ЛАБОРАТОРНАЯ РАБОТА

СОЗДАНИЕ MDI-ПРИЛОЖЕНИЙ. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ. СТАНДАРТНЫЕ ДИАЛОГИ

Цель работы:

– изучить особенности разработки MDI-приложений в Visual Studio;
– изучить способы сохранения данных в файл и загрузки из файла; – освоить механизм сериализации и десериализации объектов.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Создать учебные примеры (программы 1–4) и разобрать принцип их работы. Поместить примеры работы программ и их коды с комментариями в отчет.

2. Создать текстовый редактор **NotepadC#**, добавив недостающие пункты меню и функции.

3. На основании руководства к лабораторной работе создать MDI-приложение. Информация в окне должна отображаться в виде таблицы. Иметь возможность делать выборку данных по различным критериям. Переносить данные из одной формы в другую.

4. Добавить формы для ввода дополнительной информации об объекте и фото объекта.

5. Добавить пункты меню для сохранения объектов в файл и загрузки. При сохранении использовать стандартные диалоговые окна и механизм сериализации. В класс добавить поле «Дата создания объекта». Поле не сериализовать, а при десериализации заново устанавливать по системной дате.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Операции ввода/вывода данных в .NET

Поток – это перенос данных между программой и любым устройством ввода/вывода. Данные могут рассматриваться как **поток байтов, символов или объектов**. В пространстве имен **System.IO** есть несколько классов, позволяющих использовать различные устройства хранения, если только данные можно трактовать как байты или символы.

Потоковые классы. Stream — абстрактный класс, который является базовым для чтения и записи байтов в некоторое хранилище данных. Этот класс поддерживает синхронные и асинхронные чтение и запись. Класс Stream содержит типичные методы: **Read** , **Write**, **Seek**, **Flush** и **Close**.

Класс **FileStream**, который является производным от класса Stream, предоставляет операции чтения и записи последовательности байтов в файл. Конструктор FileStream создает экземпляр потока. Перегруженные методы класса Stream осуществляют чтение и запись в файл. У класса Stream есть и другие производные классы: **MemoryStream**, **BufferedStream** и **NetworkStream** (в System.Net.Sockets).

Программа 1. Запись и чтение 10 байтов в/из файл(а). Пример выполнить дважды. Первый раз программа создаст файл и запишет в него числа, а во второй раз прочитает и распечатает часть файла

```
using System.IO;
void main() {
    byte [] data = new byte [10];
    FileStream fs = new FileStream( "FileStreamTest.txt",
    FileMode.OpenOrCreate);
    if (fs.Length == 0)   {
            Console.WriteLine("Writing Data...");
            for (short i = 0; i < 10; i++)
                    data[i] = (byte)i;
            fs.Write(data, 0, 10);   // Запись данных
    }
    else   {
          fs.Seek(-5, SeekOrigin.End);   // Ищем конец
          int count = fs.Read(data, 0, 10);   // Чтение данных
          for (int i = 0; i < count; i++)   Console.WriteLine(data[i]);
    }
    fs.Close();
}
```

Встроенные типы данных и потоки. Если необходимо прочитать в поток или записать из потока простой тип, такой как Boolean, String или Int32, следует использовать классы **BinaryReader** и **BinaryWriter**. Нужно создать соответствующий поток (напимер, FileStream) и передать его в качестве параметра в конструктор BinaryReader или BinaryWriter. Потом можно использовать один из перегруженных методов Read или Write для чтения данных из потока или записи данных в поток.

Программа 2. Запись и чтение 10 чисел в/из файл(а). Пример выполнить дважды. Сначала файл создается, и в него записываются данные. Во второй раз данные читаются из файла.

```
void Main() {

   FileStream fs = new FileStream( "BinaryTest.bin", FileMode.OpenOrCreate);
   if (fs.Length == 0)   {
        Console.WriteLine("Writing Data...");
        BinaryWriter w = new BinaryWriter(fs);
        for (short i = 0; i < 10; i++)     w.Write(i);   // Запись
        w.Close () ;
   } else {
        BinaryReader r = new BinaryReader(fs);
        for (int i = 0; i < 10; i++)   Console.WriteLine(r.ReadInt16());
        r.Close();
   }
   fs.Close();
}
```

Классы TextReader и TextWriter. В абстрактных классах TextReader и TextWriter данные рассматриваются как последовательный поток символов (текст). TextReader имеет следующие методы: Close, Peek (Считывание элемента данных), Read, ReadBlock, ReadLine и ReadToEnd. TextWriter содержит методы типа Close, Flush, Write и WriteLine. Перегруженные методы Read читают символы из потока.

