Лабораторная работа №2

ХАРЬКОВСКИЙ НАЦИОНАЛЬНЫЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ МЕХАТРОНИКИ ТРАНСПОРТНЫХ СРЕДСТВ

Кафедра информатики

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по проведению лабораторных работ по дисциплине «Программирование» для студентов специальности 6.050201 “Системная инженерия”

Разработчик - доцент кафедры информатики
кандидат технических наук,
старший научный сотрудник
Тимонин Владимир Алексеевич

Харків 2011

Лабораторная работа №2
Исследование возможностей интегрированной среды разработки Visual C# для создания приложений по обработке текстовых файлов.

Цель работы – исследовать возможности интегрированной среды разработки Visual Studio 2010 и получить практические навыки по созданию приложений по обработке текстовых файлов.

1. Теоретические сведения

1.1. Файловый ввод-вывод на побайтовой основе

В C# предусмотрены классы, которые позволяют считывать содержимое файлов и записывать в них информацию. Так как на уровне операционной системы все файлы обрабатываются на побайтовой основе, то в C# определены методы, предназначенные для считывания байтов из файла и записи байтов в файл. Файловые операции, ориентированные на символы, используются в случае текстовых файлов.

Чтобы создать байтовый поток с привязкой к файлу, используется класс FileStream, являющийся производным от Stream и обладающий функциональными возможностями базового класса. Так как потоковые классы, включая FileStream, определены в пространстве имен System.IO, то при их использовании в начало программы необходимо включить оператор:

using System.IO;.

1.1.1. Открытие и закрытие файла

Чтобы создать байтовый поток, связанный с файлом, необходимо создать объект класса FileStream. В классе FileStream определено несколько конструкторов. Чаще всего из них используется конструктор, который открывает файл для доступа с разрешением чтения и записи:

FileStream(string filename, FileMode mode);

где параметр **filename** означает имя файла, который необходимо открыть, причем оно может включать полный путь к файлу; параметр **mode** означает, как именно должен быть открыт этот файл, или режим открытия. Параметр **mode** может принимать одно из значений, определенных перечислением FileMode (они описаны в табл. 1).

Таблица 1. Значения перечисления FileMode

Значение	Описание
FileMode.Append	Добавляет выходные данные в конец файла
FileMode.Create	Создает новый выходной файл. Существующий файл с таким же именем будет разрушен
FileMode.CreateNew	Создает новый выходной файл. Файл с таким же именем не должен существовать
FileMode.Open	Открывает существующий файл
FileMode.OpenOrCreate	Открывает файл, если он существует. В противном случае создает новый
FileMode.Truncate	Открывает существующий файл, но усекает его длину до нуля

Если попытка открыть файл оказалось неуспешной, генерируется исключение. Если файл невозможно открыть по причине его отсутствия, генерируется исключение типа FileNotFoundException. Если файл невозможно открыть из-за ошибки ввода-вывода, генерируется исключение типа IOException. Возможны также исключения следующих типов: ArgumentNullException (если имя файла представляет собой null-значение), ArgumentException (если некорректен параметр mode), SecurityException (если пользователь не обладает правами доступа) и DirectoryNotFoundException (если некорректно задан каталог).

Пример 1. Нижеприведенный фрагмент кода демонстрирует один из способов открытия файл **test.dat** для ввода данных:

```
FileStream fin;
try {
     fin = new FileStream("test.dat", FileMode.Open);
   }
catch (FileNotFoundException exc)
     {
      MessageBox.Show(exc.Message, "Отсутствие файла ");
      return;
     }
catch {
       MessageBox.Show("Не удается открыть файл!",
                              "Ошибка ввода-вывода");
        return;
       }
```

Здесь первая catch-инструкция перехватывает ошибку, связанную с отсутствием файла. Вторая, предназначенная для "всеобщего перехвата", обрабатывает другие ошибки, которые возможны при работе с файлами.

