value;   //Инициализация толщины пера
        }
```

И, щелкнув по *TrackBar* дважды в конструкторе форм, пропишем код:

```
private void trackBar1_Scroll(object sender, EventArgs e)
        {
            currentPen.Width = trackBarPen.Value;
        }
```

Все готово, можно откомпилировать и проверить новые возможности вашего графического редактора.

9. История изменений Undo и Redo.

Переходим к самой интересной части. Идея такая: создать некоторый динамический список, в котором будет хранить последние 10 изменений в редакторе (рисование, стирание). Можно сделать другое количество изменений, но предупреждаю, каждое такое изменение будет отщипывать у оперативной памяти порядка 2 мб. В пределах данной методички мы не будем рассматривать более оптимизированные способы хранения истории, т.к. важно понять сам принцип работы с нею. Итак, приступим.

Для начала нужно определиться какой именно динамический массив (список) мы будем использовать. В данной работе был выбран *List*. А так же нам потребуется переменная-счетчик для истории. Дополним поля класса этими переменными:

```
        bool drawing;
        int historyCounter;      //Счетчик истории

        GraphicsPath currentPath;
        Point oldLocation;
 public Pen currentPen;
        Color historyColor;     //Сохранение текущего цвета перед использованием ластика
        List<Image> History;    //Список для истории
```

Наш список будет хранить переменные типа *Image*, т.к. область, в которой мы рисуем в *PictureBox* так же является объектом этого типа.

В конструкторе формы нужно выделить память под данный список:

```
History = new List<Image>();            //Инициализация списка для истории
```

Теперь алгоритм следующий. При создании нового файла нужно занести только что созданное чистое поле в историю. Далее, при каждом отпускании левой клавиши мыши нужно заносить в список истории только что нарисованную область. Также нужно следить за переполнением истории. В этом же событии *MouseUp* нужно сделать проверку на ее размер. Если в ней уже более 9 элементов, то удалять первый. А также нужно очищать историю, как только пользователь захотел создать новый файл.

Дополним метод создания нового файла:

```
        private void newToolStripMenuItem_Click(object sender, EventArgs e)
        {
            History.Clear();
            historyCounter = 0;
            Bitmap pic = new Bitmap(750, 500);
            picDrawingSurface.Image = pic;
            History.Add(new Bitmap(picDrawingSurface.Image));
        }
```

Дополним событие *MouseUp*:

```
        private void picDrawingSurface_MouseUp(object sender, MouseEventArgs e)
        {
            //Очистка ненужной истории
            History.RemoveRange(historyCounter + 1, History.Count - historyCounter - 1);
            History.Add(new Bitmap(picDrawingSurface.Image));
            if (historyCounter + 1 < 10) historyCounter++;
            if (History.Count - 1 == 10) History.RemoveAt(0);
            drawing = false;
            try
            {
                currentPath.Dispose();
            }
            catch { };
        }
```

В событии MouseUp не забываем, что должны быть еще пару строк для ластика, которые нужно было прописать самостоятельно в пункте 7. Здесь они специально вырезаны.

А теперь запрограммируем пункты меню Undo:

```
private void undoToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (History.Count != 0 && historyCounter != 0)
            {
                picDrawingSurface.Image = new Bitmap(History[--historyCounter]);
            }
            else MessageBox.Show("История пуста");
        }
```

и Redo:

```
private void renoToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (historyCounter < History.Count - 1)
            {
                picDrawingSurface.Image = new Bitmap(History[++historyCounter]);
            }
            else MessageBox.Show("История пуста");
        }
```

10. Стиль пера.

У класса *Pen* есть несколько интересующих нас стилей: *Solid* (сплошная линия), *Dot* (линия, состоящая из точек) и *DashDotDot* (штрих-две точки). Стили будем менять в меню *Edit->Pen->Style*. Пропишем код для первого стиля *Solid*:

