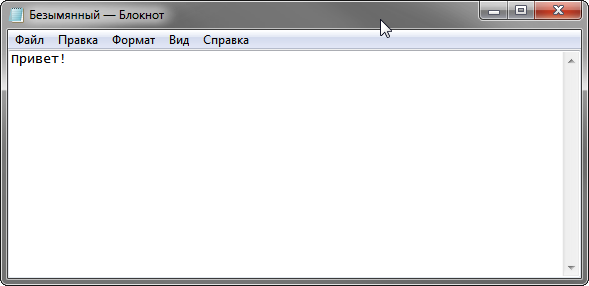
# Объектно-ориентированное программирование

# Лабораторная работа №4

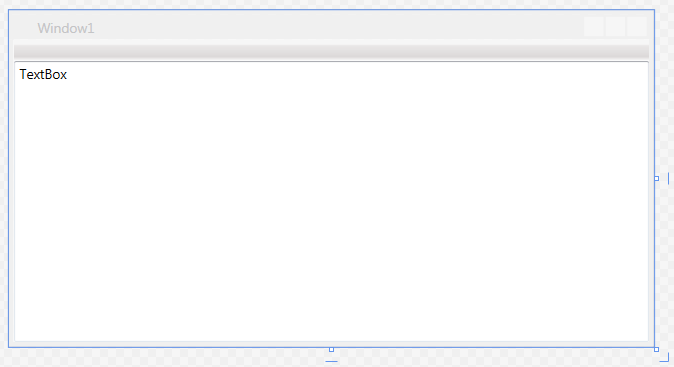
## Вариант 1. Программа «Блокнот»



Программа блокнот представляет собой простейший редактор, позволяющий создавать и изменять текстовые файлы. Попробуем создать аналогичную программу самостоятельно.

### Создание основного окна

В VisualStudio создаем новый проект WPF Application и на основную форму приложения добавляем элементы TextBox и Menu. Располагаем их в соответствии с рисунком:



Для того, чтобы элементы правильно изменяли свои размеры при масштабировании окна приложения нужно правильно установить привязки к краям окна:

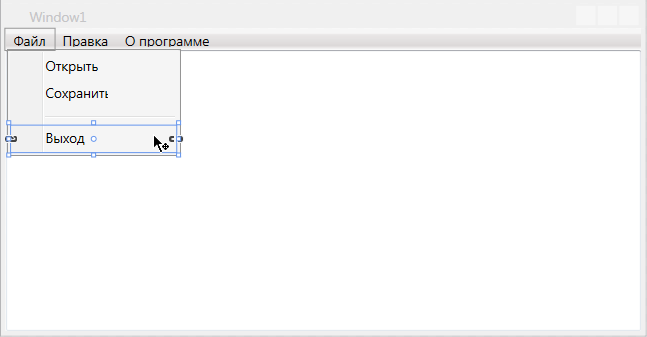


Для TextBox привязки должны быть закрыты со всех четырех сторон, а для Menu – для всех кроме нижнего края.

После этого добавляем к меню новые пункты (правый клик –> Add MenuItem):

* Файл
  + Открыть
  + Сохранить
  + Выход
* Правка
  + Отменить
  + Повторить
* О программе

После этого окно программы должно выглядеть следующим образом:



Затем всем элементам, включая пункты меню, задаем осмысленные названия, чтобы затем использовать их в коде программы, а для текстового окна включаем свойство AcceptsReturn, для возможности переноса строк по клавише Enter.

### Открытие и сохранение файлов

Для того чтобы пользователь мог указать файл для открытия и имя файла для сохранения используются диалоги открытия и сохранения. Windows предоставляет стандартные диалоги, которые мы можем использовать в своих программах. Для этого в начале программы необходимо указать директиву using Microsoft.Win32; разрешающую использование стандартных диалогов.

Там же указываем директиву using System.IO; необходимую для использования класса File, который используется для работы с файлами.