Классы StringReader и **StringWriter** являются производными классами от классов TextReader и TextWriter соответственно. StringReader и StringWriter читают и записывают данные в символьную строку, которая сохраняется в базовом объекте StringBuilder. Конструктор StringWriter может принимать объект StringBuilder.

Классы StreamReader и **StreamWriter** также являются производными классами от классов TextReader и TextWriter. Они читают текст из объекта и записывают текст в объект Stream. Можно создать объект Stream и передать его в конструктор StreamReader или StreamWriter.

Программа 3. Запись и чтение символьных строк в/из файл(а). Программу выполнить дважды: первый раз – чтобы создать файл, а затем второй раз – чтобы прочитать его.

```
void Main() {
```

```
    FileStream fs = new FileStream( "TextTest.txt", FileMode.OpenOrCreate);
    if (fs.Length == 0)      {
            Console.WriteLine("Writing Data..."); // Запись данных
            StreamWriter sw = new StreamWriter(fs);
            sw.Write(100); // Запись
            sw.WriteLine(" One Hundred"); // Сто
            sw.WriteLine("End of File"); // Конец Файла
            sw.Close();
    } else {
            String text; // Строка
            StreamReader sr = new StreamReader(fs) ;
            text = sr.ReadLine(); // текст
            while (text != null)   {
                    Console.WriteLine(text); text = sr.ReadLine();
            }
            sr.Close ();
    }
    fs.Close ();
}
```

Классы File и FileInfo. Класс File содержит методы для создания и открытия файлов, которые возвращают объекты FileStream, StreamWriter или StreamReader, производящие фактическое чтение и запись. Перегруженный метод Create возвращает объект FileStream. Метод CreateText возвращает StreamWriter. Перегруженный метод Open в зависимости от передаваемых параметров может создавать новый файл или открывать существующий для чтения или записи. Возвращаемый объект – объект FileStream. Метод OpenText возвращает StreamReader. Метод OpenRead возвращает объект FileStream. Метод OpenWrite возвращает объект типа FileStream.

Классы File и FileInfo содержат также методы копирования, удаления, перемещения файлов, проверки существования файла, чтения и изменения атрибутов файла (время создания; время последнего обращения; последнее время записи; архивный, скрытый, обычный, системный или временный; сжатый, зашифрованный; только для чтения; файл или каталог).

Программа 4. Пример использования методов класса File и FileInfo.

```
void Main() {
    File.Delete("file2.txt");        // Удалить файл "file2.txt"
```

```
    StreamWriter sw = System.IO.File.CreateText("file.txt");
    sw.Write ("Пусть каждый день твой будет светлым, ");
    sw.WriteLine("приятным, радостным и щедрым!");
    sw.Close();
    File.Move("file.txt", "file2.txt");      // Переименование
    FileInfo fileInfo = new FileInfo("file2.txt");
    Console.WriteLine( "File {0} is {1} bytes in length, created on {2}",
    fileInfo.FullName,fileInfo.Length, fileInfo.CreationTime);
    Console.WriteLine("");
    StreamReader sr = fileInfo.OpenText();
    String s = sr.ReadLine();
    while (s != null)   {
            Console.WriteLine(s); s = sr.ReadLine();
    }
    sr.Close();
}
```

Сериализация объектов. Сериализация преобразовывает объекты, такие как классы, структуры и массивы в поток байтов. При преобразовании из последовательной формы в параллельную поток байтов преобразовывается обратно в объекты.