```
        private void solidToolStripMenuItem_Click(object sender, EventArgs e)
        {
            currentPen.DashStyle = DashStyle.Solid;

            solidStyleMenu.Checked = true;
            dotStyleMenu.Checked = false;
            dashDotDotStyleMenu.Checked = false;
        }
```

Здесь первая строчка - задания стиля пера. Остальные 3 строчки нужны для установки галочки выбранного стиля в меню. Аналогично прописываем 2 других стиля.

Небольшое замечание.

При попытке сохранить файл в любом расширении, помимо *.png у пользователя будет сохранен черный квадрат. Это связано с тем, что по умолчанию PictureBox имеет прозрачный фон. Прозрачный фон поддерживает (в нашем случае) только формат png. Остальные форматы, не имеющие данной поддержки, заменяют прозрачный фон на черный. Линии, нарисованные черным пером сливаются с фоном и получается черный прямоугольник. Вот небольшой кусок кода, который может помочь решить проблему:

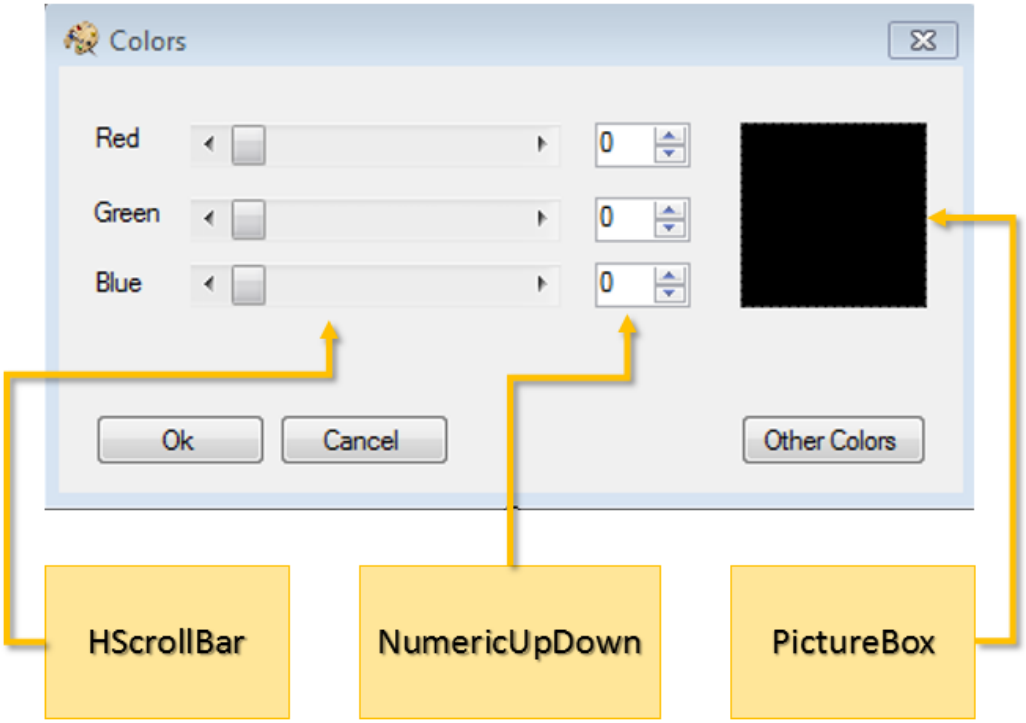
```
                Graphics g = Graphics.FromImage(picDrawingSurface.Image);

                g.Clear(Color.White);
                g.DrawImage(picDrawingSurface.Image, 0, 0, 750, 500);
```

Так же есть и другие варианты исправления этой ситуации. Всё это для самостоятельного изучения и исправления.

Лабораторная работа №3.

Тема: дополнение графического редактора, созданного в прошлой лабораторной работе.

Завершающим штрихом для нашего графического редактора будет отдельная форма для выбора цвета пера. Выглядеть она будет следующим образом:



Для того, чтобы ее добавить, нужно зайти в *проект->добавить форму Windows.*

1. Необходимо разместить все компоненты, как показано на рисунке.

2. Для компонентов *HScrollBar* нужно установить свойство *Minimum* в 0, а *Maximum* в 255. Это будет диапазон изменения оттенков цветов. Так же выставляем *LargeChage* в единицу, это будет величина прокрутки при щелчке на полосе. Точно такие же значения свойств прописываем и для *NumericUpDown*, только вместо *LargeChage* будет свойство *Increment*.

3. Теперь нам нужно связать *HScrollBar* и *NumericUpDown*. Для этого в конструкторе новой формы пропишем следующий код, который свяжет эти 2 компонента через свойство *Tag*:

```
public ColorsForm(Color color)
{
    InitializeComponent();