Если необходимо ограничить доступ только чтением или только записью, используется конструктор:

FileStream(string filename, FileMode mode, FileAccess how);

где параметр **filename** означает имя открываемого файла, а **mode** — способ его открытия. Значение, передаваемое с помощью параметра **how**, определяет способ доступа к файлу. Этот параметр может принимать одно из значений, определенных перечислением FileAccess (FileAccess.Read; FileAccess.Write; FileAccess.ReadWrite).

Например, при выполнении нижеприведенного оператора файл test.dat будет открыт только для чтения:

FileStream fin = new FileStream("test.dat", FileMode.Open, FileAccess.Read);

По завершении работы с файлом его необходимо закрыть. Для этого достаточно вызвать метод **Close()**. Его общая форма вызова имеет такой вид:

void Close();

При закрытии файла освобождаются системные ресурсы, ранее выделенные для этого файла, что дает возможность использовать их для других файлов. Метод **Close()** может генерировать исключение типа IOException.

1.1.2. Считывание байтов из файла

В классе FileStream определены два метода, которые считывают байты из файла: **ReadByte()** и **Read()**. Чтобы прочитать из файла один байт, используется метод **ReadByte()**, общая форма вызова которого имеет следующий вид:

int ReadByte();

При каждом вызове этого метода из файла считывается один байт, и метод возвращает его как целочисленное значение. При обнаружении конца файла метод возвращает **-1**. Метод может генерировать исключения типов NotSupportedExeption (поток не открыт для ввода) и ObjectDisposedException (поток закрыт).

Чтобы считать блок байтов, используется метод **Read()**, общая форма вызова которого имеет следующий вид:

int Read(byte[] buf, int offset, int numBytes);

Метод **Read()** пытается считать **numBytes** байтов в массив **buf,** начиная с элемента **buf [offset]**. Он возвращает количество успешно считанных байтов. При возникновении ошибки ввода-вывода

генерируется исключение типа IOException. Помимо прочих, возможно также генерирование исключения типа NotSupportedException, если используемый поток не поддерживает операцию считывания данных.

 **Пример 2.** Нижеприведенный пример демонстрирует использование метода **ReadByte()** для отображения содержимого текстового файла. В catch-блоке обрабатывается ошибка при отсутствии файла: "указанный файл не найден".

```
private void button1_Click(object sender, EventArgs e)
    {
      int i;
      FileStream fin;
      try
      {
        fin = new FileStream("Program.cs", FileMode.Open);
      }
      catch (FileNotFoundException exc)
      {
        MessageBox.Show(exc.Message, "Отсутствие файла ");
        return;
      }
      // Считываем байты до тех пор, пока не встретится EOF.
      do
      {
        try
        {
          i = fin.ReadByte();
        }
        catch (Exception exc)
        {
          MessageBox.Show(exc.Message);
          return;
        }
        if (i != -1) label1.Text += "  " + (char)(byte)i;
      } while (i != -1);
      fin.Close();
    }
```

1.1.3. Запись данных в файл

Чтобы записать байт в файл, используется метод WriteByte(), форма вызова которого имеет следующий вид:

void WriteByte(byte val);

Этот метод записывает в файл байт, заданный параметром **val.** При возникновении во время записи ошибки генерируется исключение типа IOException. Если соответствующий поток не открыт для вывода данных, генерируется исключение типа NotSupportedException.

С помощью метода Write() можно записать в файл массив байтов. Это делается следующим образом:

void Write(byte[] buf, int offset, int numBytes);

Метод Write () записывает в файл numBytes байтов из массива buf, начиная с элемента buf [offset]. При возникновении во время записи ошибки генерируется исключение типа IOException. Если соответствующий поток не открыт для вывода данных, генерируется исключение типа NotSupportedException. Возможны и другие исключения.

При выполнении операции вывода в файл выводимые данные зачастую не записываются немедленно на реальное физическое устройство, а буферизируются операционной системой до тех пор, пока не накопится порция данных достаточного размера, чтобы ее можно было всю сразу переписать на диск. Такой способ выполнения записи данных на диск повышает эффективность системы. Например, дисковые файлы организованы по секторам, которые могут иметь размер от 128 байт. Данные, предназначенные для вывода, обычно буферизируются до тех пор, пока не накопится такой их объем, который позволяет заполнить сразу весь сектор. Если необходимо записать данные на физическое устройство вне зависимости от того, полон буфер или нет, то используется метод **Flush()**. В случае неудачного исхода операции записи генерируется исключение типаIOException.

Завершив работу с выходным файлом, вы должны его закрыть с помощью метода **Close()**. Это гарантирует, что любые данные, оставшиеся в дисковом буфере, будут переписаны на диск. Поэтому перед закрытием файла нет необходимости специально вызывать метод **Flush()**.


Пример 3. Нижеприведенный пример демонстрирует запись данных в файл. Эта приложение сначала открывает для вывода файл с именем test.txt. Затем в этот файл записывается алфавит английского языка, после чего файл закрывается. Возможные ошибки обрабатываются с помощью блоков try/catch.

```
private void button1_Click(object sender, EventArgs e)
    {
       FileStream fout;
       // Открываем выходной файл,
       try
       {
          fout = new FileStream("test.txt", FileMode.Create);
       }
       catch (IOException exc)
       {
          MessageBox.Show(exc.Message,"Ошибка при открытии файла");
          return;
       }
       // Записываем в файл алфавит,
       try
       {
          for (char c = 'A'; c <= 'Z'; c++)
             fout.WriteByte((byte)c);
       }
       catch (IOException exc)
       {
          MessageBox.Show(exc.Message, "Ошибка при записи в файл");
          fout.Close();
       }
    }
```

1.1.4. Использование класса FileStream для копирования файла

Одно из достоинств байтового ввода-вывода с использованием класса FileStream заключается в том, что этот класс можно использовать для всех типов файлов, а не только текстовых.

Пример 4. Нижеприведенный пример демонстрирует операцию копирования файла любого типа, включая выполняемые файлы.

```
private void button1_Click(object sender, EventArgs e)
    {
      int i ;
      FileStream fin, fout;
      // Открываем входной файл,
      try {
           fin = new FileStream("program.cs", FileMode.Open);
          }
      catch (FileNotFoundException exc)
          {
           MessageBox.Show(exc.Message, "Отсутствие файла");
           return;
          }
      // Открываем выходной файл,
      try {
           fout = new FileStream("Copy.cs", FileMode.Create);
          }
      catch (IOException exc)
          {
            MessageBox.Show(exc.Message,"Ошибка при создании файла");
            return;
          }
      // Копируем файл,
      try
      {
        do
        {
          i = fin.ReadByte();
          if (i != -1) fout.WriteByte((byte)i);
        } while (i != -1);
      }
      catch (IOException exc)
      {
        MessageBox.Show(exc.Message, "Ошибка при чтении файла");
      }
      fin.Close();
      fout.Close();
    }
```

1.2. Файловый ввод-вывод с ориентацией на символы

Несмотря на то, что байтовая обработка файлов получила широкое распространение, C# также поддерживает символьные потоки, которые работают непосредственно с Unicode-символами. Поэтому, если необходимо сохранить Unicode-текст, лучше всего выбрать именно символьные потоки. В общем случае, чтобы выполнять файловые операции на символьной основе, поместите объект класса FileStream внутрь объекта класса StreamReader или класса StreamWriter. Эти классы автоматически преобразуют байтовый поток в символьный и наоборот.

Класс StreamWriter — производный от класса TextWriter, а StreamReader —производный от TextReader. Следовательно, классы StreamWriter и StreamReader имеют доступ к методам и свойствам, определенным их базовыми классами.

1.2.1. Использование класса StreamWriter

Чтобы создать выходной поток для работы с символами, необходимо поместить объект класса Stream (например, FileStream) в объект класса StreamWriter. В классе StreamWriter определено несколько конструкторов. Самый популярный из них выглядит следующим образом:

StreamWriter(Stream stream);

где параметр **stream** означает имя открытого потока. Этот конструктор генерирует исключение типа Argument Except ion, если поток **stream** не открыт для вывода, и исключение типа ArgumentNullException, если он (поток) имеет null-значение.

Созданный объект класса StreamWriter автоматически выполняет преобразование символов в байты.

Пример 5. Нижеприведенный пример демонстрирует использование класса StreamWriter. Рассмотрим простую утилиту "клавиатура-диск", которая считывает строки текста, вводимые с клавиатуры, и записывает их в файл test.txt.

```
private void button1_Click(object sender, EventArgs e)
    {
      FileStream fout;
      try
      {
        fout = new FileStream("test.txt", FileMode.Create);
      }
      catch (IOException exc)
      {
        MessageBox.Show(exc.Message, "Не удается открыть файл");
        return;
      }
      StreamWriter fstr_out = new StreamWriter(fout);
      try
      {
        fstr_out.Write(textBox1.Text);
      }
      catch (IOException exc)
      {
         MessageBox.Show(exc.Message, "Ошибка при работе с файлом");
         return;
      }
      fstr_out.Close();
    }
```

Иногда удобнее открывать файл с помощью класса StreamWriter. Для этого используется один из следующих конструкторов:

StreamWriter(string filename);

StreamWriter(string filename, bool appendFlag);

где параметр **filename** означает имя открываемого файла (имя может включать полный путь к файлу), параметр **appendFlag** типа **bool** (если **appendFlag** равен значению **true**, выводимые данные добавляются в конец существующего файла. В противном случае заданный файл перезаписывается).

В обоих случаях, если файл не существует, он создается, а при возникновении ошибки ввода-вывода генерируется исключение типа IOException (также возможны и другие исключения).

Пример 6. Нижеприведенный пример демонстрирует использование класса StreamWriter для открытия файла напрямую (модифицированная версия примера 5).

```
private void button1_Click(object sender, EventArgs e)
    {
      StreamWriter fstr_out;
      try
      {
        fstr_out = new StreamWriter("test.txt");
      }
      catch (IOException exc)
      {
        MessageBox.Show(exc.Message, "Не удается открыть файл");
        return;
      }
      try
      {
        fstr_out.Write(textBox1.Text);
      }
      catch (IOException exc)
      {
         MessageBox.Show(exc.Message, "Ошибка при работе с файлом");
         return;
      }
      fstr_out.Close();
    }
```

1.2.2. Использование класса StreamReader

Чтобы создать входной поток с ориентацией на обработку символов, необходимо поместить байтовый поток в класс-оболочку StreamReader. В классе StreamReader определено несколько конструкторов. Чаще всего используется конструктор:

StreamReader(Stream stream);

где параметр **stream** означает имя открытого потока. Этот конструктор генерирует исключение типа ArgumentNullException, если поток **stream** имеет null-значение, и исключение типа ArgumentException, если поток **stream** не открыт для ввода. После создания объект класса StreamReader автоматически преобразует байты в символы.

Пример 7. Нижеприведенный пример демонстрирует использование класса StreamWriter. Рассмотрим простую утилиту "клавиатура-диск", которая считывает текстовый файл test.txt и отображает его содержимое на экране.

```
private void button1_Click(object sender, EventArgs e)
    {
      FileStream fin;
      string s;
      try
      {
        fin = new FileStream("test.txt", FileMode.Open);
      }
      catch (FileNotFoundException exc)
      {
        MessageBox.Show(exc.Message, "Не удается открыть файл");
```

```
            return;
        }
        StreamReader fstr_in = new StreamReader(fin);
        while ((s = fstr_in.ReadLine()) != null)
        {
            listBox1.Items.Add(s);
        }
        fstr_in.Close();
    }
```

Обратите внимание на то, как определяется конец файла. Если ссылка, возвращаемая методом ReadLine(), равна значению **null**, значит, конец файла достигнут.

Как и в случае класса StreamWriter, иногда проще открыть файл, напрямую используя класс StreamReader. Для этого обратитесь к этому конструктору.

StreamReader(string filename);

где параметр **filename** означает имя открываемого файла, которое может включать полный путь к файлу. Указанный файл должен существовать. В противном случае генерируется исключение типа FileNotFoundException. Если параметр **filename** равен значению **null**, генерируется исключение типа ArgumentNullException, а если он представляет собой пустую строку, — исключение типа ArgumentException.

2. Рабочее задание

Задание 1. Разработать приложение «Чтение-запись символов», с помощью которого можно записывать в файл «Symbol.txt» символы, введенные с клавиатуры.

При разработке интерфейса приложения использовать компоненты **Label, Button, TextBox, ListBox** (один из вариантов интерфейса представлен на рис. 1).

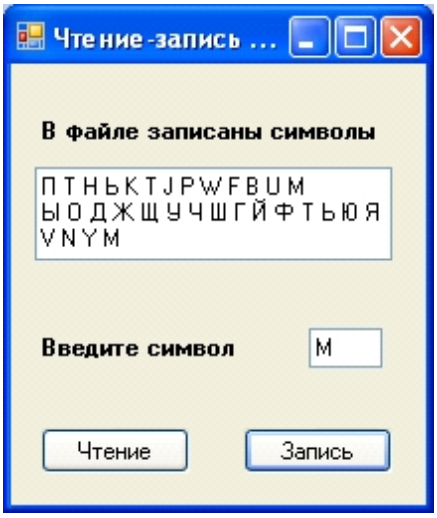

Рис. 1. Внешний вид приложения «Чтение-запись символов»

Задание 2. Разработать приложение «Поиск слов», с помощью которого можно выводить на экран из текста, хранящегося в файле «Text.txt», слова, начинающиеся с гласных букв.

При разработке интерфейса приложения использовать компоненты **Label, Button, TextBox, ListBox** (один из вариантов интерфейса представлен на рис. 2).

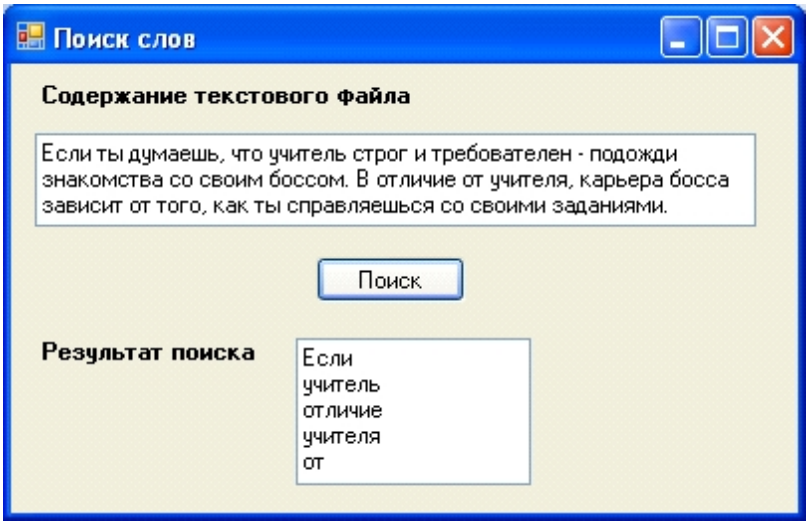

Рис. 2. Внешний вид приложения «Поиск слов»

3. Контрольные вопросы

Литература

1. Голощапов А.Л. Microsoft Visual Studio 2010. – СПб.:БХВ-Петербург, 2011. – 544 с.: ил.
2. Культин Н.Б. Microsoft Visual C# в задачах и примерах. – СПб.: БХВ-Петербург, 2009. – 320 с.: ил.
3. Лабор В.В. Си Шарп: Создание приложений для Windows. – Мн.: Харвест, 2003. – 384 с.
4. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. Пер. с англ. - М.: «Русская Редакция», 2002.- 576 с.: ил.
5. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 2. Пер. с англ. - М.: «Русская Редакция», 2002.- 624 с.: ил.
6. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0. Пер. с англ. - М.: Издательский дом "Вильямс", 2011. — 1392 с.: ил.
7. Фаронов В.В. Программирование на языке C#. – СПб.: Питер, 2007. – 240 с.: ил.
8. Фленов М.Е. Библия C#. - СПб.: БХВ-Петербург, 2011.– 560с.: ил.