Сериализация может осуществляться в двух форматах: **бинарном и XML-формате**. Пространство имен **System.Runtime. Serialization** используется для сериализации в общей системе типов. Пространство имен **System.Xml.Serialization** используется для сериализации XML-документов.

Чтобы информировать каркас, что класс может быть преобразован в последовательную форму, нужно пометить класс атрибутом **[Serializable]**. Любое поле или свойство, которые не должны быть преобразованы в последовательную форму, могут быть отмечены атрибутом [**NonSerialized**].

```
[Serializable]      // Объявление сериализации объектов класса

public class Personage{
    public Personage(string name, int age)     {
            this.name = name; this.age = age; } static int wishes;
    public string name, status, wealth;
    int age;
    public Personage couple;
    void SaveState(){        //сериализация в бинарном формате
            BinaryFormatter bf = new BinaryFormatter();
        FileStream fs = new FileStream ("State.bin",FileMode.Create,
        FileAccess.Write);
```

```
            bf.Serialize(fs,this);
            fs.Close();
        }
        void BackState(ref Personage fisher){    //десериализация
            BinaryFormatter bf = new BinaryFormatter();
            FileStream fs  =  new  FileStream ("State.bin",FileMode.Open,
            FileAc-cess.Read);
            fisher = (Personage)bf.Deserialize(fs);
            fs.Close();
        }
    }
```

РАЗРАБОТКА МНОГООКОННОГО ПРИЛОЖЕНИЯ

Разберем создание MDI (Multiple Document Interface, многодокументный интерфейс) приложений на примере разработки программы «Блокнот».

1. Создайте новое приложение (**программа 5**) и назовите его NotepadC#. Установите следующие свойства формы (табл. 1)

Таблица 1

Свойства формы NotepadC#

Форма, свойства	Значение
Name	Frmmain
Icon	Путь C:\Program Files\Microsoft Visual Studio....
Text	NotepadC#
WindowState	Maximized

2. Создайте меню приложения со следующими пунктами (рис.1). Для этого вызовите контекстное меню, установив курсор мыши на компоненте menuStrip1, расположенной на панели невидимых компонент, и выберите пункт Edit Items. Каждый пункт главного меню имеет свое окно свойств, в котором задаются значения свойств Name и Text(рис.2). В поле Text перед словом New стоит знак & – так называемый амперсанд, указывающий, что N должно быть подчеркнуто и будет частью встроенного клавиатурного интерфейса Windows. Когда пользователь на клавиатуре нажимает клавишу Alt и затем N, выводится подменю New.

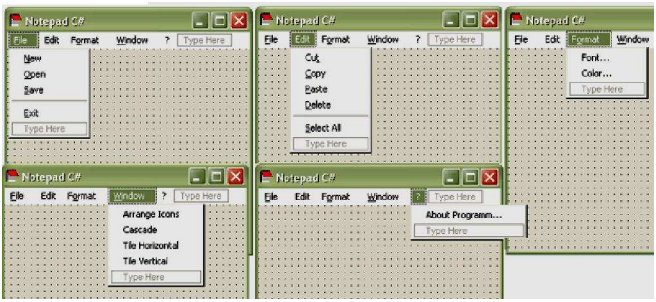

Рис. 1. Пункты главного меню приложения NotepadC#

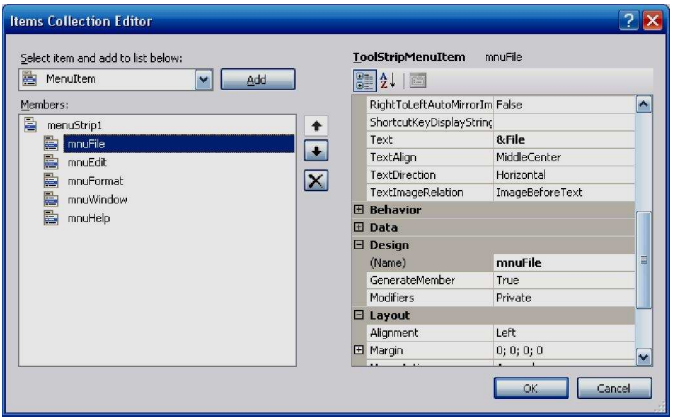

Рис. 2. Свойства пункта меню New

Свойства пунктов меню в приложении NotepadC# приводятся в табл. 2.