    Scroll_Red.Tag = numeric_Red;
    Scroll_Green.Tag = numeric_Green;
    Scroll_Blue.Tag = numeric_Blue;


    numeric_Red.Tag = Scroll_Red;
    numeric_Green.Tag = Scroll_Green;
    numeric_Blue.Tag = Scroll_Blue;

    numeric_Red.Value = color.R;
    numeric_Green.Value = color.G;
    numeric_Blue.Value = color.B;
}
```

Для отличия между собой все 6 компонентов были переименованы.

4. Т.к. задача заключается в связывании двух компонентов *HScrollBar* и *NumericUpDown*, то воспользуемся событием *ValueChanged* у обоих компонентов.

Для *HScrollBar*:

```
private void Scroll_Red_ValueChanged(object sender, EventArgs e)
{
    ScrollBar scrollBar = (ScrollBar)sender;
    NumericUpDown numericUpDown = (NumericUpDown)scrollBar.Tag;
    numericUpDown.Value = scrollBar.Value;
}
```

Для NumericUpDown:

```
private void numeric_Red_ValueChanged(object sender, EventArgs e)
{
    NumericUpDown numericUpDown = (NumericUpDown)sender;
    ScrollBar scrollBar = (ScrollBar)numericUpDown.Tag;
    scrollBar.Value = (int)numericUpDown.Value;
}
```

Аналогично прописывается еще 2 события для оставшихся двух *HScrollBar* и 2 события для двух оставшихся *NumericUpDown*.

5. Теперь нам потребуется одна глобальная для класса переменная *colorResult*:

```
Color colorResult;
```

А также отдельная функция (назовем ее *UpdateColor*), которая будет смешивать цвет на основании положения ползунков и закрашивать данным цветом *PictureBox*. Т.к. в данном *PictureBox* мы не будем рисовать, а будем использовать его лишь для определения цвета пера, то инициализировать его

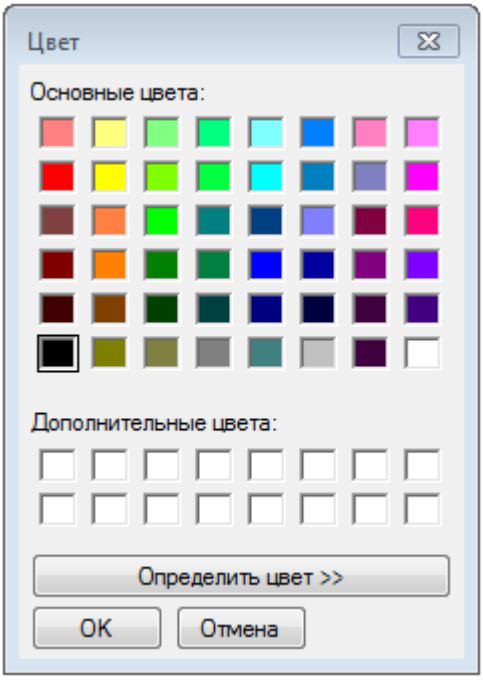
свойство Image не придется, как это делали в предыдущей части лабораторной работы. Для *UpdateColor* не принципиально откуда брать значения – из *HScrollBar* или из *NumericUpDown*, т.к. они теперь связаны между собой. Для примера взят именно *HScrollBar*:

```
private void UpdateColor()
        {
            colorResult = Color.FromArgb(Scroll_Red.Value, Scroll_Green.Value,
Scroll_Blue.Value);
            picResultColor.BackColor = colorResult;
        }
```

Вызывать ее можно либо в *HScrollBar*, либо в NumericUpDown, т.к. опять же не имеет значения, ведь после 4 пункта мы связали эти 2 компонента. Выбирается любой компонент и для всех трех его цветов вызывается функция *UpdateColor()* в любом месте.

Фича #1

В .Net есть готовое решение по выбору цвета. С ним думаю все знакомы, выглядит оно следующим образом:



Данное решение называется ColorDialog. Будем вызывать его по кнопке Other Colors. Помимо того, что будет появляться это окно, нужно будет еще привязать RGB выбранного цвета к в HScrollBar и NumericUpDown. Для этого будет служить следующий код:

```
private void buttonOther_Click(object sender, EventArgs e)
        {
            ColorDialog colorDialog = new ColorDialog();
            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                Scroll_Red.Value = colorDialog.Color.R;
                Scroll_Green.Value = colorDialog.Color.G;
                Scroll_Blue.Value = colorDialog.Color.B;

                colorResult = colorDialog.Color;