После этого дважды кликаем по пункту меню открыть и в созданном методе пишем следующее:

private void menuOpen\_Click(object sender, RoutedEventArgs e)

{

// создаем объект диалога открытия

var dialog = new OpenFileDialog();

// устанавливаем фильтр файлов

dialog.Filter = "Текстовые файлы | \*.txt";

// показываем диалог

var result = dialog.ShowDialog();

//если диалог закрыли кнопкой "Открыть",

// то result будет равен true,

//если закрыли крестиком или кнопкой "Отмена"

// то result будет равен false

if (result == true)

{

//Метод ReadAllText - открывает текстовый файл

//и возвращает его содержимое

textBox.Text = File.ReadAllText(dialog.FileName);

}

}

И аналогично для команды сохранения файла:

private void menuSave\_Click(object sender, RoutedEventArgs e)

{

// создаем объект диалога сохранения

var dialog = new SaveFileDialog();

// устанавливаем фильтр файлов

dialog.Filter = "Текстовые файлы | \*.txt";

// показываем диалог

var result = dialog.ShowDialog();

//если диалог закрыли кнопкой "Сохранить",

// то result будет равен true,

//если закрыли крестиком или кнопкой "Отмена"

// то result будет равен false

if (result == true)

{

// Метод WriteAllText - записывает указанный текст

// в файл с выбранным в диалоге именем

File.WriteAllText(dialog.FileName, textBox.Text);

}

}

Так же добавляем обработчики пунктов меню Отменить и Повторить, код для них выглядит следующим образом:

private void MenuUndo\_Click(object sender, RoutedEventArgs e)

{

textBox.Undo();

}

private void MenuRedo\_Click(object sender, RoutedEventArgs e)

{

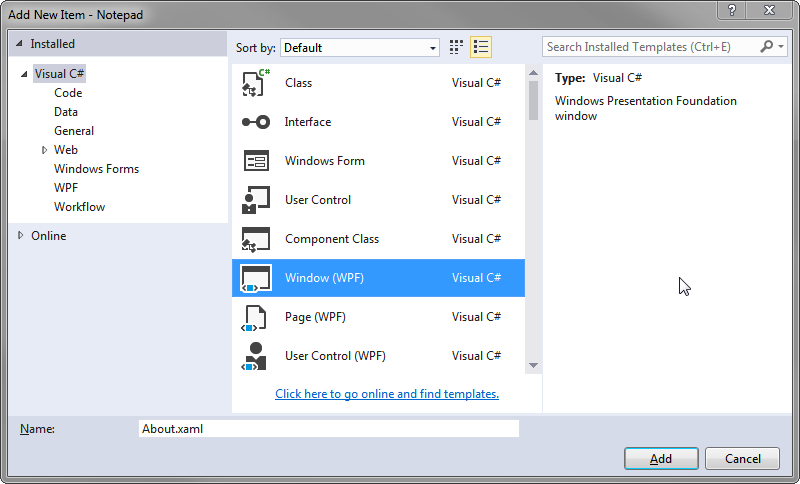
textBox.Redo();

}

В них мы вызываем встроенные в класс TextBox методы Undo и Redo, которые позволяют отменить или повторить изменения текста.

### Добавление окна «О программе»

Для добавления к программе нового окна нужно выбрать Project->Add Window и в открывшемся диалоге выбрать Window (WPF).



После этого откроется редактор форм в котором будет открыто свежесозданное окно. Оформите его по своему вкусу, обязательно укажите имена авторов, группу и год создания.