Таблица 2

Пункты главного меню приложения NotepadC#

Name	Text	Shortcut
mnuFile	&File	
mnuNew	&New	Ctrl+N
mnuOpen	&Open	Ctrl+O
mnuSave	&Save	Ctrl+S
menultem5	-	
mnuExit	&Exit	Alt+F4
mnuEdit	&Edit	
mnuCut	Cu&t	Ctrl+X
mnuCopy	&Copy	Ctrl+C
mnuPaste	&Paste	Ctrl+V
mnuDelete	&Delete	Del
mnuSelectAll	&SelectAll	Ctrl+A
mnuForrmt	F&ormat	
mnuFont	Font…	
mnuColor	Color…	
mnuWindow	&Window	
mnuArrangelcons	Arrange Icons	
mnuCascade	Cascade	
mnuTileHorizontal	Tile Horizontal	
mnuTileVertical	Tile Vertical	
mnuHelp	?	
mnuAbout	About Program...	

3. В MDI-приложениях главная форма содержит в себе несколько документов, каждый из которых является холстом в графических программах или полем для текста в редакторах. В окне Solution Explorer щелкаем правой кнопкой на имени проекта и в появившемся контекстном меню выбираем Add/New Item/ Windows Form. В появившемся окне называем форму – blank.h. В нашем проекте появилась новая форма – будем называть ее **дочерней**. В режиме дизайна перетаскиваем на нее элемент управления RichTextBox, размер содержимого текста в нем не ограничен 64 Кб; кроме того, RichTextBox позволяет редактировать цвет текста, добавлять изображения. Свойству Dock этого элемента устанавливаем значение Fill.

Переходим в режим дизайна формы frmmain и устанавливаем свойству **IsMdiContainer** значение **true**. Цвет формы при этом становится темно-серым. Новые документы будут у нас появляться при нажатии

пункта меню New, поэтому дважды щелкаем в этом пункте и переходим в код.

При создании нескольких документов будем называть их ДокументN, где N – номер документа. Переключаемся в код формы blank, и в классе blank объявляем поле

public String DocName;

Переключаемся в код формы frmmain и в классе frmmain объявляем переменную **private int openDoc;**

Присваиваем переменной DocName часть названия по шаблону, в который включен счетчик числа открываемых документов, затем это значение передаем свойству Text создаваемой формы frm:

```
private void mnunew_Click(object   sender, EventArgs   e)
{       frm = new blank();
        frm.DocName = "Документ " + ++openDoc;
        frm.Text = frm.DocName; frm.MdiParent = this;
        frm.Show();
}
```

4. Для каждого пункта меню пишем обработчики событий, выполняющие соответсвтвующие пункту действия.

Перечисление MdiLayout. При работе с несколькими документами в MDI-приложениях удобно упорядочивать их на экране. В пункте меню Window реализуем процедуру выравнивания окон. Создаем обработчики:

```
private void mnuArrangeIcons_Click(object sender, EventArgs e){
     this.LayoutMdi(MdiLayout.ArrangeIcons); }
private void mnuCascade_Click(Object sender, EventArgs e)     {
      this.LayoutMdi(MdiLayout.Cascade); }
private void mnuTileHorizontal_Click(object sender, EventArgs e) {
       this.LayoutMdi(MdiLayout.TileHorizontal); }
private void mnuTileVertical_Click(object sender, EventArgs e)     {
       this.LayoutMdi(MdiLayout.TileVertical); }
```

Метод LayoutMdi содержит перечисление MdiLayout, содержащее четыре члена. ArrangeIcons переключает фокус на выбранную форму, в свойстве MdiList пункта меню ArrangeIcons устанавливаем также значение true. При открытии нескольких новых документов окна располагаются каскадом, их можно расположить горизонтально – значение TileHorizontal или вертикально – значение TileVertical.