                UpdateColor();
            }
        }
```

6. Теперь переходим к связке двух форм. Связывать их будет лишь один параметр, который будет передаваться из формы 2 по нажатию на кнопку ***Ok*** в форму 1 – это созданный цвет пера *colorResult*. Существует много вариантов по передаче данных из одной формы в другую внутри одного проекта, разберем некоторые из них:

1. Изменение модификатора доступа
В Form2 установить модификатор доступа для контрола/поля public
В любом месте Form1

Код C#

```
1  Form2 f = new Form2();
2  f.ShowDialog();
3  this.textBox1.Text = f.textBox1.Text;
```

+ Самый быстрый в реализации и удобный способ
- Противоречит всем основам ООП
- Возможна передача только из более поздней формы в более раннюю
- Форма f показывается только с использованием ShowDialog(), т.е. в первую форму управление вернется только по закрытию второй. Избежать этого можно, сохранив ссылку на вторую форму в поле первой формы.

2. Использование открытого свойства/метода. Способ очень похож на первый
В классе Form2 определяем свойство (или метод)

Код C#

```
1  public string Data
2  {
```

Код C#

```
    get
    {
        return textBox1.Text;
    }
}
```

В любом месте Form1

Код C#

```
Form2 f = new Form2();
f.ShowDialog();
this.textBox1.Text = f.Data;
```

+ Противоречит не всем основам ООП
- Минусы те же

3. Передача данных в конструктор Form2

Изменяем конструктор Form2

Код C#

```
public Form2(string data)
{
    InitializeComponent();
    //Обрабатываем данные
    //Или записываем их в поле
    this.data = data;
}
string data;
```

А создаем форму в любом месте Form1 так:

Код C#

```
Form2 f = new Form2(this.textBox1.Text);
f.ShowDialog();
//Или f.Show();
```

+ Простой в реализации способ
+ Не нарушает ООП
- Возможна передача только из более ранней формы в более позднюю

4. Передача ссылки в конструктор

Изменяем конструктор Form2

Код C#

```
1  public Form2(Form1 f1)
2  {
3      InitializeComponent();
4      //Обрабатываем данные
5      //Или записываем их в поле
6      string s = f1.textBox1.Text;
7  }
```

А создаем форму в любом месте Form1 так, т.е. передаем ей ссылку на первую форму

Код C#

```
1  Form2 f = new Form2(this);
2  f.ShowDialog();
3  //Или f.Show();
```

\+ Доступ ко всем открытым полям/функциям первой формы
\+ Передача данных возможна в обе стороны
\- Нарушает ООП

5. Используем свойство 'родитель'

При создании второй формы устанавливаем владельца

Код C#

```
1  Form2 f = new Form2();
2  f.Owner = this;
3  f.ShowDialog();
```

Во второй форме определяем владельца

Код C#

```
1  Form1 main = this.Owner as Form1;
2  if(main != null)
3  {
4      string s = main.textBox1.Text;
5      main.textBox1.Text = "OK";
6  }
```

\+ Доступ ко всем открытым полям/функциям первой формы
\+ Передача данных возможна в обе стороны
\+ Не нарушает ООП

6. Используем отдельный класс

Создаем отдельный класс, лучше статический, в основном namespace, т.е. например в файле Program.cs

Код C#

```
1  static class Data
2  {
3      public static string Value { get; set; }
4  }
```

Его открытые свойства/методы доступны из любой формы.

Код C#