Затем, для того, чтобы это окно появлялось по кнопке меню, возвращаемся к главной форме и дважды кликаем по пункту меню «О программе». В открывшемся редакторе кода пишем следующее:

private void MenuAbout\_Click(object sender, RoutedEventArgs e)

{

// создаем новое окно

var about = new About();

// и показываем его на экране

about.Show();

}

### Задание

После выполнения шагов выше все базовые функции текстового редактора у вас реализованы. Необходимо добавить еще несколько штрихов:

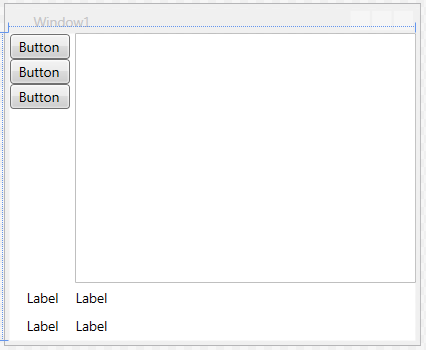
1. Текущий пункт меню «Сохранить» переименовать в «Сохранить как». Добавить новый пункт меню «Сохранить», который будет сохранять файл без диалога, если файл уже выбран пользователем.
2. Добавить пункт меню «Создать», который будет очищать текущий текст.
3. Добавить предупреждение если файл был изменен после последней операции сохранения перед открытием, созданием нового или закрытием программы. Используйте событие TextChanged элемента TextBox для того, чтобы узнать об изменении текста.
4. Добавьте элемент StatusBar в который выводите количество слов в текущем файле.

## Вариант 2. Рисовалка

Создадим простую программу для рисования

### Создание основного окна

В VisualStudio создаем новый проект WPF Application и на основную форму приложения добавляем следующие элементы: 3 Button, один Canvas, 4 Label и один Border. Canvas помещаем внутрь Border, для того, чтобы вокруг Canvas была рамка. И организуем их примерно следующим образом:

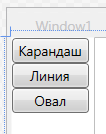


Для того, чтобы элементы правильно изменяли свои размеры при масштабировании окна приложения нужно правильно установить привязки к краям окна:



Label должны быть привязаны к нижнему и левому краю, а Border ко всем четырем краям.

Затем именуем и подписываем кнопки:



Удаляем текст с каждого элемента Label, и для каждого устанавливаем свой цвет фона, примерно так:



Для элемента Canvas устанавливаем любой непрозрачный цвет фона.

После чего переходим к написанию кода.

### Рисование линий

В программе у нас будет 3 режима рисования: рисование прямых линий, кривых и овалов. Для этого определим в теле класса MainWindow соответствующий тип и переменную этого типа:

enum DrawMode

{

PEN,

LINE,

ELLIPSE

}

DrawMode mode;

Переключение режима рисования будет производиться по нажатию одной из кнопок. Для этого создаем обработчики нажатия кнопок:

private void btnPen\_Click(object sender, RoutedEventArgs e)

{

mode = DrawMode.PEN;

}

private void btnLine\_Click(object sender, RoutedEventArgs e)

{

mode = DrawMode.LINE;

}

private void btnEllipse\_Click(object sender, RoutedEventArgs e)

{

mode = DrawMode.ELLIPSE;

}

Для того чтобы что-то нарисовать, необходимо обработать всего два события – нажатие кнопки мыши, то есть начало рисования, и перемещение мыши по полю. Переходим к форме, выбираем элемент Canvas и в списке его событий находим MouseDown и MouseMove. Создаем обработчики этих событий.

Для того чтобы нарисовать прямую линию, нам нужно запомнить координаты мыши в момент начала рисования, создать новый элемент Line и при каждом перемещении мыши обновлять координаты одного из концов этого элемента.

В теле класса MainWindow (рядом с enum DrawMode) добавим переменную line которая будет ссылаться на рисуемую в данный момент линию, а так же координаты начала линии

Point startPoint;

Line line;

В теле обработчика нажатия на кнопку напишем следующее:

private void canvas\_MouseDown(object sender, MouseButtonEventArgs e)

{

//запоминаем координаты начала отрезка

// метод GetPosition возвращает координаты мыши в данный момент

// относительно элемента canvas

startPoint = e.GetPosition(canvas);

// в зависимости от режима действуем по разному

switch(mode)

{

case DrawMode.PEN:

break;

case DrawMode.LINE:

line = new Line();

line.Stroke = new SolidColorBrush(Colors.Red);

line.X1 = startPoint.X;

line.X2 = startPoint.X;

line.Y1 = startPoint.Y;

line.Y2 = startPoint.Y;

canvas.Children.Add(line);

break;

case DrawMode.ELLIPSE:

break;

}

}

А в обработчике движения мыши следующее:

private void canvas\_MouseMove(object sender, MouseEventArgs e)

{

//обрабатываем движения мыши только если нажата левая кнопка

if(e.LeftButton == MouseButtonState.Pressed)

{

switch (mode)

{

case DrawMode.PEN:

break;

case DrawMode.LINE:

//обновляем координаты второго конца отрезка

line.X2 = e.GetPosition(canvas).X;

line.Y2 = e.GetPosition(canvas).Y;

break;

case DrawMode.ELLIPSE:

break;

}

}

}

И, наконец, в конструкторе формы установим режим по умолчанию:

public MainWindow()

{

InitializeComponent();

mode = DrawMode.LINE;

}

Теперь можно скомпилировать и запустить программу. Если вы все сделали правильно, то должны рисоваться красные прямые линии.

### Кривые линии

Рисование кривых похоже на прямые с небольшими отличиями. В отличие от прямой линии мы не создаем одну линию и изменяем ее координаты при движении мыши, а аппроксимируем кривую большим количеством коротких прямых. То есть, при каждом движении мыши мы будем рисовать короткую прямую линию соединяющую предыдущее положение мыши с текущим.

Добавим в код обработчика начала рисования (canvas\_MouseDown) следующий код:

case DrawMode.PEN:

startPoint = e.GetPosition(canvas);

break;

Говоря проще, при начале рисования мы ничего не делаем, а только сохраняем отправную точку.

Код обработчика движения (canvas\_MouseMove) будет сложнее:

case DrawMode.PEN:

var currentPoint = e.GetPosition(canvas);

line = new Line();

line.Stroke = new SolidColorBrush(Colors.Red);

line.X1 = startPoint.X;

line.Y1 = startPoint.Y;

line.X2 = currentPoint.X;

line.Y2 = currentPoint.Y;

canvas.Children.Add(line);

startPoint = currentPoint;

break;

Тут происходит создание новой линии соединяющий предыдущее и текущее положение мыши и сохранение положения.

Теперь если запустить программу и выбрать режим «Карандаш», должны отрисовываться кривая красная линия.

### Рисование овалов

Рисование овалов не сложнее линий. Для начала объявим в теле класса еще одну переменную Ellipse:

Point startPoint;

Line line;

Ellipse ellipse;

Затем, при начале рисования так же как и с прямой линией создаем объект Овал и добавляем его на экран:

case DrawMode.ELLIPSE:

ellipse = new Ellipse();

ellipse.Stroke = new SolidColorBrush(Colors.Red);

canvas.Children.Add(ellipse);

Canvas.SetTop(ellipse, startPoint.Y);

Canvas.SetLeft(ellipse, startPoint.X);

ellipse.Width = 0;

ellipse.Height = 0;

break;

А при движении мыши будем обновлять его высоту и ширину:

case DrawMode.ELLIPSE:

Point curPoint = e.GetPosition(canvas);

double width = curPoint.X - startPoint.X;

double height = curPoint.Y - startPoint.Y;

if(width < 0) {

width = -width;

Canvas.SetLeft(ellipse, curPoint.X);

}

if(height < 0) {

height = -height;

Canvas.SetTop(ellipse, curPoint.Y);

}

ellipse.Width = width;

ellipse.Height = height;

break;

Ширину и высоту вычисляем как разность между начальным и текущим положением мыши. Так как ширина и высота не может быть отрицательной, необходимо обработать ситуацию, когда мышь после начала рисования идет влево или вверх. В этом случае ширина и высота получаются отрицательные и необходимо сменить верхнюю и левую точки овала (см. условие width < 0 выше).

После этого можно проверить работу программы и убедиться, что овалы рисуются правильно.

### Смена цвета рисования

Так как мы в самом начале уже установили различные цвета для элементов Label, то мы можем их использовать для установки цвета рисования. Так как у Label нет события Click, воспользуемся событием MouseDown. Сперва добавим еще две переменных для хранения основного цвета и цвета заливки для овала:

Point startPoint;

Line line;

Ellipse ellipse;

DrawMode mode;

Color color;

Color fillColor;

Затем на форме выберем один из лейблов, в списке событий найдем MouseMove и кликнем по нему два раза, откроется редактор. Напишем следующий код:

private void lblRed\_MouseDown(object sender, MouseButtonEventArgs e)

{

if (e.LeftButton == MouseButtonState.Pressed)

{

color = ((SolidColorBrush)lblRed.Background).Color;

}

else if(e.RightButton == MouseButtonState.Pressed)

{

fillColor = ((SolidColorBrush)lblRed.Background).Color;

}

}

Для того чтобы получить цвет фона лейбла необходимо его фон привести к типу SolidColorBrush и взять свойство Color. Как не сложно догадаться, данный код при нажатии левой кнопки мыши устанавливает основной цвет, а при нажатии правой кнопки – цвет заливки.

Код аналогичный вышеприведенному можно повторить для каждого из цветов. Но чтобы сэкономить время и место, можно сделать следующее. Обработчик события MouseDown принимает два параметра sender и e. Sender – это ни что иное как отправитель события, и в нашем случае сам лейбл на который было произведено нажатие. Перепишем метод следующим образом:

private void label1\_MouseDown(object sender, MouseButtonEventArgs e)

{

if (e.LeftButton == MouseButtonState.Pressed)

{

color = ((SolidColorBrush)((Label)sender).Background).Color;

}

else if(e.RightButton == MouseButtonState.Pressed)

{

fillColor = ((SolidColorBrush)((Label)sender).Background).Color;

}

}

Здесь мы приводим параметр sender к типу Label, и уже затем берем его свойство Background. После этого, идем в редактор форм, и для каждого элемента Label предназначенного для выбора цвета указываем в качестве обработчика события MouseDown, вышенаписанный обработчик. Для этого вместо двойного щелчка просто вписываем (или вставляем из буфера) название этого обработчика (ну или зовем меня, чтобы я показал, как это сделать).

После этого, можно запустить программу и убедиться, что все работает, как задумано. Можно выбирать один из трех режимов рисования и задавать любой из доступных цветов.

### Задание:

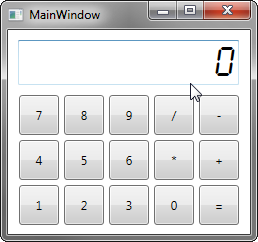
1. Добавить возможность рисовать прямоугольники
2. Добавить 16 цветов
3. Попробовать отключить сохранение текущей позиции мыши в методе MouseMove при рисовании кривых. Добавить такой режим как отдельный.

## Вариант 3. Калькулятор

Разработаем простое приложение «Калькулятор».

### Создание основного окна

В VisualStudio создаем новый проект WPF Application и на основную форму приложения добавляем элементы TextBox и Button так, чтобы получилось похоже на изображение:



После этого зададим всем элементам правильные названия (btn0.. btn9, btnPlus, btnMult и тд).

### Ввод цифр

Перед тем как производить какие-либо вычисления, необходимо ввести числа. С этого и начнем. Для вычислений у нас должны использоваться два числа, объявим для них переменные:

long a;

long b;

Так же у нас будет поддерживаться 4 операции сложение, вычитание, умножение и деление. Объявим новый тип, и перменную этого типа которая будет указывать на выполняемую операцию:

enum Operations

{

NONE,

SUM,

SUBST,

MULT,

DIV

}

Operations operation;

Вводить числа можно либо нажатиями на кнопки приложения, либо с клавиатуры. Для начала сделаем первым способом. Создадим обработчик нажатия на кнопку 1:

private void btn1\_Click(object sender, RoutedEventArgs e)

{

if(operation == Operations.NONE)

{

a = a \* 10 + 1;

textBox.Text = a.ToString();

}

else

{

b = b \* 10 + 1;

textBox.Text = b.ToString();

}

}

Если у нас еще не выбрана операция, значит пользователь вводит первый операнд А, если операция определена, значит пользователь вводит второй операнд B. И цифра 1 прибавляется младшим разрядом к уже введенным цифрам операнда. Для того, чтобы не повторять этот код для каждой из 9 цифр, перепишем его таким образом, чтобы он работал для каждой кнопки:

private void btn1\_Click(object sender, RoutedEventArgs e)

{

if(operation == Operations.NONE)

{

a = a \* 10 + int.Parse(((Button)sender).Content.ToString());

textBox.Text = a.ToString();

}

else

{

b = b \* 10 + int.Parse(((Button)sender).Content.ToString());

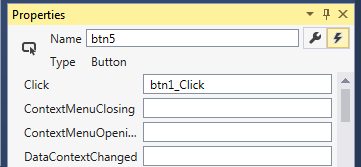
textBox.Text = b.ToString();

}

}

Параметр sender – это отправитель события, в нашем случае – это кнопка, на которую нажал пользователь. Мы берем цифру, написанную на кнопке, преобразуем ее из строки в число и прибавляем младшим разрядом к уже введенным цифрам операнда.

После этого мы должны подключить остальные кнопки цифр к этому обработчику. Для этого переключаемся в режим формы, и в каждой кнопке в поле Click вписываем название метода:



Если теперь запустить программу, то будет вводиться первый операнд и его значение будет отображаться в textBox.

### Выбор операции

Выбор операции – это нажатие на кнопку соответствующую одной из операций. После того как пользователь нажал на нее, мы должны перейти к набору второго операнда, и запомнить, какую операцию выбрал пользователь, чтобы выполнить ее тогда, когда он нажмет на кнопку «Равно». Сделать это очень просто. Щелкнем дважды по кнопке «+» и введем:

private void btnPlus\_Click(object sender, RoutedEventArgs e)

{

operation = Operations.SUM;

}

То же самое повторим для трех остальных кнопок.

### Вычисление выражения

Вычисление выражения производится тогда, когда пользователь нажимает на кнопку «=». В этот момент мы должны вспомнить какая операция была выбрана ранее, и произвести соответствующие вычисления над операндами. Для этого используем switch:

private void btnCalc\_Click(object sender, RoutedEventArgs e)

{

switch(operation)

{

case Operations.SUM:

a = a + b;

break;

case Operations.SUBST:

a = a - b;

break;

case Operations.MULT:

a = a \* b;

break;

case Operations.DIV:

a = a / b;

break;

}

b = 0;

textBox.Text = a.ToString();

}

Тут все довольно очевидно. Для каждой из операций выполняется нужное вычисление, ее результат сохраняется в первом операнде и выводится на экран.

Простейший калькулятор готов.

Задание:

1. Добавить операции «Xy», «1/Х», «X2», sqrt(x), sin(x), cos(x)
2. Добавить возможность выполнения вычислений при повторном нажатии на операцию вместо «=». То есть: вводим первое число, нажимаем +, вводим второе число, нажимаем +, <На экране появляется сумма первых двух и программа ожидает ввод второго операнда. Как это выглядит можно посмотреть на стандартном калькуляторе.
3. Добавить повторение операции при повторном нажатии на «=». То есть: вводим первый операнд, операцию, второй операнд, нажимаем «=», выполняется операция, нажимаем «=» еще раз, операция выполняется еще раз и вместо первого операнда подставляется результат предыдущей операции. Так же можно посмотреть на стандартном калькуляторе.