Вырезание, копирование и вставка текстовых фрагментов. При

создании нового документа он сразу будет занимать всю область главной формы. Для этого установим свойство WindowState формы blank Maximized. Создадим обработчики для стандартных операций вырезания, копирования и вставки. Элемент управления RichTextBox имеет свойство SelectedText, которое содержит выделенный фрагмент текста. Оно будет использоваться при работе с текстом. В коде формы blank объявляем переменную, в которой будет храниться буферизованный фрагмент текста: **private String BufferText**;

Далее создаем соответствующие методы:

```
public void Cut() {  // Вырезание текста
    this.BufferText = richTextBox1.SelectedText;
    richTextBox1.SelectedText = “”;}
public void Copy() {                 // Копирование текста
    this.BufferText = richTextBox1.SelectedText;}
public void Paste() {                // Вставка
    richTextBox1.SelectedText = this.BufferText; }
   // Выделение всего текста – используем свойство SelectAll элемента
управления RichTextBox
public void SelectAll() {
    richTextBox1.SelectAll();}
public void Delete() {       // Удаление
richTextBox1.SelectedText = “”;
this.BufferText = “”;
}
```

Переключаемся в режим дизайна формы и создаем обработчики для пунктов меню:

```
private void mnucut_Click(object sender, EventArgs e) {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.Cut(); }
private void mnucopy_Click(object sender, EventArgs e) {
    blank frm = (blank)(this.ActiveMdiChild);
    frm.Copy(); }
private void mnuselectAll_Click(object sender, EventArgs e) {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.SelectAll(); }
private void mnupaste_Click(object sender, EventArgs e) {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.Paste(); }
```

```
private void mnudelete_Click(object sender, EventArgs e) {
blank frm = (blank) (this.ActiveMdiChild);
    frm.Delete(); }
```

Свойство **ActiveMdiChild** переключает фокус на текущую форму, если их открыто несколько, и вызывает один из методов, определенных в дочерней форме. Теперь можно выполнять все стандартные операции с текстом.

Контекстное меню. Дублирует некоторые действия основного меню. Добавим элемент управления contextMenuStrip из окна ToolBox на форму blank. Добавляем пункты контекстного меню точно так же, как мы это делали для главного.

Свойства Text и Shortcut пунктов меню оставляем прежними. Если мы установим затем для свойства ShowShortcut значение false, то сочетания клавиш будут работать, но в самом меню отображаться не будут. Свойства Name будут следующими: для пункта Cut – cmnuCut, для Copy – cmnuCopy и т. д.

В обработчике пунктов просто вызываем соответствующие методы:

```
private void cmnuCut_Click(object sender, EventArgs e) {
    Cut(); }
private void cmnuCopy_Click(object sender, EventArgs e) {
    Copy();}
private void cmnuPaste_Click(object sender, EventArgs e)      {
    Paste(); }
private void cmnuDelete_Click(object sender, EventArgs e) {
    Delete(); }
private void cmnuSelectAll_Click(object sender, EventArgs e) {
    SelectAll();}
```

Необходимо определить, где будет появляться контекстное меню. Элемент RichTextBox, так же как и формы frmmain и blank, имеет свойство ContextMenuStrip, где мы и указываем contextMenuStrip1, поскольку нам нужно отображать меню именно в текстовом поле. Запускаем приложение – теперь в любой точке текста доступно меню.

5. Для пункта Format создать обработчики выбора цвета и шрифта самостоятельно.

ДИАЛОГОВЫЕ ОКНА

Выполняя различные операции с документом – открытие, сохранение, печать, предварительный просмотр, мы сталкиваемся с соответствующими диалоговыми окнами. Разработчикам .NET не приходится заниматься созданием окон стандартных процедур: элементы **OpenFileDialog, SaveFile Dialog, ColorDialog, PrintDialog** содержат уже готовые операции.

OpenFileDialog. Добавьте на форму frmmain элемент управления OpenFileDialog из окна панели инструментов ToolBox.