```
1  Data.Value = "111";
```

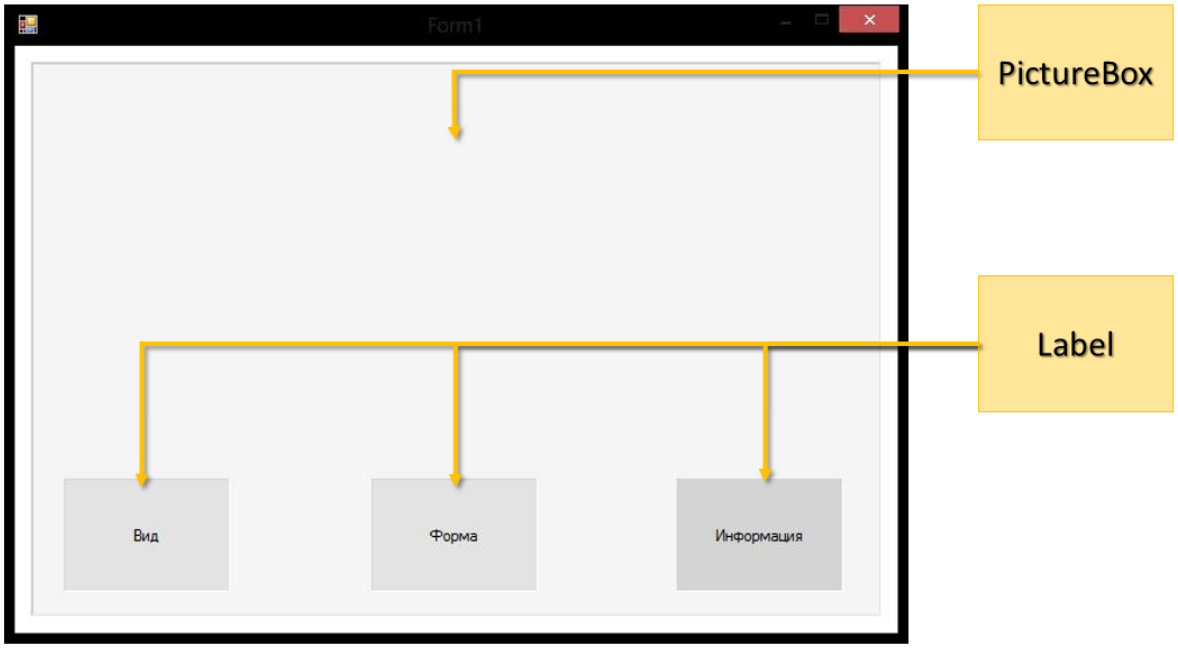
+ Самый удобный способ, когда данные активно используются несколькими формами.

Приведённый выше обзор методов, взят с интернет-источника, с полным списком методов можно ознакомиться перейдя по ссылке: http://www.cyberforum.ru/windows-forms/thread110436.html#post629892

Это задание для самостоятельного решения. Рекомендуем использовать способ, который не нарушает законы ООП и не вносит ряд костылей в код. (В ходе данной работы был использован вариант под номером 5).

Лабораторная работа №4.

Тема: *Перетаскивание графических объектов с помощью мыши.*



Требования к работе:

1) На форме, в границах PictureBox, должны быть созданы три графических объекта: квадрат, эллипс и прямоугольник.

2) Реализована возможность перетаскивания этих объектов по форме.

3) Должно быть выполнено следующее условие: если один из объектов, будет перенесён на Label «Вид», то в третьем Lable «Информация», должны появиться данные о том какого цвета и что это за фигура.

4) При перетаскивании окружности или квадрата на Label «Форма», объект должен меняться на противоположный (т.е. если перенесли квадрат, то он становится окружностью, а окружность, в свою очередь, становится квадратом и наоборот).

Задание 1. Создание визуальной части приложения.

1) Расположите на форме PictureBox.
2) Добавьте три компонента Label

Задание 2. Программирование элементов.

1) Рисование фигур.
2) Реализация Drag’n’Drop.

Порядок действий:

1) Для того, чтобы в элементе PictureBox нарисовать фигуру необходимо выполнить следующие действия:

а) Создать три структуры Rectangle в шапке класса вашей формы:

```
public partial class ЛабораторнаяРабота4 : Form

{

Rectangle Rectangle = new Rectangle(10, 10, 200, 100);
Rectangle Circle = new Rectangle(220, 10, 150, 150);
Rectangle Square = new Rectangle(380, 10, 150, 150);
```

Примечание: первые два параметра - X и Y, вторые – ширина и высота соответственно.

б) Создать обработчик события Paint для PictureBox, в котором, используя методы FillRectangle и FillEllipse класса Graphiscs для параметра обработчика рисования *PaintEventArgs*, «залейте» три необходимых фигуры:

```
e.Graphics.FillEllipse(Brushes.Red, Circle);
e.Graphics.FillRectangle(Brushes.Blue, Square);
e.Graphics.FillRectangle(Brushes.Yellow, Rectangle);
```

2) Для реализации Drag’n’Drop’ нужно:

а) Там же, где объявлялись структуры, создайте три переменные типа bool с начальным значением – false. Эти переменные нужны для проверки «кликнули» ли мы на какой-либо наш объект.

б) Так же создайте еще по две переменных типа int со значением равным 0 для каждого объекта (т.е. 6 переменных). Данные переменные будут фиксировать как изменяются координаты объектов во время перетаскивания.

в) Теперь, в обработчике события MouseDown вашего PictureBox, вам нужно фиксировать изменение координат для ваших объектов, а так же устанавливать флаг, что тот или иной объект выбран.

Пример для прямоугольника, где Rectangle – сам объект, e – параметр для событий мыши (*в данном примере, с помощью него мы получаем координаты указателя мыши*), RectangleX/RectangleY – те самые int-овские переменные, в которые мы и заносим текущие полученные координаты:

