Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Ильшат Ринатович Мухаметзянов
Должность: директор
Дата подписания: 13.07.2023 12:35:18
Уникальный программный ключ:
aba80b84033c9ef196388e9ea0434f90a83a40954ba270e84bcbe64f02d1d8d0

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение
высшего образования «Казанский национальный исследовательский технический
университет им. А.Н. Туполева-КАИ»
(КНИТУ-КАИ)
Чистопольский филиал «Восток»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ

по дисциплине

МЕТОДЫ ПРОГРАММИРОВАНИЯ

Индекс по учебному плану: **Б1.В.ДВ.01.02**
Направление подготовки: **09.03.01 Информатика и вычислительная техника**
Квалификация: **Бакалавр**
Профиль подготовки: **Вычислительные машины, комплексы, системы и сети**
Типы задач профессиональной деятельности: **проектный, производственно-технологический**

Рекомендовано УМК ЧФ КНИТУ-КАИ

Чистополь

2023 г.

Практическая работа № 1

Делегаты

В C# существует возможность введения в базовом классе методов, а их реализацию оставить «на потом». Такие методы называют абстрактными. Абстрактный метод автоматически является и виртуальным, но писать это нельзя. Класс, в котором имеется хотя бы один абстрактный метод, тоже называется абстрактным и такой класс может служить только в качестве базового класса. Создать объекты абстрактных классов невозможно, потому что там нет реализации абстрактных методов. Чтобы класс- наследник абстрактного класса не был, в свою очередь, абстрактным (хотя и это не запрещено), там должны содержаться переопределения всех наследованных абстрактных методов.

Модифицируем приведенный выше пример.

```
namespace Virtual1
{
        abstract class Shape
        {  /* Создается абстрактный класс, наличие abstract обязательно!
*/
        public int a,h;
                public Shape (int x,int y)
                {
                a=x;
                h=y;
                }
                public abstract  void Show_area();
                                    //  реализация не нужна – метод абстрактный,
                                  // фиксируется лишь интерфейс метода
        }
        class Class1
        {
```

```
static void Main(string[] args)
{
    Shape q; //указатель на абстрактный класс
// q=new Shape(4,6);   это ошибка, нельзя создать объект
//                                          абстрактного класса!
    q = new Tri(10,20);
    q.Show_area();
    q = new Square(10,20);
    q.Show_area();
        Console.ReadLine();
}        }        }
```

Указатель на абстрактный класс может указывать на любой класс-наследник.

Делегат — это объект, который может ссылаться на метод. Во время выполнения программы один и тот же делегат можно использовать для вызова различных методов, просто заменив метод, на который ссылается этот делегат. Таким образом, метод, который будет вызван делегатом, определяется не в период компиляции программы, а во время ее работы. Делегат в C# соответствует указателю на функцию в C++.

Общий вид объявления делегата:

delegate тип_возвращаемого_значения имя_делегата (список_формальных_параметров);

Для использования делегата должен быть объявлен указатель на делегата

Имя_делегата   указатель_на_делегата;

и этот указатель должен быть инициирован:

указатель_на_делегата = new имя_делегата (имя_функции);

Здесь используется только имя функции (параметры не указываются).

Рассмотрим использование делегата на нескольких примерах.

Пример 1

Дополним программу следующей строкой:

```
namespace Virtual1
{
        delegate void del1();   // объявление  делегата del1
//  далее следует программа из раздела 3.12
}
```

Объявленному делегату del1()могут соответствовать только функции без формальных параметров и без возвращаемого значения.

Главная функция будет иметь вид:

```
static void Main(string[] args)
                {
         Shape q, r;
         q = new Tri(10,20);
         del1 f1 = new del1(q.Show_area);
// объявим переменную типа делегат f1 и поставим ей в соответствие
// функцию Show_area из объекта q класса  Tri
//
         r = new Square(10,20);
         del1 f2 = new del1(r.Show_area);
// объявим переменную типа делегат f2 и поставим ей в соответствие
// функцию Show_area из объекта  r класса  Square
         f1();  // идентичен вызову q.Show_area();
         f2();  // идентичен вызову r.Show_area();
        Console.ReadLine();
                }
```

Примечание. В данной функции переменная r лишняя, вместо нее можно было использовать q.

В этом примере переменная типа «делегат» лишь заменяет имя функции, при этом фиксируется объект, к которому относится представляемая функция.

Пример 2

Делегаты позволяют использовать имя функции в качестве формальных параметров.

```
namespace Deleg
{
        delegate double fun1(double x); //объявление делегата
        class Test
        {
        protected double []x;
        protected double y=0;
                protected int n;
                public Test()
                {
                        n=5;
                        x=new double[n];
                        for(int i=0;i<x.Length;i++)
                        {
                                Console.Write("X["+i+"]=");
                                x[i]=Convert.ToDouble(Console.ReadLine());
                        }
                }
                        public  void gg(fun1 ff)
                                        // формальный параметр – функция
                        {
                                for(int i=0;i<x.Length;i++)

                                y+=(double)ff(x[i]);
```

```
// использование формального параметра – функции
                    Console.WriteLine("Summa={0:##.###}",y);
                    }
              public static double w1(double p)
              {return Math.Sin(p);}
              public  double w2(double p)
              {return Math.Log(p);}
       }
       class Class1
       {
              static void Main(string[] args)
              {
                    Test tt=new Test();
                            //   объявление переменной типа класс Test
                    fun1 f1=new fun1(Test.w1);
// объявление переменной f1 типа делегат fun1, функция  w1
// статическая,  поэтому на нее можно ссылаться через имя класса Test
                    tt.gg(f1);
// использование функции в качестве фактического параметра

                    fun1 f2;
          f2=new fun1(tt.w2);
// объявление переменной f1 типа делегат fun1,  на функцию  w2
// можно ссылаться только через имя объекта tt
                    tt.gg(f1);
// использование функции в качестве фактического параметра

                    Console.ReadLine();
              }             }          }
```

Делегату fun1 могут соответствовать только функции, имеющие тип возвращаемого значения double и один формальный параметр типа double.

Пример 3. Многоадресный делегат. Одному делегату можно ставить в соответствие несколько функций. В таком случае они будут выполнены в такой последовательности, как они были прикреплены к делегату.

```
namespace Deleg_2
{
delegate int Deleg(ref string st);        //  объявим делегат
    class Class1
    {
        public static int met1(ref string x)
        {
        Console.WriteLine("I am Metod 1");
        x+=" 11111";
   return 5;
        }
        public static int met2(ref string y)
        {
        Console.WriteLine("I am Metod 2");
            y+=" AAAAA";
     return 55;
        }
        static void Main(string[] args)
        {
            Deleg d1;
     int k;
            string r="******";
            Deleg d2=new Deleg(met1);
              // связываем делегат и функцию
```

```
Deleg d3=new Deleg(met2);
d1=d2;   // присоединим первый делегат
d1+=d3;  // добавим второй делегат
k=d1(ref r);
Console.WriteLine(r);
Console.WriteLine("k=" + k);
Console.ReadLine();
}            }         }
```

В качестве ответа получим:

I am Metod 1

I am Metod 2

****** 11111 AAAAA

K=55

Как видно из примера, в случае, когда делегат имеет возвращаемое значение (в нашем случае int), значением многоадресного делегата будет значение, возвращенное последней функцией (в нашем случае met2).

Для исключения функции из многоадресного делегата необходимо писать d1-=d3;   (вычитать удаляемый делегат, в данном случае d3).

Таким образом, делегаты расширяют средства программирования в двух случаях:

- делегаты как указатели на функцию. Это позволяет использовать функции в качестве формальных/фактических параметров других функций;

- многоадресные делегаты. Что дает образом, получим возможность одним вызовом обеспечить выполнение ряда функций.

Использование делегата в роли псевдонима функции может иногда уменьшить объем наших записей , но не расширяет наши возможности.

На основе делегатов построено еще одно важное средство C#: событие (event). Событие — это автоматическое уведомление о выполнении

некоторого действия. События работают следующим образом. Объект, которому необходима информация о некотором событии, регистрирует обработчик для этого события. Когда ожидаемое событие происходит, вызываются все зарегистрированные обработчики. Обработчики событий представляются делегатами. События — это члены класса, которые объявляются с использованием ключевого слова event. Наиболее распространенная форма объявления события имеет следующий вид:

event событийный_делегат объект;

Здесь элемент *событийный_делегат* означает имя делегата, используемого для поддержки объявляемого события, а элемент объект — это имя создаваемого событийного объекта.

Пример 4

```
namespace Event1
{
    delegate void MyEvent(); //объявим делегат события
    class My
    {   //  класс события
    public event MyEvent activate; //объявим событие activate
        public void fire()
        {
        if(activate!=null)activate();
        }
    }
    class Demo
    {
        static void handler()
        { // функция – обработчик события
        Console.WriteLine("Что-то случилось . . . ");
        }
        static void Main(string[] args)
```

```
{
        My evt=new My();
        evt.activate+=new MyEvent(handler);
        // метод handler регистрируется в качестве обработчика события
        evt.fire();
        Console.ReadLine();
}       }       }
```

Все события активизируются посредством делегата. Следовательно, событийный делегат определяет сигнатуру для события. В данном случае параметры отсутствуют, однако событийные параметры разрешены. Затем создается класс события My. При выполнении следующей строки кода, принадлежащей этому классу, объявляется событийный объект MyEvent. Кроме того, внутри класса My объявляется метод fire(), который в этой программе вызывается, чтобы сигнализировать о событии (другими словами, этот метод вызывается, когда происходит событие). Как показано в следующем фрагменте кода, он вызывает обработчика события посредством делегата if(activate!=null)activate();

Обратите внимание на то, что обработчик события вызывается только в том случае, если делегат activate не равен null-значению. Поскольку другие части программы, чтобы получить уведомление о событии, должны зарегистрироваться, можно сделать так, чтобы метод fire () был вызван до регистрации любого обработчика события. Чтобы предотвратить вызов null-объекта, событийный делегат необходимо протестировать и убедиться в том, что он не равен null-значению. Внутри класса Demo создается обработчик события handler (). В этом примере обработчик события просто отображает сообщение, но ясно, что другие обработчики могли бы выполнять более полезные действия. В методе Main() создается объект класса My, а метод handler() регистрируется в качестве обработчика этого события. Обратите внимание на то, что обработчик добавляется в список с использованием

составного оператора "+=". Следует отметить, что события поддерживают только операторы "+=" и " –= ".

Подобно делегатам события могут предназначаться для многоадресной передачи. В этом случае на одно уведомление о событии может отвечать несколько объектов.

Пример 5

```
namespace Events2
{
    delegate void MyEvent();
//    определение делегата, на основе которого будет определено событие
    class My
    {
        public event MyEvent activate;  //определение события
        public void fire()
        {
            if (activate!=null) activate();
        }
    }
    class X
    { // первый обработчик
        public void Xhandler()
    {
    Console.WriteLine("I am X");
    }
    }
    class Y
    {
        public void Yhandler()
        {    // второй обработчик
```

```
            Console.WriteLine("I am Y");
            }
        }
        class Class1
        {
            static void handler()
            {   // третий обработчик, функция статическая
            Console.WriteLine("I am base");
            }
            static void Main(string[] args)
            {
                My evt =new My();
                X  x1= new X();
                Y  y1=new Y();
                evt.activate+=new MyEvent(handler);
    // так можно писать, если функция-обработчик статическая
                evt.activate+=new MyEvent(x1.Xhandler);
                evt.activate+=new MyEvent(y1.Yhandler);
    //если функция обычная, то ссылка на нее только через объект соответствующего класса
                evt.fire();
                Console.ReadLine();
            }       }       }
```

Практическая работа №2

В рамках практической работы требуется создать графическое приложение, 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.

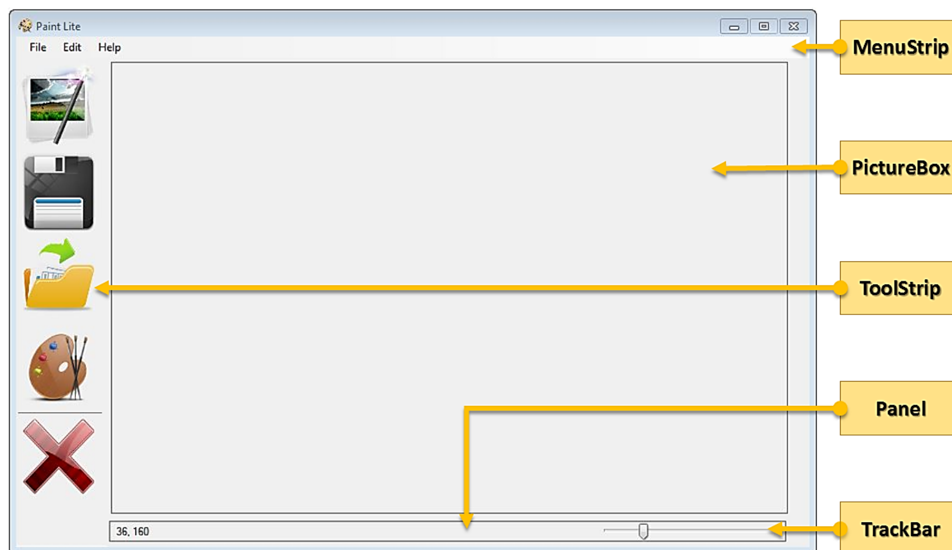
Процедуры обработки этих событий получают следующие параметры:

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.

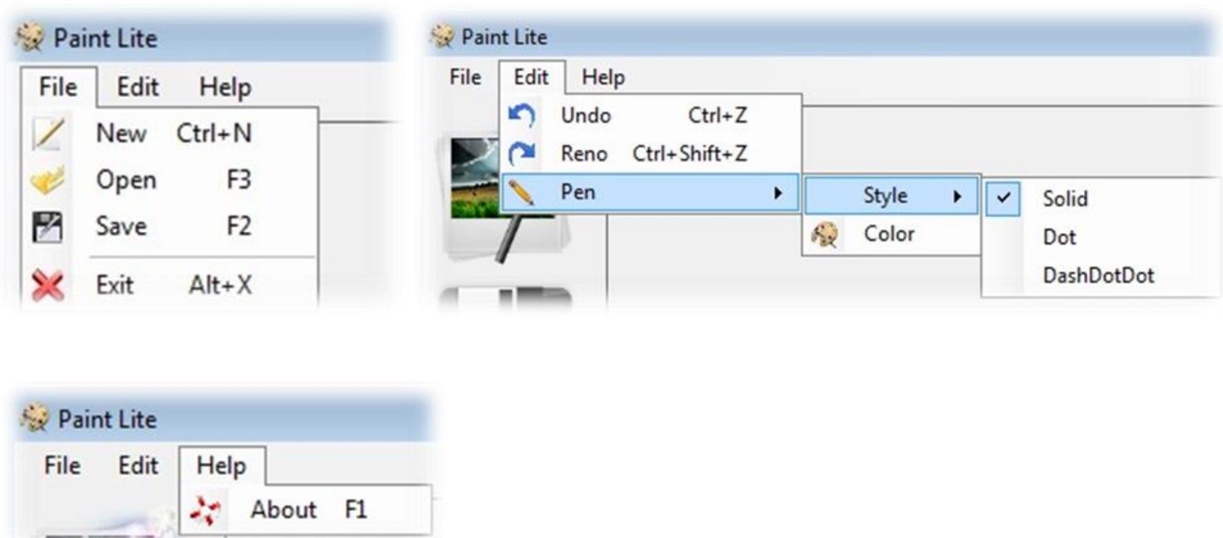
X, Y. Координаты указателя мыши в пикселях относительно клиентной области окна с координатами (0,0) в верхнем левом углу.

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.

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.

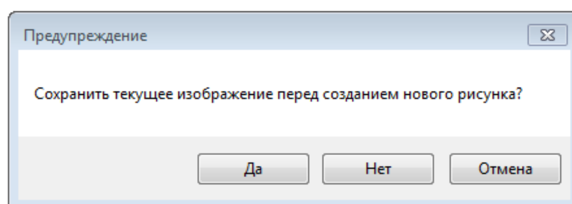
```
if (SaveDlg.FileName != "")     //Если введено не пустое имя
            {
                System.IO.FileStream fs =
                  (System.IO.FileStream)SaveDlg.OpenFile();

                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();
            }
```

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:

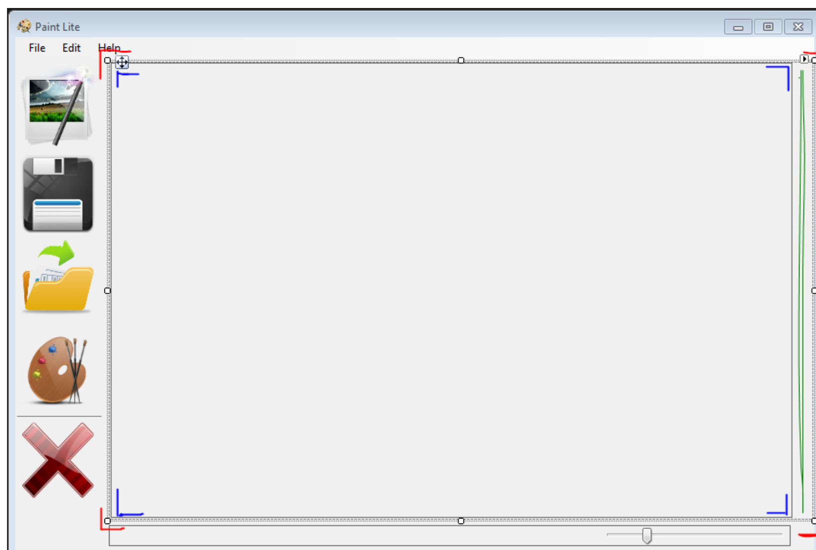
```
OpenFileDialog OP = new OpenFileDialog();
   OP.Filter = "JPEG Image|*.jpg|Bitmap Image|*.bmp|GIF Image|*.gif|PNG Image|*.png";
```

```
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;
```

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.

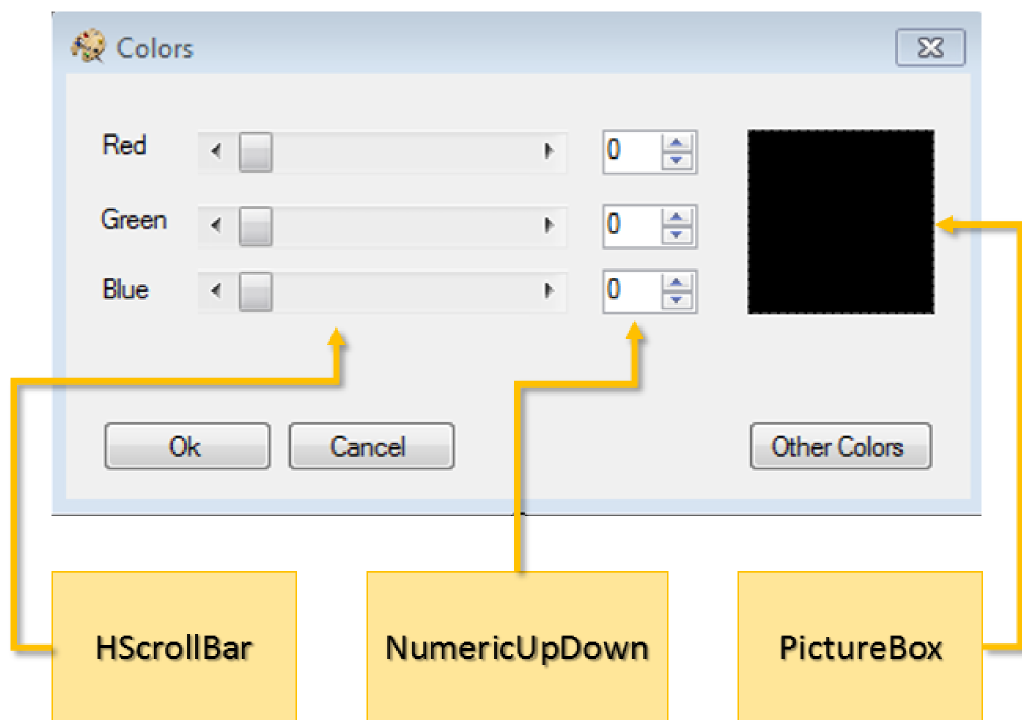
Небольшое замечание.

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.

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:

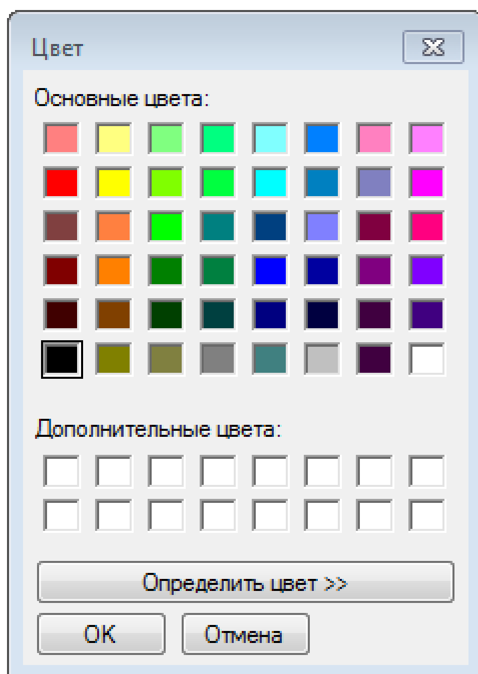
```
private void UpdateColor()
    {
        colorResult = Color.FromArgb(Scroll_Red.Value, Scroll_Green.Value, Scroll_Blue.Value);
        picResultColor.BackColor = colorResult;
    }
```

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#

Form2 f = new Form2();
f.ShowDialog();
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#

Код C#

```
public string Data
{
    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;
```

Код C#

```
}
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#

```
public Form2(Form1 f1)
{
    InitializeComponent();

            //Обрабатываем данные
    //Или записываем их в поле
            string s = f1.textBox1.Text;
}
```

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

Код C#

Код C#

```
Form2 f = new Form2(this);
f.ShowDialog();
//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#

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

Во второй форме определяем владельца

Код C#

```
Form1 main = this.Owner as Form1;
if(main != null)
{
    string s = main.textBox1.Text;
    main.textBox1.Text = "OK";
}
```

+ 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#

```
static class Data
{
    public static string Value {
get; set; }
}
```

Его открытые свойства/методы доступны из любой формы.

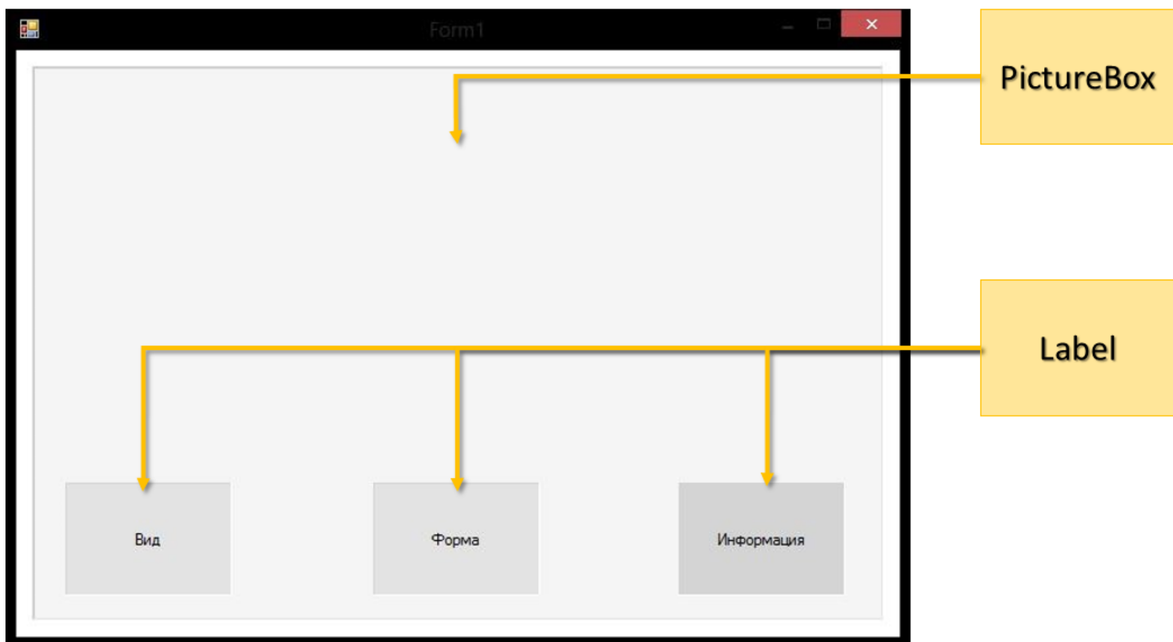
Код C#

```
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.



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.

Расположите на форме PictureBox.

Добавьте три компонента Label

Задание 2. Программирование элементов.

Рисование фигур.

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:

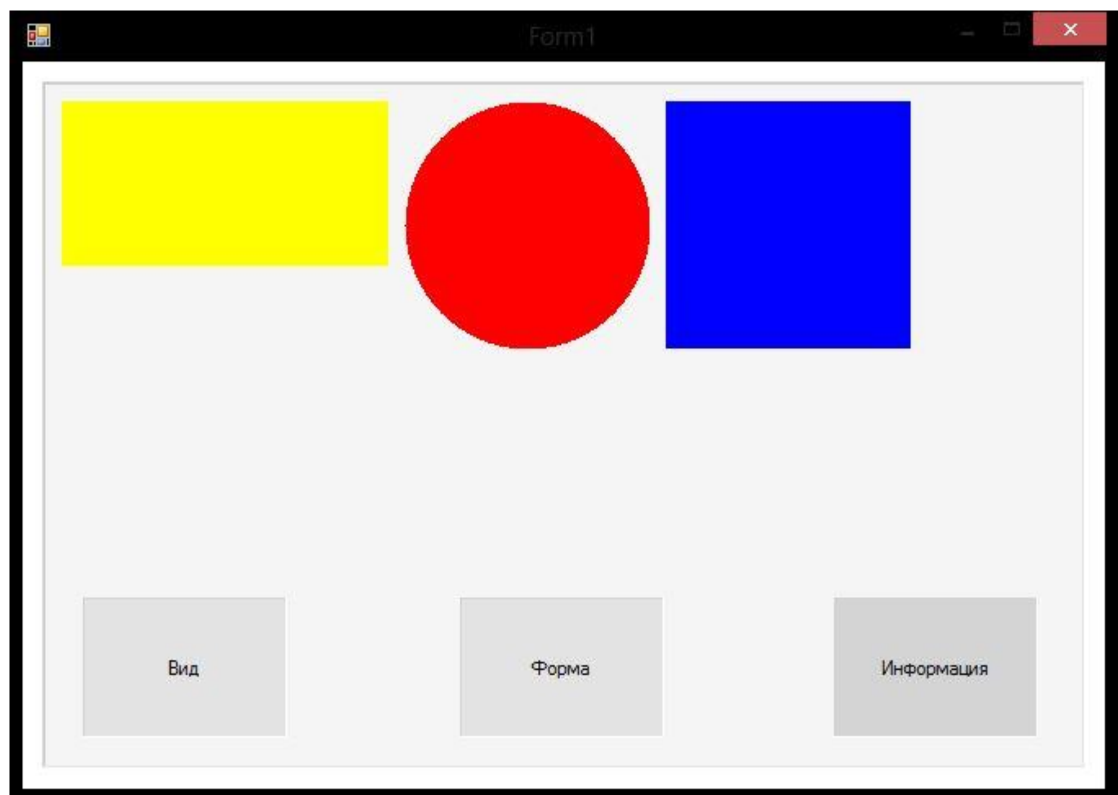
```
if (LastClicked == 2)
      {
         if ((label2.Location.X < Circle.X + Circle.Width) && (label2.Location.X > Circle.X))
         {
            if ((label2.Location.Y < Circle.Y + Circle.Height) && (label2.Location.Y > Circle.Y))
            {
               X = Circle.X;
               Y = Circle.Y;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

Пример готовой программы:



Практическая работа №3

Графика и перерисовка

В пространстве имен System.Drawing − определены основные структуры для представления:

- точки (координат) − Point и PointF
- размера − Size и SizeF
- прямоугольных областей − Rectangle и RectangleF.

Буква F в конце названия структуры означает, что, в отличие от обычных структур (без F), поля структуры имеют не целочисленные значения, а значения вещественного типа (float).

Структура Size

Структура Size предназначена для хранения ширины и высоты объекта и имеет, для этого, соответствующие открытые свойства Width и Height, доступные как для записи, так и для чтения. При создании объекта Size, с помощью конструктора по умолчанию:

```
Size sz = new Size();
```

свойства Width и Height устанавливаются в ноль.

Изменить значения свойств в последствии можно, например, следующим образом:

```
sz.Width = 40;
sz.Height = 60;
```

Структура содержит два конструктора с аргументами:

```
public Size(int, int);
public Size(Point);
```

Конструкторы с аргументами позволяют, в момент создания, инициализировать разные экземпляры структуры по-разному:

```
Size sz1= new Size(10,20);      // sz1.Width = 10, sz1.Height = 20
Size sz2 = new Size(15,50);     // sz2.Width = 15, sz2.Height = 50
```

Структура Point

Структура Point содержит открытые свойства X и Y целого типа, доступные, как для записи, так и для чтения.

Для создания точки “pt” можно использовать конструктор по умолчанию:

```
Point pt   = new Point();
```

Конструктор по умолчанию при создании точки обнуляет значения свойств X и Y.

В дальнейшем координаты точки можно изменить, например, следующим образом:

```
pt.X =25;
pt.Y=75;
```

Инициализировать новую точку класса Point, можно используя, конструкторы с аргументами:

```
public Point(Size);
public Point(int, int);
```

Например:

```
Point pt1 = new Point(10,20); // pt1.X =10, pt1.Y=20
Size sz = new  Size(27,45);
Point pt2 = new Point(sz);    // pt2.X=27, pt2.Y=45
```

Открытый метод структуры:

public void **Offset(** int *dx*, int *dy*)**;**

изменяет текущие координаты точки по формулам:

X=X+dx и Y=Y+dy;

Структура Rectangle

Структура предназначена для определения координат и размера прямоугольника. Структура содержит открытые свойства, часть из которых доступна только для чтения, а часть, как для чтения, так и для записи.

В структуре определены два конструктора с аргументами:

```
public Rectangle(
  int x,           // x-координата левого верхнего угла прямоугольника
  int y,           // y-координата левого верхнего угла прямоугольника
  int width,       // ширина прямоугольника
  int height       // высота   прямоугольника
);
public Rectangle(
  Point location,  // координата левого верхнего угла прямоугольника
  Size size        // размер прямоугольника
);
```

С помощью этих конструкторов можно определять размеры и местоположение прямоугольников при их создании:

```
Point pt =  new Point(10,15);
Size   sz = new Size (50,70);
Rectangle rct = new Rectangle(pt,sz);
Rectangle rect = new Rectangle(20,20,50,30);
```

Структура Rectangle содержит ряд доступных методов. Рассмотрим некоторые из них.

Метод:

```
public void Intersect(Rectangle);
```

Возвращает структуру, которая описывает прямоугольник, представляющий пересечение двух прямоугольников. Если не имеется никакого пересечения, все свойства структуры обнуляются.

Например:

```
Rectangle rect,rct;
rect = new Rectangle(20,25,50,55);
rct = new Rectangle(10,10,30,40);
rect.Intersect(rct);
```

выполнение, приведенного фрагмента кода установит значения свойства структуры прямоугольника rect следующим образом:

X=20, Y=25, Width=20, Height=25.

Метод:

public static Rectangle Union(Rectangle a, Rectangle b);

Возвращает структуру, описывающий минимальный по размерам прямоугольник, включающий в себя прямоугольники a и b.

Методы public void Offset(Point pos) и public void Offset(int x, int y) смещают координаты левой верхней точки прямоугольника по обеим осям на величину, задаваемую параметрами методов.

Представление цвета

Представление цвета осуществляется с помощью экземпляров структуры System.Drawing.Color.

Для задания цвета используется статический метод класса:

public static Color.FromArgb(int red, int green, int blue);

Параметры метода red, green и blue задают интенсивность красной, зеленой и синей составляющей цвета. Значение каждой компоненты цвета может изменяться в диапазоне от 0 до 255. Это позволяет отобразить 2^{24} различных цветов.

Для задания цвета можно также использовать один из перегруженных методов FromArgb:

```
public static Color FromArgb(int alpha, Color cr);
public static Color FromArgb(int alpha,  int red, int green, int blue);
```

Дополнительный параметр alpha задает степень прозрачности цвета. Чем меньше это число, тем меньше насыщенность цвета и тем более прозрачным является этот цвет. Значение параметра alpha может изменяться в диапазоне от 0 до 255. Значение 0 определяет полностью прозрачный (бесцветный), а значение 255 − полностью насыщенный (непрозрачный) цвет.

Структура Color содержит более 200 статических свойств, доступных только для чтения. Эти свойства задают именованные или, так называемые, Интернет − цвета, которые правильно воспроизводятся всеми WEB браузерами.

Например:

```
Color clr2 = Color.Beige;      // бежевый
Color clr3 = Color.Magenta;    // сиреневый
Color clr4 = Color.Orange;     // оранжевый
```

Кисти и перья

Графические объекты рисуются с помощью перьев и кистей.

Сплошные кисти создаются как экземпляры класса System.Drawing.SolidBrush, например:

```
Brush br2 = new SolidBrush(Color.Magenta);
Brush br3 = new SolidBrush(Color.FromArgb(200,10,120));
```

Параметр color конструктора public SolidBrush(Color color)
класса SolidBrush задает цвет сплошной кисти.

В классе System.Drawing.Brushes определено большое количество статических свойств, возвращающих кисть Интернет цветов. Создание таких кистей выглядит следующим образом:

```
Brush brr = Brushes.Orange;
```

В классе System.Drawing.Drawing2D.HatchBrush определены штриховые кисти.

Конструкторы класса:

```
public HatchBrush(HatchStyle hatchstyle, Color foreColor, Color backColor);
public HatchBrush(HatchStyle hatchstyle, Color foreColor);
```

Параметры:

foreColor − цвет штриха кисти;

backColor − цвет фонового штриха кисти (по умолчанию − черный цвет);

hatchstyle − стиль штриховой кисти.

Существует большое количество доступных стилей, наиболее типичными являются:

Cross − решетчатая кисть;

DiagonalCross − диагональная решетчатая кисть;

Horizontal − горизонтальная штриховка;

Vertical − вертикальная штриховка.

Например, создание кисти с вертикальной штриховкой синего цвета и фоновым штрихом бежевого цвета будет выглядеть следующим образом:

```
Brush br1 = new HatchBrush(HatchStyle.Vertical,Color.Blue,Color.Beige);
```

Перья описываются классом System.Drawing.Pen.

Конструкторы класса:

```
public Pen(Color color);
```

```
public Pen(Color color, float width);
public Pen( Brush brush);
public Pen(Brush brush, float width);
```

Параметры:

color − цвет пера;

width − толщина пера;

brush −кисть.

Примеры создания перьев:

```
Pen pn   = new Pen(Color. Magenta);
Pen pn1 = new Pen(Color.Orange,5);
Pen pn2 = new Pen(Brushes.Orange);
Pen pn3 = new Pen(Brushes.Magenta,10);
Pen pn4 = new Pen(Color.FromArgb(125,155, 0));
Pen pn5 = new Pen(Color.FromArgb(25,155,200),10);
```

В классе System.Drawing.Pens содержится множество статических свойств, описывающих перья с интернет цветом и толщиной в один пиксель. Создание таких перьев выглядит следующим образом:

```
Pen pn6 = Pens.Brown;
Pen pn7 = Pens.Magenta;
```

Рисование линий и фигур

Для вывода текстовой и графической информации в окно приложения необходимо использовать контекст устройства.

Контекст устройства в среде .NET инкапсулирован («завернут») в базовом классе System.Drawing.Graphics. Для создания объекта класса Graphics необходимо использовать метод CreateGraphics(), возвращающий ссылку на объект класса Graphics:

```
Graphics  dc = CreateGraphics();
```

Полученный объект dc содержит контекст устройства, позволяющий осуществлять вывод информации в окно приложения.

Класс Graphics содержит множество методов, позволяющих рисовать различные графические фигуры. Рассмотрим некоторые из них.

Рисование контуров прямоугольников осуществляется с помощью методов:

```
public void DrawRectangle( Pen pen,  Rectangle rect);
public void DrawRectangle( Pen pen, int x, int y, int width, int height);
public void DrawRectangle( Pen pen, float x, float y, float width, float height);
```

Рисование контуров эллипсов осуществляется с помощью методов:

```
public void DrawEllipse ( Pen pen,  Rectangle rect);
public void DrawEllipse ( Pen pen, int x, int y, int width, int height);
public void DrawEllipse ( Pen pen, float x, float y, float width, float height);
```

Рисование закрашенных эллипсов и прямоугольников осуществляется с помощью методов:

```
public void FillEllipse( Brush brush, Rectangle rect);
public void FillEllipse( Brush brush, int x, int y, int width, int height);
public void FillEllipse( Brush brush, float x, float y, float width, float height);
public void FillRectangle( Brush brush, Rectangle rect);
public void FillRectangle( Brush brush, int x, int y, int width, int height);
public void FillRectangle( Brush brush, float x, float y, float width, float height);
```

Параметры методов означают следующее:

pen − перо;

brush − кисть;

rect − прямоугольник;

x − координата x левого верхнего угла прямоугольника;

y − координата y левого верхнего угла прямоугольника;

width − ширина прямоугольника;

height −высота прямоугольника;

Рисование линий осуществляется с помощью перегруженных методов:

public void DrawLine(Pen pen, Point pt1, Point pt2);

public void DrawLine(Pen pen, PointF pt1, PointF pt2);

public void DrawLine(Pen pen, int x1, int y1, int x2, int y2);

public void DrawLine(Pen pen, float x1, float y1, float x2, float y2);

Параметры методов означают следующее:

pen − перо;

pt1 − начальная точка рисования;

pt2 − конечная точка рисования;

x1 и y1 − координаты начальной точки рисования;

x2 и y2 − координаты конечной точки рисования;

Примеры использоания функций:

```
dc.DrawRectangle(Pens.OrangeRed,5,10,25,45);
dc.DrawEllipse(Pens.Magenta,100,125,20,15);
dc.FillEllipse(Brushes.BlueViolet,45,50,20,15);
dc.DrawLine(Pens.Green,20,40,60,70);
```

Рисование текста

Для рисования текста используют перегруженный метод DrawString. Рассмотрим два из шести перегруженных методов DrawString:

public void DrawString(string s, Font fnt, Brush br, PointF pt);

public void DrawString(string s, Font fnt, Brush br, RectangleF ltRct);

Параметры:

s –строка символов,

fnt – шрифт текста,

br – кисть,

pt – точка, определяющая координаты вывода текста,

ltRct – прямоугольник, внутри которого выводится текст, если же текст не вмещается в область прямоугольника, то он (текст) обрезается.

Например:

```
Font fnt = new Font("Arial",10); //Шрифт Arial, размер 10
dc.DrawString("Привет!",fnt, Brushes.Green,10,20);
```

Перерисовка окна приложения

Если свернуть окно приложения, затем вновь развернуть его, то мы, к сожалению, заметим, что изображение на поверхности окна исчезло. Операционная система не восстанавливает содержимого окна. Восстановлением графики и текста должно заниматься само приложение. Операционная система в необходимых случаях вырабатывает сообщение (событие Paint), которое «говорит», что окно приложения не корректно и его необходимо перерисовать. Перерисовка окна должна происходить по событию Paint. Метод-обработчик этого события имеет заголовок:

```
private void Form_Paint(object sender, System.Windows.Forms.PaintEventArgs e)
```

Для этого метода нет необходимости создавать контекст устройства, он передается методу с помощью параметра e. Для получения контекста устройства необходимо выполнить следующую операцию:

```
Graphics dc = e.Graphics;
```

В теле этой функции необходимо выполнить все действия для перерисовки окна.

Очень часто перерисовка окна должна происходить в определенные моменты времени по инициативе приложения. Это бывает необходимо при выводе на экран анимации.

«Заставить» операционную систему выработать событие Paint можно путем вызова метода Invalidate(), который является членом класса

System.Windows.Forms.Form. Существуют несколько перегруженных версий этого метода. Одна из них принимает в качестве параметра прямоугольник, который определяет область окна для перерисовки. Используемая нами версия без параметров перерисовывает все окно.

Практическая работа №4

Дочерние окно и элементы интерфейса

Вариант 1

Написать Windows-приложение, которое выполняет анимацию изображения.

Создать меню с командами Show picture, Choose, Animate, Stop, Quit.

Команда Quit завершает работу приложения. При выборе команды Show picture в центре экрана рисуется объект, состоящий из нескольких графических примитивов.

При выборе команды Choose открывается диалоговое окно, содержащее:

- поле типа TextBox с меткой Speed для ввода скорости движения объекта;
- группу Direction из двух переключателей (Up-Down, Left-Right) типа RadioButton для выбора направления движения;
- кнопку типа Button.

По команде Animate объект начинает перемещаться в выбранном направлении до края окна и обратно с заданной скоростью, по команде Stop — прекращает движение.

Вариант 2

Написать Windows-приложение, которое по заданным в файле исходным данным строит график или столбиковую диаграмму.

Создать меню с командами Input data, Choose, Line, Bar, Quit.

Команды Line и Bar недоступны. Команда Quit завершает работу приложения.

При выборе команды Input data из файла читаются исходные данные (файл сформировать самостоятельно).

По команде Choose открывается диалоговое окно, содержащее:

- список для выбора цвета графика типа ListBox;
- группу из двух переключателей (Line, Bar) типа RadioButton;

- кнопку типа Button.

Обеспечить возможность ввода цвета и выбора режима: построение графика (Line) или столбиковой диаграммы (Bar). После указания параметров становится доступной соответствующая команда меню.

По команде Line или Bar в главном окне приложения выбранным цветом строится график или диаграмма. Окно должно содержать заголовок графика или диаграммы, наименование и градацию осей. Изображение должно занимать все окно и масштабироваться при изменении размеров окна.

Вариант 3

Написать Windows-приложение, которое строит графики четырех заданных функций.

Создать меню с командами Chart, Build, Clear, About, Quit.

Команда Quit завершает работу приложения. При выборе команды About открывается окно с информацией о разработчике.

Команда Chart открывает диалоговое окно, содержащее:

- список для выбора цвета графика типа ListBox;
- список для выбора типа графика типа ListBox, содержащий четыре пункта: sin(x), sin(x+π/4), cos(x), cos(x–π/4);
- кнопку типа Button.

Обеспечить возможность выбора цвета и вида графика. После щелчка на кнопке OK в главном окне приложения строится график выбранной функции на интервале от -π/2 до +π/2. Окно должно содержать заголовок графика, наименование и градацию осей. Изображение должно занимать все окно и масштабироваться при изменении размеров окна.

Команда Clear очищает окно.

Вариант 4

Написать Windows-приложение — графическую иллюстрацию сортировки одномерного массива методом «выбора».

Создать меню с командами File, Animate, About, Exit.

Команда Animate недоступна. Команда Exit завершает работу приложение. Команда About открывает окно с информацией о разработчике. Для выбора файла исходных данных (команда File) использовать объект класса OpenFileDialog.

Из выбранного файла читаются исходные данные для сортировки (сформировать самостоятельно не менее трех файлов различной длины с данными целого типа).

После чтения данных становится доступной команда Animate.

При выборе команды Animate в главном окне приложения отображается процесс сортировки в виде столбиковой диаграммы. Каждый элемент представляется столбиком соответствующего размера. На каждом шаге алгоритма два элемента меняются местами. Окно должно содержать заголовок. Изображение должно занимать все окно.

Вариант 5

Написать Windows-приложение — графическую иллюстрацию сортировки одномерного массива методом пузырька.

Создать меню с командами File, Animate, About, Exit.

Команда Animate недоступна. Команда Exit завершает работу приложение. Команда About открывает окно с информацией о разработчике. Для выбора файла исходных данных (команда File) использовать объект класса OpenFileDialog.

Из выбранного файла читаются исходные данные для сортировки (сформировать самостоятельно не менее трех файлов различной длины с данными целого типа).

После чтения данных становится доступной команда Animate.

При выборе команды Animate в главном окне приложения отображается процесс сортировки в виде столбиковой диаграммы. Каждый элемент представляется столбиком соответствующего размера. На каждом шаге алгоритма два элемента меняются местами. Окно должно содержать заголовок. Изображение должно занимать все окно.

Процесс – объект, который создается ОС для приложения (не для приложения .NET) в момент его запуска. Характеризуется собственным адресным пространством, которое напрямую недоступно другим процессам.

В рамках процесса создаются потоки (один – первичный – создается всегда). Это последовательность выполняемых команд процессора. В приложении может быть несколько потоков (первичный поток и дополнительные потоки).

Потоки в процессе разделяют совместно используемые данные и имеют собственные стеки вызовов и локальную память потока (Thread Local Storage – TLS). TLS потока содержит информацию о ресурсах, используемых потоком, о регистрах, состоянии памяти, выполняемой инструкции процессора.

Практическая работа №5

Обзор пространства имен System.Threading

В этом пространстве объявляются типы, которые используются для создания многопоточных приложений: работа с потоком, средства синхронизации доступа к общим данным, примитивный вариант класса Timer.

Тип	Назначение
Interlocked	Синхронизация доступа к общим данным
Monitor	Синхронизация потоковых объектов при помощи блокировок и управления ожиданием
WaitCallback	Делегат, представляющий методы для рабочих элементов (объектов) класса ThreadPool
Mutex	Синхронизация процессов
Thread	Собственно класс потока, работающего в среде выполнения .NET. В текущем домене приложения с

	помощью этого класса создаются новые потоки
ThreadPool	Класс, предоставляющий средства управления набором взаимосвязанных потоков
ThreadStart	Класс-делегат для метода, который должен быть выполнен перед запуском потока
Timer	Вариант класса-делегата, который обеспечивает передачу управления некоторой функции-члену (неважно какого класса!) в указанное время. Сама процедура ожидания выполняется потоком в пуле потоков
TimerCallback	Класс-делегат для объектов класса Timer
WaitHandle	Объекты представители этого класса являются объектами синхронизации (обеспечивают многократное ожидание)

Класс Thread. Общая характеристика

Thread-класс представляет управляемые потоки: создает потоки и управляет ими — устанавливает приоритет и статус потоков. Это объектная оболочка вокруг определенного этапа выполнения программы внутри домена приложения.

Статические члены класса Thread	Назначение
CurrentThread	Свойство. Только для чтения. Возвращает ссылку на поток,

	выполняемый в настоящее время
GetData() SetData()	Обслуживание слота текущего потока
GetDomain() GetDomainID()	Получение ссылки на домен приложения (на ID домена), в рамках которого работает указанный поток
Sleep()	Блокировка выполнения потока на определенное время
Нестатические члены	Назначение
IsAlive	Свойство. Если поток запущен, то true
IsBackground	Свойство. Работа в фоновом режиме. GC работает как фоновый поток
Name	Свойство. Дружественное текстовое имя потока. Если поток никак не назван – значение свойства установлено в null. Поток может быть поименован единожды. Попытка переименования потока возбуждает исключение
Priority	Свойство. Значение приоритета потока. Область значений – значения перечисления ThreadPriority
ThreadState	Свойство. Состояние потока. Область значений – значения перечисления ThreadState
Interrupt()	Прерывание работы текущего потока
Join()	Ожидание появления другого потока (или определенного промежутка времени) с последующим завершением
Resume()	Возобновление выполнения потока после приостановки
Start()	Начало выполнения ранее созданного потока,

	представленного делегатом класса ThreadStart
Suspend()	Приостановка выполнения потока
Abort()	Завершение выполнения потока посредством генерации исключения
TreadAbortException	В останавливаемом потоке. Это исключение следует перехватывать для продолжения выполнения оставшихся потоков приложения. Перегруженный вариант метода содержит параметр типа object, который может включать дополнительную специфичную для данного приложения информацию

Именование потока

Потоки рождаются безымянными. Это означает, что у объекта, представляющего поток, свойство Name имеет значение null. Ничего страшного. Главный поток все равно изначально поименовать некому. То же самое и с остальными потоками. Однако никто не может помешать потоку поинтересоваться своим именем — и получить имя. Несмотря на то, что это свойство при выполнении приложения играет вспомогательную роль, повторное переименование потоков недопустимо. Повторное изменение значения свойства Name приводит к возбуждению исключения.

```
using System;
using System.Threading;
namespace ThreadApp_1
 {
 class StartClass
 {
       static void Main(string[] args)
```

```
{
int i = 0;
 bool isNamed = false;
do
 {
try
 {
 if (Thread.CurrentThread.Name == null)
  {
  Console.Write("Get the name for current Thread > ");
  Thread.CurrentThread.Name = Console.ReadLine();
  }
 else
  {
  Console.WriteLine("Current Thread : {0}.",      Thread.Current
 Thread. Name);
 if (!isNamed)
  {
  Console.Write("Rename it. Please...");
  Thread.CurrentThread.Name = Console.ReadLine();
  }
 }
 }
 catch (InvalidOperationException e)
 {
 Console.WriteLine("{0}:{1}",e,e.Message);
 isNamed = true;
 }
i++;
```

```
}
while (i < 10);
}

}
}
```

Характеристики точки входа дополнительного потока

Сигнатура точки входа в поток определяется характеристиками класса-делегата:

```
public delegate void ThreadStart();
```

Класс-делегат здесь с пустым списком параметров. Очевидно, что точка входа обязана соответствовать этой спецификации. У функции, представляющей точку входа, должен быть тот же самый список параметров, то есть пустой.

Пустой список параметров функции, представляющей точку входа потока, – это не самое страшное ограничение. Если учесть то обстоятельство, что создаваемый поток не является первичным потоком, то это означает, что вся необходимая входная информация может быть получена заранее и представлена в классе в доступном для функций – членов данного класса виде, то для функции, представляющей точку входа, не составит особого труда эту информацию получить! А зато можно обойтись минимальным набором функций, обслуживающих данный поток.

Запуск вторичных потоков

Вторичные потоки запускаются последовательно из главного потока. А уже последовательность выполнения этих потоков определяется планировщиком.

Вторичный поток также вправе поинтересоваться о собственном имени. Надо всего лишь расположить этот код в правильном месте. И чтобы он выполнился в правильное время:

```
using System;
using System.Threading;
```

```
namespace ThreadApp_1
{
class Worker
{
	int allTimes;
	int n;
	// Конструктор умолчания...
	public Worker()
	{
	n = 0;
	allTimes = 0;
	}
	// Конструктор с параметрами...
	public Worker(int nKey, int tKey)
	{
	n = nKey;
	allTimes = tKey;
	}
	public void DoItEasy()
	{

	int i;
	for (i = 0; i < allTimes; i++)
	{
	if (n == 0)
	Console.Write("{0,25}\r",i);
	else
	Console.Write("{0,10}\r",i);
	}
	Console.WriteLine("\nWorker was here!");
	}
```

```
}

class StartClass
{
static void Main(string[] args)
{
Worker w0 = new Worker(0,100000);
Worker w1 = new Worker(1,100000);
ThreadStart t0, t1;
t0 = new ThreadStart(w0.DoItEasy);
t1 = new ThreadStart(w1.DoItEasy);
Thread th0, th1;
th0 = new Thread(t0);
th1 = new Thread(t1);
// При создании потока не обязательно использовать делегат.
// th1 = new Thread(w1.DoItEasy);
th0.Start();
th1.Start();
}
}
}
```

Первичный поток ничем не лучше любых других потоков приложения. Он может скоропостижно завершиться раньше всех им же порожденных потоков. Приложение же завершается после выполнения последней команды в последнем выполняемом потоке. Неважно, в каком.

Приостановка выполнения потока

Обеспечивается статическим методом Sleep(). Метод статический – это значит, что всегда производится не "усыпление", а "самоусыпление" выполняемого в данный момент потока. Выполнение методов текущего

потока блокируется на определенные интервалы времени. Все зависит от выбора перегруженного варианта метода. Планировщик потоков смотрит на поток и принимает решение относительно того, можно ли продолжить выполнение усыпленного потока.

В самом простом случае целочисленный параметр определяет временной интервал блокировки потока в миллисекундах.

Если значение параметра установлено в 0, поток будет остановлен до того момента, пока не будет предоставлен очередной интервал для выполнения операторов потока.

Если значение интервала задано с помощью объекта класса TimeSpan, то момент, когда может быть возобновлено выполнение потока, определяется с учетом закодированной в этом объекте информации:

```
// Поток заснул на 1 час, 2 минуты, 3 секунды:
Thread.Sleep(new TimeSpan(1,2,3));
:::::
// Поток заснул на 1 день, 2 часа, 3 минуты, 4 секунды, 5 миллисекунд:
Thread.Sleep(new TimeSpan(1,2,3,4,5));
```

Значение параметра, представленное выражением

System.Threading.Timeout.Infinite

позволяет усыпить поток на неопределенное время. А разбудить поток при этом можно с помощью метода Interrupt(), который в этом случае вызывается из другого потока:

Всегда надо помнить: приложение выполняется до тех пор, пока не будет выполнен последний оператор последнего потока. И неважно, выполняются ли при этом потоки, "спят" либо просто заблокированы.

Отстранение потока от выполнения

Обеспечивается нестатическим методом Suspend(). Поток входит в "кому", из которой его можно вывести, вызвав метод Resume(). Этот вызов, естественно, должен исходить из другого потока. Если все не отстраненные от выполнения потоки оказались завершены и некому запустить

отстраненный поток – приложение в буквальном смысле "зависает". Операторы в потоке могут выполняться, а выполнить их невозможно по причине отстранения потока от выполнения. Вот приложение и зависает...

```
using System;
 using System.Threading;
public class ThreadWork
 {
 public static void DoWork()
 {
 for(int i=0; i<10; i++)
 {
 Console.WriteLine("Thread – working.");
Thread.Sleep(25);
 }

Console.WriteLine("Thread - still alive and working.");
Console.WriteLine("Thread - finished working.");
 }
 }
class ThreadAbortTest
 {
 public static void Main()
 {
 ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
 Thread myThread = new Thread(myThreadDelegate);
 myThread.Start();
 Thread.Sleep(10);
 Console.WriteLine("Main - aborting my thread.");
 myThread.Suspend();
 Console.WriteLine("Main ending.");
```

```
}
}
```

Ко всему прочему следует иметь в виду, что метод Suspend() уже устарел.

Не следует использовать методы Suspend и Resume для синхронизации активности потоков. Просто в принципе ничего нельзя будет сделать, когда код в выполняемом потоке будет приостановлен с помощью метода Suspend(). Одним из нежелательных последствий подобного устранения от выполнения может оказаться взаимная блокировка потоков.

Завершение потоков

Первый вариант остановки потока тривиален. Поток завершается после выполнения последнего оператора выполняемой цепочки операторов. Допустим, в ходе выполнения условного оператора значение некоторой переменной сравнивается с фиксированным значением и в случае совпадения значений управление передается оператору return:

```
for (x=0;;x++)
 {
 if (x==max)
 return; // Все. Этот оператор оказался последним.
 else
 {
  ::::::::::
  }
 }
```

Поток может быть остановлен в результате выполнения метода Abort(). Эта остановка является достаточно сложным делом.

При выполнении этого метода происходит активация исключения ThreadAbortException. Естественно, это исключение может быть перехвачено в соответствующем блоке catch. Во время обработки исключения допустимо

выполнение самых разных действий, которые осуществляются в этом самом "остановленном" потоке. В том числе возможна и реанимация остановленного потока путем вызова метода ResetAbort().

При перехвате исключения CLR обеспечивает выполнение операторов блоков finally, которые выполняются все в том же потоке.

Таким образом, остановка потока путем вызова метода Abort не может рассматриваться как немедленная остановка выполнения потока:

```
   using System;
 using System.Threading;

public class ThreadWork
 {
 public static void DoWork()
 {
 int i;

try
 {
  for(i=0; i<100; i++)
  {
     Console.WriteLine("Thread – working {0}.", i);
    Thread.Sleep(10);
    }
 }
 catch(ThreadAbortException e)
 {
  //6.
  //– Ну дела! А где это мы...
  Console.WriteLine("Thread – caught ThreadAbortException – resetting.");
```

```
 Console.WriteLine("Exception message: {0}", e.Message);
 Thread.ResetAbort();
}
 finally
 {
 //7.
 //– Вот где бы мы остались, если бы не удалось отменить
 // остановку потока! finally блок... Отстой!
 Console.WriteLine("Thread – in finally statement.");
 }
Console.WriteLine("Thread – still alive and working.");
Console.WriteLine("Thread – finished working.");
 }
 }

class ThreadAbortTest
 {
 public static void Main()
 {
ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
 Thread myThread = new Thread(myThreadDelegate);
myThread.Start();
Thread.Sleep(50);
Console.WriteLine("Main – aborting my thread.");
 myThread.Abort();
Console.WriteLine("Main ending.");
 }
 }
```

Практическая работа №6

Развертывание приложения

Microsoft Visual Studio.Net поддерживает самые разные способы развертывания приложения: от самых простых, до сложных, с использованием Windows Installer.

Когда приложение полностью готово, необходимо его полностью подготовить к развертыванию, т.е. к установке на компьютере клиента с минимальными усилиями.

Развертывание приложения с помощью простого копирования.

Полное развертывание имеет ряд ограничений:

1. Все файлы, необходимые для работы приложения должны находиться в этом же каталоге где и приложение (библиотеки, ресурсы, файлы и т.д.).
2. На целевом компьютере до начала развертывания должна быть установлена среда .NET Framework.
3. Развертываемое приложение не должно требовать файлов или ресурсов. Кроме тех, которые уже установлены на компьютере.

Обычно такое развертывание осуществляется с помощью команды

X COPY D:\MyApp C:\MyApp/S

Откуда куда

Пуск/Все программы/Стандартные/Командная строка

Создание проекта установочной программы.

Для Windows Forms поддерживается два вида установочных проектов:

1. для приложений (setup project)
2. для дополнительных модулей Merge Module projects.

Первый применяется для получения дистрибутива, пригодного для развертывания исполняемого приложения, а второй для элементов управления и компонентов, которые не являются отдельными приложениями и не подлежат непосредственному развертыванию.

Для создания дистрибутива готовой программы необходимо добавить к решению проект установочной программы.

Для этого в пункте File проекта необходимо выбрать команду Add Project/New project.

Откроется диалоговое окно Add New Project, содержащее две панели:

- левая – панель Project Types
- правая – панель Templates

На левой панели выбираем: Setup and Deployment Projects.

А на правой: Setup Wizard.

Затем щелкаем на кнопке ОК.

В результате запускается мастер Setup Wizard и появляется окно с приветствием Setup Wizard(1 of 5). В этом окошке необходимо щелкнуть кнопку Next.

Появляется второе окно Setup Wizard(2 of 5) и страницей Choose a project type, на которой необходимо задать тип проекта с помощью одного из четырех флажков: два флажка для исполняемой сборки

○ Create a Setup for a Windows Application

○ Create a Setup for a Web Application

И два для установки неисполняемой сборки (dll)

○ Create a merge module for Windows Installer

○ Create a dowloadable CAB file.

Обычно (для нас) первый флаг.

Затем Next.

Появится третье окно со страницей: Choose Project Outputs To Include, которая позволяет выбрать файлы решения для включения в проект установочной программы.

На этой странице отображается шесть **флажков**:

□ Documentation Files from…

□ Primary output from…

□ Localized resources from…

□ Debug Symbols from…

□ Content Files from…

□ Source Files from…

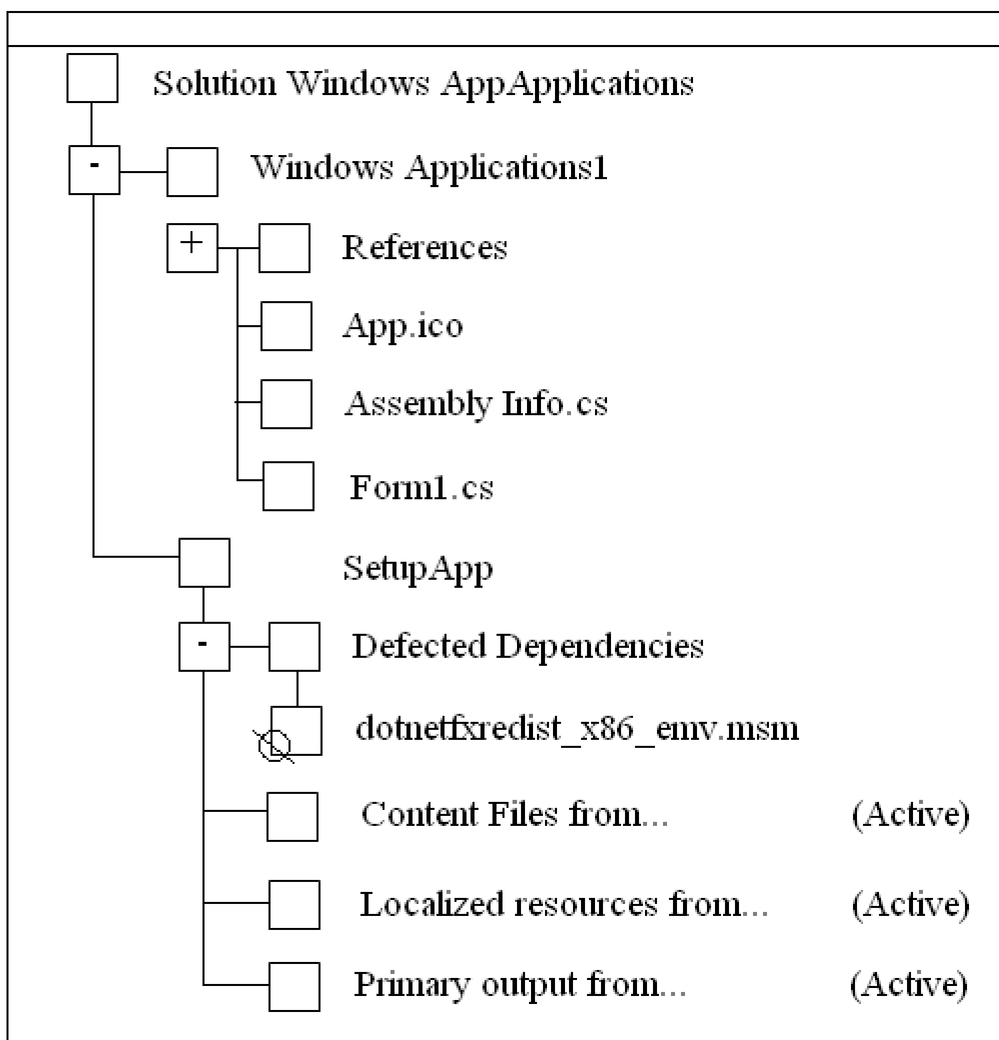
Для того чтобы включить в проект EXE и DLL – файлы необходимо установить флажок Primary output from… . При этом внизу на панели Description появится краткое описание. Обычно еще установлен флажок Content Files from… и флажок Localized resources from…

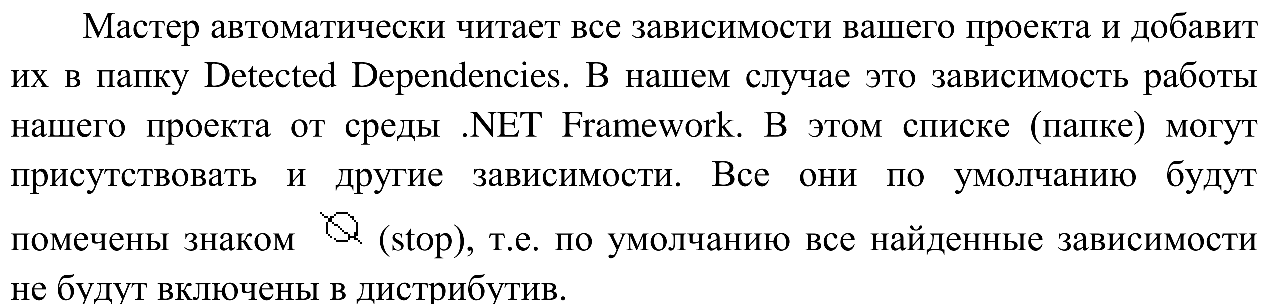
Можно еще на этапе отладки включить информацию для отладки и исходные тексты, но в конечном варианте их не должно быть.

Затем нажимается кнопка Next, и мастер переходит к 4 шагу из 5 и открывается страница Choose To Include, позволяющая добавить к проекту любые дополнительные файлы, например текстовые файлы, справочные файлы в формате HTML и т.п.

Добавление осуществляется с помощью кнопки Add(если не хотите что-либо добавлять → Next). Появляется страница Create Project последнего шага работы мастера, отображающая информацию о параметрах вашего проекта. Щелкнув кнопку Finish, мы создадим проект установочной программы и добавим его к решению.

Новый проект отобразится в окне Solution Explorer.





Мастер автоматически читает все зависимости вашего проекта и добавит их в папку Detected Dependencies. В нашем случае это зависимость работы нашего проекта от среды .NET Framework. В этом списке (папке) могут присутствовать и другие зависимости. Все они по умолчанию будут помечены знаком (stop), т.е. по умолчанию все найденные зависимости не будут включены в дистрибутив.

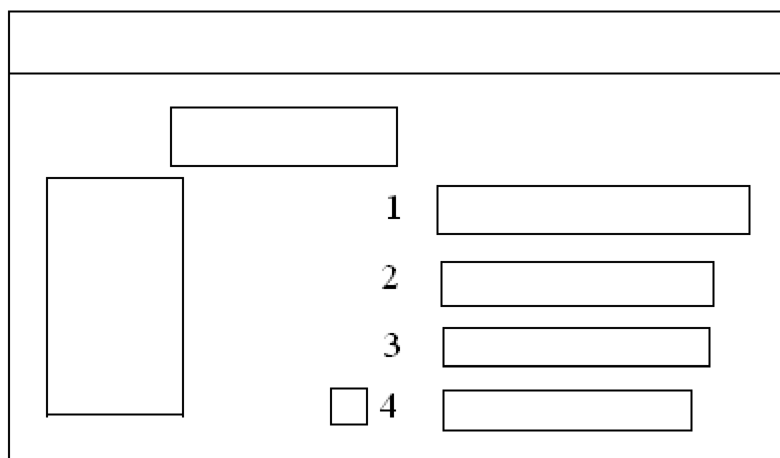
Для включения найденной зависимости в дистрибутив необходимо щелкнуть правой кнопкой на названии файла и сбросить флажок Exclude для этого файла. Включать dotnet… следует только в том случае, если на целевом компьютере нет среды .NET.

Параметры компоновки установочной программы.

Обычно, в результате компиляции установочного проекта, получается, по крайней мере один файл с расширением .msi, в котором хранятся все конфигурационные данные и содержимое, необходимое для установки приложения. Но можно, в зависимости от носителя, на котором

предполагается переносить установочные файлы, размещать пакет на нескольких отдельных файлах, а также включать в него установочные файлы Windows Installer, сами они отсутствуют на целевом компьютере.

Для изменения параметров компоновки установочной программы используется диалоговое окно свойств проекта установочной программы. Для этого необходимо в Solution Explorer, щелкнув на имени установочного проекта SetupApp правой кнопкой мыши и выбрать из контекстного меню команду Properties – откроется диалоговое окно свойств проекта.



1. **Output File Name** определяет путь и имя, которое будет назначено выходному файлу(установочному) по умолчанию: name_cataloge\project_name.msi

   Или msm для дополнительных модулей.

   Можно изменить каталог с помощью кнопки Browse…

2. **Package Files** определяет способ установки выходных файлов решения. По умолчанию все выходные файлы упаковываются в один дистрибутивный файл вместе с установочной программой. Это обеспечивает высокую степень сжатия при минимальной сложности установки.
   In Setup file

   In Cabinet file(s) – упаковка в несколько CAB файлов, при этом можно укусить размер CAB – файлов в байтах (1,44 для *****). Для этого используется окно CAB size, которое становится доступным, если выбрана опция In Cabinet file.

   Если установлен флажок Unlimited, то будет один непрерывный CAB – файл.

Если установлен флажок Custom file, то

□ Unlimited ↙ здесь задается max размер

□ Custom [ ] CAB файла.

Опция As loose uncompressed files – позволяет скомпилировать в виде не сжатых файлов.

3. **Bootstrapper –** этот параметр определяет необходимо ли включить в дистрибутив пакет Windows Installer. Если выбран этот параметр, то генерируются дополнительные файлы, которые используются для установки на целевом компьютере Windows Installer. Для ОС XP – это не требуется, т.к. он там уже стоит, поэтому для XP можно выбрать None. В противном случае будут сгенерированы следующие файлы:

   Setup.exe проверяет установлен ли на целевом компьютере Windows Installer и если нет, то вызывает в зависимости от версии ОС InstMsiA.exe или InstMsiW.exe для установки Windows Installer. После этого запускается Windows Installer для установки приложения, извлекая его из Msi файла.

   InstMsiA.exe – для Windows 95/98

   InstMsiW.exe – NT/200

   Размер каждого из этих файлов при максимальном сжатии около 1,85 МГб.

   Setup.exe содержит имя Msi файла, который Setup.exe обрабатывает после проверки Windows Installer.

   При выборе параметра Web Bootstrapper открывается диалоговое окно Web Bootstrapper Settings. В нем указывается с помощью свойства: Setup folder URL – Web каталог, в котором находится программа установки и Windows Installer Upgrade folder Url указывается отдельный каталог для загрузки через Web InstMsiA.exe и InstMsiW.exe файлов.

   Т.е. программы для установки Windows Installer и самого приложения можно хранить в разных Web каталогах. По умолчанию – в одном.

Compression определяет степень сжатия. None – нет. Optimized For Speed – компромисс между скоростью и объектом. Optimized For Size – max степень сжатия.

На данном этапе можно считать, что все подготовлено для создания дистрибутива, и можно выбрать в окне Solution Explorer правой кнопкой SetupApp и в открывшемся контекстном поле меню выбрать пункт компиляции Build.

Затем сгенерированные файлы можно скопировать на выбранный для распространения носитель.

Конфигурирование проекта установочной программы.

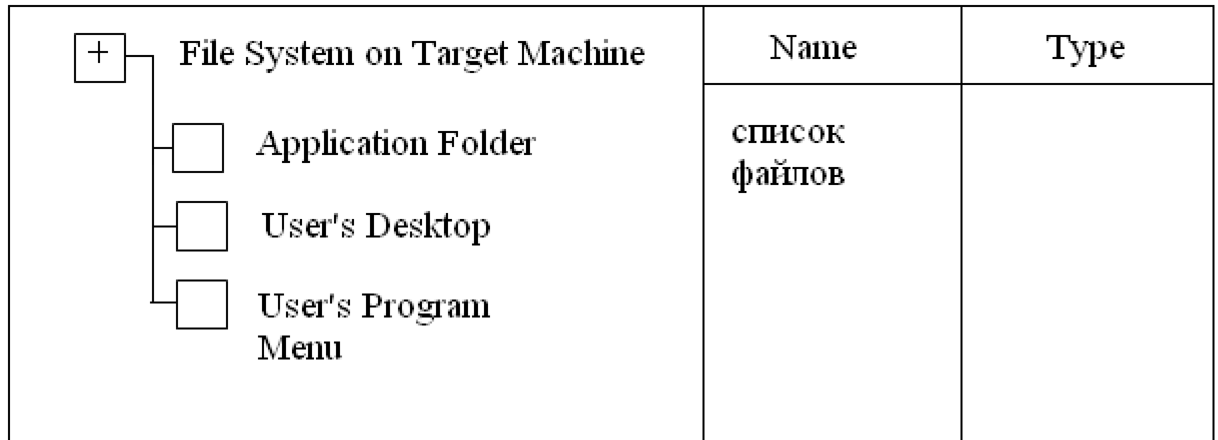
В Visual Studio.Net имеется шесть редакторов свойств установочной программы:

1. Редактор файловой системы (File System Editor) – позволяет выбрать каталог файловой системы целевого компьютера.
2. Редактор реестра (Registry Editor) создает записи в реестре в процессе установки.
3. Редактор типов файлов (File Type Editor) связывает типы файлов с обрабатываемыми приложениями.
4. Редактор пользовательского интерфейса (User Interface Editor) – модифицирует пользовательский интерфейс программы.
5. Редактор нестандартных действий (Custom Actions Editor) – залает нестандартные действия, выполняемые установочной программой.
6. Редактор условий (Launch Conditions Editor) – определяет необходимые условия для начала установки.

1. Редактор файловой системы.

Редактор файловой системы позволяет манипулировать файловой системой целевого компьютера: записывать выходные файлы в различные каталоги, создавать на целевом компьютере не новые каталоги, а также создавать и добавлять ярлыки.

Для запуска редактора файловой системы необходимо выделить SetupApp в окне Solution Explorer и выбрать в меню View команду Editor/File System Editor – окно редактора.



На правой панели отображается список выходных файлов проекта установочной программы, а на левой структура каталогов целевого компьютера.

Первоначально она включает в себя три папки:

- Каталог приложения
- Рабочий стол
- Поле для программ пользователя

Можно добавить свои собственные папки, если щелкнуть левую панель правой кнопкой и выбрать из контекстного меню Add Special Folder, а затем стандартную папку или создать свою. По умолчанию выходные файлы записываются в каталог приложения. Целевой каталог можно изменить, выделив его на правой панели и перетащив его на левую панель в необходимую папку.

Редактор файловой системы позволяет:

1. Добавить не сжатый файл к дистрибутиву.
2. Разместить сборки в GAC во время установки приложения.
3. Добавить в какую-либо папку ярлык:
    - Для этого на левой панели щелкнуть Application Folder, затем выбрать из списка на правой панели файл, для которого необходимо создать ярлык.
    - Щелкнуть правой кнопкой файл, для которого необходимо создать ярлык и выбрать из контекстного меню команду Create Shortevt. Ярлык для файла создается и добавляется на панель.
    - Перетащить ярлык с правой панели в необходимый каталог на левой панели.

Редактор условий установки

Launch Conditions ****** позволяет определять условия, которым целевой компьютер должен удовлетворять для начала установки, например, версия Windows. Кроме того, можно проверять наличие определенных файлов, записей реестра, компонентов, и принимать решения о начале установки по результатам проверки.

Окно редактора поделено на две части: в первой задают объект (файл, раздел реестра или компонент), и во второй – условие, зависящее от наличия этого компонента на целевом компьютере.

Если условия выполняются, то процесс установки продолжается, в противном случае прекращает, и откат.

Основная литература:

1. Тюкачев, Н. А. C#. Основы программирования : учебное пособие для ву-зов / Н. А. Тюкачев, В. Г. Хлебостроев. — 4-е изд., стер. — Санкт-Петербург : Лань, 2021. — 272 с. — ISBN 978-5-8114-7266-6. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/158960 (да-та обращения: 04.07.2023). — Режим доступа: для авториз. пользователей.

2. Технология разработки программного обеспечения : учеб. пособие / Л.Г. Гагарина, Е.В. Кокорева, Б.Д. Виснадул ; под ред. Л.Г. Гагариной. — М. : ИД «ФОРУМ» : ИНФРА-М, 2017. — 400 с. — Режим доступа: http://znanium.com/bookread2.php?book=768473

Дополнительная литература

3. Гуриков С.Р. Введение в программирование на языке Visual C# : учеб. по-собие / С.Р. Гуриков. — М. : ФОРУМ : ИНФРА-М, 2017. — 447 с. — Режим до-ступа: http://znanium.com/bookread2.php?book=752394

4. Объектно-ориентированное программирование с примерами на C#: Учеб-ное пособие / Хорев П.Б. - М.: Форум, НИЦ ИНФРА-М, 2016. - 200 с. — Режим доступа: http://znanium.com/bookread2.php?book=529350

5. Ступина, А. А. Технология надежностного программирования задач авто-матизации управления в технических системах [Электронный ресурс] : монография / А. А. Ступина, С. Н. Ежеманская. - Красноярск : Сиб. федер. ун-т, 2011. - 164 с. — Режим доступа: http://znanium.com/bookread2.php?book=442655

6. Осипов, Н.А. Разработка Windows приложений на C#. [Электронный ре-сурс] — Электрон. дан. — СПб. : НИУ ИТМО, 2012. — 74 с. — Режим доступа: https://e.lanbook.com/reader/book/40725/#50