Свойство FileName задает название файла, которое будет находиться в поле "Имя файла:" при появлении диалога. Свойство Filter задает ограничение файлов, которые могут быть выбраны для открытия. Через вертикальную разделительную линию можно задать смену типа расширения, отображаемого в выпадающем списке "Тип файлов". Зададим Text Files (*.txt)|*.txt|All Files(*.*)|*.*, что означает обзор либо текстовых файлов, либо всех. Свойство InitialDirectory позволяет задать директорию, откуда будет начинаться обзор. Если свойство не установлено, исходной директорией будет рабочий стол.

Для работы с файловыми потоками в коде формы blank подключаем пространство имен System.IO. Создаем метод Open:

```
public void Open(String OpenFileName)   {
     if (OpenFileName == null) {      return;       }
     else   {
          StreamReader sr = new StreamReader(OpenFileName);
          richTextBox1.Text = sr.ReadToEnd(); sr.Close();
          DocName = OpenFileName;
     }
}
```

Добавим обработчик пункта меню Open формы frmmain:

```
private void mnuopen_Click(object   sender, EventArgs   e)

{ // задание доступных расширений файлов программно
openFileDialog1.Filter = "Text Files (*.txt)|*.txt|All Files(*.*)|*.*";
if (openFileDialog1.ShowDialog() == .DialogResult.OK) {
     blank frm = new blank(); frm.Open(openFileDialog1.FileName);
     frm.MdiParent = this;    //Указываем родительскую форму
```

```
    //Присваиваем переменной DocName имя открываемого файла
    frm.DocName = openFileDialog1.FileName;
    //Свойству Text формы присваиваем переменную DocName
    frm.Text = frm.DocName;
    frm.Show();
}
}
```

Для корректного чтения кириллицы текст в блокноте должен быть сохранен в кодировке Unicode.

SaveFileDialog. Для сохранения файлов добавляем на форму frmmain элемент управления saveFileDialog1. Свойства этого диалога такие же, как у OpenFileDialog. Переходим в код формы blank и создаем метод Save:

```
public void Save(String SaveFileName)   {
    if (SaveFileName == null) {       return;    }
    else   {
            StreamWriter sw = new StreamWriter(SaveFileName);
            sw.WriteLine(richTextBox1.Text); sw.Close(); //Устанавливаем
            имя документа DocName = SaveFileName;
    }
}
```

Добавляем обработчик пункта меню Save формы frmmain:

```
private void mnuSave_Click(object   sender, EventArgs   e)    {

   saveFileDialog1.Filter = "Text Files (*.txt)|*.txt|All Files(*.*)|*.*";
   if (saveFileDialog1.ShowDialog() == .DialogResult.OK)
     { blank frm = (blank)(this.ActiveMdiChild);
     frm.Save(saveFileDialog1.FileName); frm.MdiParent = this;
     frm.DocName = saveFileDialog1.FileName; frm.Text = frm.DocName;
    }
}
```

При сохранении внесенных изменений в уже сохраненном файле вместо его перезаписи вновь появляется окно SaveFileDialog. Изменим программу так, чтобы можно было сохранять и перезаписывать файл. В конструкторе формы frmmain после InitializeComponent отключим доступность пункта меню Save: **mnuSave.Enabled = false;**

Переключаемся в режим дизайна формы frmmain и добавляем

пункт меню **Save As** после пункта Save. Устанавливаем следующие свойства этого пункта: Name – mnuSaveAs, Shortcut – Ctrl+Shift+S, Text Save &As. В обработчике Save As вставляем вырезанный обработчик пункта Save и добавляем включение доступности Save:

mnuSave.Enabled = true;

Сохранять изменения требуется как в только что сохраненных документах, так и в документах, созданных ранее и открытых для редактирования. Поэтому добавим в метод Open включение доступности пункта меню Save:

```
private void mnuOpen_Click(object sender, EventArgs e) {
    mnuSave.Enabled = true; }
```

В обработчике пункта Save добавим простую перезапись файла – вызов метода Save формы blank:

```
private void mnuSave_Click(object  sender, EventArgs  e)    {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.Save(frm.DocName); }
```

Теперь, если мы работаем с несохраненным документом, пункт Save неактивен, после сохранения он становится активным и, кроме того, работает сочетание клавиш Ctrl+S. Можно сохранить копию текущего документа, вновь воспользовавшись пунктом меню Save As.

Сохранение файла при закрытии формы. Всякий раз, когда мы закрываем документ Microsoft Word, в который внесли изменения, появляется окно предупреждения, предлагающее сохранить документ. Добавим аналогичную функцию в наше приложение. В классе blank создаем переменную, которая будет фиксировать сохранение документа:

public bool IsSaved = false;

В обработчик методов Save и Save As формы frmmain добавляем изменение значения этой переменной:

```
private void mnuSave_Click(object sender, EventArgs e) {
    …
    frm.IsSaved = true; }
private void mnuSaveAs_Click(object sender, EventArgs e) {
    …
    frm.IsSaved = true; }
```

Переходим в режим дизайна формы blank, и в окне свойств переключаемся на события формы, щелкнув на значок с молнией. В поле

события FormClosed дважды щелкаем и переходим в код:

```
private void blank_FormClosed(object sender, FormClosedEventArgs e)
{ if(IsSaved ==true)
        if(MessageBox.Show("Do you want save changes in "
        + this.DocName + "?", "Message", MessageBoxButtons.YesNo,
        MessageBoxIcon.Question) == .DialogResult.Yes) {
                this.Save(this.DocName);
        }
}
```

OpenFileDialog и SaveFileDialog для SDI-приложений. При создании MDI-приложений приходится разделять код для открытия и сохранения файлов, как мы делали для приложения NotepadC#. В случае SDI-приложений весь код будет находиться в одном обработчике. Создаем новое приложение, называем его TextEditor. На форме размещаем элемент управления TextBox и устанавливаем следующие свойства (табл.3).

Таблица 3

Свойства элемента управления TextBox

TextBox, свойство	Значение
Name	txfflox
Dock	Fill
Multiline	true
Text	Да

Добавляем на форму элемент menuStrip1, в котором будет всего три пункта – File, Open и Save (свойства пунктов см. в табл. 2). Из окна ToolBox перетаскиваем элементы OpenFileDialog и SaveFileDialog – свойства этих элементов в точности такие же, как и у диалогов приложения NotepadC#. Переходим в код формы. Добавляем обработчик для пункта меню Open:

```
private void mnuOpen_Click(object sender, EventArgs e)
{
   openFileDialog1.ShowDialog();
   String fileName = openFileDialog1.FileName;
   FileStream filestream=File.Open(fileName, FileMode.Open, Fi-leAccess.Read);
   if(filestream != null) {
        StreamReader streamreader = new StreamReader(filestream);
```

```
        txtBox.Text = streamreader.ReadToEnd();
        filestream.Close();
    }
}
```

Добавляем обработчик для пункта меню Save:

```
private void mnuSave_Click(object sender, EventArgs e)
{  saveFileDialog1.ShowDialog();
   String fileName=saveFileDialog1.FileName;
   FileStream filestream=File.Open(fileName, FileMode.Create,
   Fi-leAccess.Write);
   if(filestream != null) {
        StreamWriter streamwriter = new StreamWriter(filestream);
        streamwriter.Write(txtBox.Text);
        streamwriter.Flush();
        filestream.Close();
   }
}
```

ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ

1. Что такое поток? Какой класс является родоначальником всех потоков?
2. Какие бывают потоки?
3. В каких форматах можно сохранять файловые потоки?
4. Режимы работы с файлом.
5. Основные методы работы с файлом.
6. Какие возможности имеют классы **File**, **FileInfo?**
7. Что такое сериализация? Для чего она применяется?
8. Что такое десериализация? Для чего она применяется?
9. Как задать сериализацию объектов класса?
10. В каких форматах можно сериализовать данные?
11. Как исключить некоторые свойства объекта при сериализации?
12. Как десериализовать объект?
13. Что такое MDI-приложение? Как создать такое приложение?
14. Что такое контекстно-зависимое меню? Как создать контекстно-зависимое меню?
15. Какая компонента позволяет отображать на форме рисунок?
16. Какая компонента служит для ввода текста и многострочного текста на форме?
17. Как создать и вызвать стандартные диалоговые окна: подтверждение действия, сохранение в файл, загрузка из файла?