```
if ((e.X < Rectangle.X + Rectangle.Width) && (e.X > Rectangle.X))
        {
                if ((e.Y < Rectangle.Y + Rectangle.Height) && (e.Y > Rectangle.Y))
                {
                        RectangleClicked = true;

                        RectangleX = e.X - Rectangle.X;
                        RectangleY = e.Y - Rectangle.Y;

                }
        }
```

Для эллипса и квадрата всё делается аналогично.

г) Создайте обработчик события MouseUp, для PictureBox, в котором просто всем трём переменным типа bool задайте false. Т.е. когда кнопка мыши отпускается, мы устанавливаем всем трём объектам флаги, что ни один из них не «кликнут».

д) Создайте обработчик события MouseMove, всё так же, для PictureBox. Здесь вам нужно проверить какой из объектов в данный момент перетаскивается и нужному присвоить координаты, которые мы считывали в событии MouseDown.

Пример для окружности:

```
if (CircleClicked)
{
        Circle.X = e.X - CircleX;
        Circle.Y = e.Y - CircleY;
}
```

После того, как вы сделаете это для остальных двух объектов, добавьте строку:

```
pictureBox1.Invalidate();
```

В ней вызывается перерисовка PictureBox’a.

3) Выполнение остальной части задания.

Чтобы при перетаскивании объекта в определённую область, у нас происходили какие-то события, нам нужно просто считывать координаты объекта и сравнивать их с координатами нужной нам области (в нашем случае это Label).

а) В событии MouseMove, нам необходимо сверить попадает ли какой-либо из наших объектов в границы Label «Вид». Алгоритм подобен тому, что мы делали в обработчике MouseDown ранее.

Пример для квадрата:

```
if ((label1.Location.X < Square.X + Square.Width) && (label1.Location.X > Square.X))
            {
                if ((label1.Location.Y < Square.Y + Square.Height) && (label1.Location.Y
> Square.Y))
                {
                    label3.Text = "Синий квадрат";
                }
            }
```

б) Создайте ещё пять переменных типа int:

```
        int X, Y, dX, dY;
        int LastClicked = 0;
```

А теперь в обработчике события MouseUp, вам необходимо, как и в прошлом пункте определять положение вашего объекта и сверять его с Label «Форма». Но теперь, предварительно нам нужно знать какая именно из фигур была перенесена. Для этого и нужна переменная LastClicked, в которой содержится информация об этом. (значение этой переменной вы должны задать в MouseUp-событии. Например: 1 – прямоугольник, 2 – круг и 3 – окружность). Переменные X, Y, dX и dY нужны для сохранения координат одной из фигур, дабы передать их другой фигуре.

Пример для изменения формы круга на форму квадрата:

```
if (LastClicked == 2)
            {
                if ((label2.Location.X < Circle.X + Circle.Width) && (label2.Location.X >
Circle.X))
                {
                    if ((label2.Location.Y < Circle.Y + Circle.Height) &&
(label2.Location.Y > Circle.Y))
                    {
                        X = Circle.X;
                        Y = Circle.Y;
                        dX = CircleX;
                        dY = CircleY;
```

```
                    Circle.X = Square.X;
                    Circle.Y = Square.Y;
                    CircleX = SquareX;
                    CircleY = SquareY;

                    Square.X = X;
                    Square.Y = Y;
                    SquareX = dX;
                    SquareY = dY;
                }
            }
        }
```

Для обратного изменения (квадрат становится кругом) всё делается аналогично.

Дополнительное задание: попробуйте сделать так, чтобы при «клике» на определённую фигуру, она становилась на передний план.

Подсказка: в событии Paint, фигуры рисуются в прямом порядке, в котором вы их установили. Т.е. если сначала вы «залили» прямоугольник, а потом круг, то круг будет на переднем плане. Значит, для выполнения, этого задания, вам нужно проверять, как фигура выбрана, а далее просто менять порядок прорисовки.

Пример готовой программы: