Лабораторная работа №3

РАБОТА С ФАЙЛАМИ

Цель работы: изучить основные принципы и приёмы разработки приложений, использующих файлы для хранения данных.

Краткие сведения по теме

Большинство задач в программировании так или иначе связаны с работой с файлами и каталогами. Иногда требуется прочитать текст из файла или наоборот произвести запись, удалить файл или целый каталог, не говоря уже о более комплексных задачах, таких как создание текстового редактора и других подобных.

Фреймворк **.NET** предоставляет большие возможности по управлению и манипуляции файлами и каталогами, которые по большей части сосредоточены в пространстве имён **System.IO**. Классы, расположенные в этом пространстве имён (такие как **Stream**, **StreamWriter**, **FileStream** и др.), позволяют управлять файловым вводом-выводом.

Работу с файловой системой начнём с самого верхнего уровня – с дисков. Для представления диска в пространстве имён **System.IO** имеется класс **DriveInfo**.

Этот класс имеет статический метод **GetDrives**, который возвращает имена всех логических дисков компьютера. Также он предоставляет ряд полезных свойств:

- **AvailableFreeSpace** указывает на объём доступного свободного места на диске в байтах;
- **DriveFormat** получает имя файловой системы;
- **DriveType** представляет тип диска;
- **IsReady** указывает на готовность диска (например, DVD-диск может быть не вставлен в дисковод);
- **Name** получает имя диска;
- **TotalFreeSpace** получает общий объём свободного места на диске в байтах;
- **TotalSize** получает общий размер диска в байтах;
- **VolumeLabel** получает или устанавливает метку тома.

Получим имена и свойства всех дисков на компьютере:

```
using System;
using System.Collections.Generic;
using System.IO;
namespace FileApp
{
    class Program
    {
        static void Main(string[] args)
        {
            DriveInfo[] drives = DriveInfo.GetDrives();
            foreach (DriveInfo drive in drives)
            {
                Console.WriteLine("Название: {0}", drive.Name);
                Console.WriteLine("Тип: {0}", drive.DriveType);
                if (drive.IsReady)
                {
                    Console.WriteLine("Объём диска: {0}",
drive.TotalSize);
                    Console.WriteLine("Свободное пространство:
{0}", drive.TotalFreeSpace);
                    Console.WriteLine("Метка: {0}",
drive.VolumeLabel);
                }
                Console.WriteLine();
            }
            Console.ReadLine();
        }
    }
}
```

Для работы с каталогами в пространстве имён **System.IO** предназначены сразу два класса: **Directory** и **DirectoryInfo**.

Класс **Directory** предоставляет ряд статических методов для управления каталогами:

- **CreateDirectory(path)** создаёт каталог по указанному пути **path**;
- **Delete(path)** удаляет каталог по указанному пути **path**;
- **Exists(path)** определяет, существует ли каталог по указанному пути **path** и если существует, возвращает **true**, если нет **false**;
- **GetDirectories(path)** получает список каталогов в каталоге **path**;
- **GetFiles(path)** получает список файлов в каталоге **path**;
- **Move(sourceDirName, destDirName)** перемещает каталог;
- **GetParent(path)** получает родительский каталог.

Класс **DirectoryInfo** предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на **Directory**. Некоторые из его свойств и методов:

- **Create()** создаёт каталог;
- **CreateSubdirectory(path)** создаёт подкаталог по указанному пути **path**;
- **Delete()** удаляет каталог;
- **Свойство Exists** определяет, существует ли каталог;
- **GetDirectories()** получает список каталогов;
- **GetFiles()** получает список файлов;
- **MoveTo(destDirName)** перемещает каталог;
- **Parent** получает родительский каталог;
- **Root** получает корневой каталог.

Рассмотрим на примерах применение этих классов.

Получение списка файлов и подкаталогов:

```
string dirName = "C:\\";
if (Directory.Exists(dirName))
{
    Console.WriteLine("Подкаталоги:");
    string[] dirs = Directory.GetDirectories(dirName);
    foreach (string s in dirs)
    {
        Console.WriteLine(s);
    }
    Console.WriteLine();
    Console.WriteLine("Файлы:");
    string[] files = Directory.GetFiles(dirName);
    foreach (string s in files)
    {
        Console.WriteLine(s);
    }
}
```

Обратите внимание на использование слешей в именах файлов. Если используется одинарный, то перед всем путём ставим знак @: @"C:\Program Files". Если используется двойной, то тогда: "C:\\".

Создание каталога:

```
string path = @"C:\SomeDir";
string subpath = @"program\avalon";
DirectoryInfo dirInfo = new DirectoryInfo(path);
```

```
if (!dirInfo.Exists)
{
    dirInfo.Create();
}
dirInfo.CreateSubdirectory(subpath);
```

Сначала необходимо проверить, существует директория с таким именем или нет: если нет, то создать её будет нельзя и приложение выбросит ошибку. В итоге получаем следующий путь: "C:\SomeDir\program\avalon".

Получение информации о каталоге:

```
string dirName = "C:\\Program Files";
DirectoryInfo dirInfo = new DirectoryInfo(dirName);
Console.WriteLine("Название каталога: {0}", dirInfo.Name);
Console.WriteLine("Полное название каталога: {0}",
dirInfo.FullName);
Console.WriteLine("Время создания каталога: {0}",
dirInfo.CreationTime);
Console.WriteLine("Корневой каталог: {0}", dirInfo.Root);
```

Если просто применить метод **Delete** к непустой папке, в которой есть какие-нибудь файлы или подкаталоги, то приложение выбросит ошибку. Поэтому надо передать в метод **Delete** дополнительный параметр булевого типа, который укажет, что папку надо удалять со всем содержимым:

```
string dirName = @"C:\SomeFolder";
try
{
    DirectoryInfo dirInfo = new DirectoryInfo(dirName);
    dirInfo.Delete(true);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

или

```
string dirName = @"C:\SomeFolder";
Directory.Delete(dirName, true);
```

Перемещение каталога:

```
string oldPath = @"C:\SomeFolder";
string newPath = @"C:\SomeDir";
DirectoryInfo dirInfo = new DirectoryInfo(oldPath);
if (dirInfo.Exists && Directory.Exists(newPath) == false)
{
    dirInfo.MoveTo(newPath);
}
```

При перемещении каталога надо учитывать, что новый, в который необходимо перемесить всё содержимое старого, не должен существовать.

Подобно паре **Directory/DirectoryInfo** для работы с файлами предназначена пара классов **File/FileInfo**. С их помощью можно создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства класса **FileInfo**:

- **CopyTo(path)** копирует файл в новое место по указанному пути **path**;
- **Create()** создаёт файл;
- **Delete()** удаляет файл;
- **MoveTo(destFileName)** перемещает файл в новое место;
- свойство **Directory** получает родительский каталог в виде объекта **DirectoryInfo**;
- свойство **DirectoryName** получает полный путь к родительскому каталогу;
- свойство **Exists** указывает, существует ли файл;
- свойство **Length** получает размер файла;
- свойство **Extension** получает расширение файла;
- свойство **Name** получает имя файла;
- свойство **FullName** получает полное имя файла.

Класс **File** реализует похожую функциональность с помощью статических методов:

- **Copy()** копирует файл в новое место;
- **Create()** создаёт файл;
- **Delete()** удаляет файл;
- **Move** перемещает файл в новое место;
- **Exists(file)** определяет, существует ли файл.

Получение информации о файле:

```
string path = @"C:\apache\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    Console.WriteLine("Имя файла: {0}", fileInf.Name);
    Console.WriteLine("Время создания: {0}",
fileInf.CreationTime);
    Console.WriteLine("Размер: {0}", fileInf.Length);
}
```

Удаление файла:

```
string path = @"C:\apache\hta.txt";
FileInfo fileInf = new FileInfo(path);
```

```
if (fileInf.Exists)
{
    fileInf.Delete();
    // альтернатива с помощью класса File
    // File.Delete(path);
}
```

Перемещение файла:

```
string path = @"C:\apache\hta.txt";
string newPath = @"C:\SomeDir\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.MoveTo(newPath);
    // альтернатива с помощью класса File
    // File.Move(path, newPath);
}
```

Копирование файла:

```
string path = @"C:\apache\hta.txt";
string newPath = @"C:\SomeDir\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.CopyTo(newPath, true);
    // альтернатива с помощью класса File
    // File.Copy(path, newPath, true);
}
```

Метод **CopyTo** класса **FileInfo** принимает два параметра: путь, по которому файл будет копироваться, и булевое значение, которое указывает, надо ли при копировании перезаписывать файл (если **true**, то файл при копировании перезаписывается). Если в качестве последнего параметра передать значение **false**, то если такой файл уже существует, приложение выдаст ошибку.

Метод **Copy** класса **File** принимает три параметра: путь к исходному файлу и тот, по которому файл будет копироваться, и булевое значение, указывающее, будет ли файл перезаписываться.

Класс **FileStream** представляет возможности по считыванию из файла и записи в него. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

- свойство **Length** возвращает длину потока в байтах;
- свойство **Position** возвращает текущую позицию в потоке;

• метод **long Seek(long offset, SeekOrigin origin)** устанавливает позицию в потоке со смещением на количество байт, указанных в параметре **offset**;

• метод **Write** записывает в файл данные из массива байтов, принимает три параметра **Write(byte[] array, int offset, int count)**: **array** – массив байтов, куда будут помещены считываемые из файла данные; **offset** – представляет смещение в байтах в массиве **array**, в который будут помещены считанные байты; **count** – максимальное число байтов, предназначенных для чтения, но если в файле находится меньшее количество байтов, то все они будут считаны;

• метод **Read** считывает данные из файла в массив байтов и возвращает количество успешно считанных байтов, принимает три параметра **int Read(byte[] array, int offset, int count)**: **array** – массив байтов, откуда данные будут записываться в файл; **offset** – смещение в байтах в массиве **array**, откуда начинается запись байтов в поток; **count** – максимальное число байтов, предназначенных для записи.

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определённую структуру, **FileStream** может быть очень даже полезен для извлечения определённых порций информации и её обработки.

В качестве примера рассмотрим считывание-запись в текстовый файл:

```
Console.WriteLine("Введите строку для записи в файл:");
string text = Console.ReadLine();
// запись в файл
using (FileStream fstream = new
FileStream(@"C:\SomeDir\noname\note.txt", FileMode.OpenOrCreate))
{
    // преобразуем строку в байты
    byte[] array = System.Text.Encoding.Default.GetBytes(text);
    // запись массива байтов в файл
    fstream.Write(array, 0, array.Length);
    Console.WriteLine("Текст записан в файл");
}
// чтение из файла
using (FileStream fstream =
File.OpenRead(@"C:\SomeDir\noname\note.txt"))
{
    // преобразуем строку в байты
    byte[] array = new byte[fstream.Length];
```

```
    // считываем данные
    fstream.Read(array, 0, array.Length);
    // декодируем байты в строку
    string textFromFile =
System.Text.Encoding.Default.GetString(array);
    Console.WriteLine("Текст из файла: {0}", textFromFile);
}
Console.ReadLine();
```

Разберём этот пример. И при чтении, и при записи используется оператор **using**. Не надо путать данный оператор с директивой **using**, которая подключает пространства имён в начале файла кода. Оператор **using** позволяет создавать объект в блоке кода, по завершению которого вызывается метод **Dispose** у этого объекта, и, таким образом, объект уничтожается. В данном случае в качестве такого объекта служит переменная **fstream**.

Объект **fstream** создаётся двумя разными способами: через конструктор и один из статических методов класса **File**.

Здесь в конструктор передаётся два параметра: путь к файлу и перечисление **FileMode**. Данное перечисление указывает на режим доступа к файлу и может принимать следующие значения:

- **Append**, если файл существует, то текст добавляется в конец файла. Если файла нет, то он создаётся. Файл открывается только для записи.
- **Create** создаёт новый файл и если такой файл уже существует, то он перезаписывается.
- **CreateNew** создаёт новый файл и если такой файл уже существует, то в приложение выбрасывает ошибку.
- **Open** открывает файл и если файл не существует, выбрасывается исключение.
- **Create** создаёт новый файл и если такой файл уже существует, то он перезаписывается.
- **OpenOrCreate** если файл существует, он открывается, если нет – создаётся новый.
- **Truncate** если файл существует, то он перезаписывается. Файл открывается только для записи.

Статический метод **OpenRead** класса **File** открывает файл для чтения и возвращает объект **FileStream**.

Конструктор класса **FileStream** также имеет ряд перегруженных версий, позволяющий более точно настроить создаваемый объект. Все эти версии можно посмотреть на Microsoft Developer Network (MSDN).

При записи и при чтении применяется объект кодировки **Encoding.Default** из пространства имён **System.Text**. В данном случае использовали два его метода: **GetBytes** для получения массива байтов из строки и **GetString** – строки из массива байтов.

В итоге введённая строка записывается в файл **noname**. По сути это бинарный, не текстовый файл, хотя если в него записать только строку, то можно будет просмотреть его, открыв в текстовом редакторе. Однако, если записать в него случайные байты, то могут возникнуть проблемы с его пониманием:

```
fstream.WriteByte(13);
fstream.WriteByte(103);
```

Нередко бинарные файлы представляют определённую структуру, зная которую можно взять из файла нужную порцию информации или наоборот записать в конкретном месте файла определённый набор байтов. Например, в wav-файлах непосредственно звуковые данные начинаются с сорок четвёртого байта, а до него идут различные метаданные – количество каналов аудио, частота дискретизации и т.д.

С помощью метода **Seek()** мы можем управлять положением курсора потока, начиная с которого производится считывание или запись в файл. Этот метод принимает два параметра: **offset** (смещение) и позиция в файле. Позиция в файле описывается тремя значениями:

- **SeekOrigin.Begin** – начало файла;
- **SeekOrigin.End** – конец файла;
- **SeekOrigin.Current** – текущая файла.

Курсор потока, с которого начинается чтение или запись, смещается вперёд на значение **offset** относительно позиции, указанной в качестве второго параметра. Смещение может быть как отрицательным, тогда курсор сдвигается назад, так и положительным – курсор сдвигается вперёд.

Рассмотрим на примере:

```
using System.IO;
using System.Text;
class Program
{
    static void Main(string[] args)
    {
        string text = "hello world";
```

```
            // запись в файл
            using (FileStream fstream = new
           FileStream(@"D:\note.dat", FileMode.OpenOrCreate))
            {
                // преобразуем строку в байты
                byte[] input = Encoding.Default.GetBytes(text);
                // запись массива байтов в файл
                fstream.Write(input, 0, input.Length);
                Console.WriteLine("Текст записан в файл");
                // перемещаем указатель в конец файла, до конца файла
- пять байт
                fstream.Seek(-5, SeekOrigin.End); // минус 5 символов
с конца потока
                                                  // считываем четыре
символов с текущей позиции
                byte[] output = new byte[4];
                fstream.Read(output, 0, output.Length);
                // декодируем байты в строку
                string textFromFile =
Encoding.Default.GetString(output);
                Console.WriteLine("Текст из файла: {0}",
textFromFile); // worl

// заменим в файле слово world на слово house
                string replaceText = "house";
                fstream.Seek(-5, SeekOrigin.End); // минус 5 символов
с конца потока
                input = Encoding.Default.GetBytes(replaceText);
                fstream.Write(input, 0, input.Length);
                // считываем весь файл
                // возвращаем указатель в начало файла
                fstream.Seek(0, SeekOrigin.Begin);
                output = new byte[fstream.Length];
                fstream.Read(output, 0, output.Length);
                // декодируем байты в строку
                textFromFile = Encoding.Default.GetString(output);
                Console.WriteLine("Текст из файла: {0}",
textFromFile); // hello house
            }
            Console.Read();
        }
}
```

Консольный вывод:

```
Текст записан в файл
Текст из файла: world
Текст из файла: hello house
```

Вызов **fstream.Seek(-5, SeekOrigin.End)** перемещает курсор потока в конец файлов назад на пять символов, т. е. после записи в новый файл строки **"hello world"** курсор будет стоять на позиции символа **"w"**.

После этого считываем четыре байта, начиная с символа **"w"**. В данной кодировке один символ будет представлять один байт. Поэтому чтение четырёх байтов будет эквивалентно чтению четырёх символов: **"worl"**.

Затем опять перемещаемся в конец файла, не доходя до конца пять символов (т. е. опять на позицию символа **"w"**), и осуществляем запись строки **"house"**. Таким образом, строка **"house"** заменяет строку **"world"**.

В примерах, рассмотренных выше, для закрытия потока применяется конструкция **using**. После того как все операторы и выражения в блоке **using** отработают, объект **FileStream** уничтожается. Однако, возможно выбрать и другой способ:

```
FileStream fstream = null;
try
{
    fstream = new FileStream(@"D:\note3.dat",
   FileMode.OpenOrCreate);
    // операции с потоком
}
catch (Exception ex)
{
}
finally
{
    if (fstream != null)
        fstream.Close();
}
```

Если использовать конструкцию **using**, то надо явным образом вызвать метод **Close()**: **fstream.Close()**.

Класс **FileStream** не очень удобно применять для работы с текстовыми файлами. К тому же для этого в пространстве **System.IO** определены специальные классы: **StreamReader** и **StreamWriter**.

Класс **StreamReader** позволяет легко считывать весь текст или отдельные строки из текстового файла. Среди его методов можно выделить следующие:

- **Close** закрывает считываемый файл и освобождает все ресурсы;
- **Peek** возвращает следующий доступный символ, если символов больше нет, то возвращает 1;
- **ReadLine** считывает одну строку в файле;
- **ReadToEnd** считывает весь текст из файла;

• **Read** считывает и возвращает следующий символ в численном представлении и имеет перегруженную версию:

```
Read(char[] array, int index, int count)
```

где **array** – массив, куда считываются символы; **index** – индекс в массиве **array**, начиная с которого записываются считываемые символы; **count** – максимальное количество считываемых символов.

Считаем текст из файла различными способами:

```
string path = @"C:\SomeDir\hta.txt";

try
{
    Console.WriteLine("******считываем весь файл********");
    using (StreamReader sr = new StreamReader(path))
    {
        Console.WriteLine(sr.ReadToEnd());
    }
    Console.WriteLine();
    Console.WriteLine("******считываем построчно********");
    using (StreamReader sr = new StreamReader(path,
System.Text.Encoding.Default))
    {
        string line;
        while ((line = sr.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
    }
    Console.WriteLine();
    Console.WriteLine("******считываем блоками********");
    using (StreamReader sr = new StreamReader(path,
System.Text.Encoding.Default))
    {
        char[] array = new char[4];
        // считываем 4 символа
        sr.Read(array, 0, 4);
        Console.WriteLine(array);
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Как и в случае с классом **FileStream** здесь используется конструкция **using**.

В первом случае мы разом считываем весь текст с помощью метода **ReadToEnd()**.

Во втором случае считываем построчно через цикл **while**, присваивая сначала переменной **line** результат функции **sr.ReadLine()**, а затем проверяем, не равна ли она **null**:

```
while ((line = sr.ReadLine()) != null)
```

и когда объект **sr** дойдёт до конца файла и больше строк не останется, то метод **sr.ReadLine()** будет возвращать **null**.

В третьем случае считываем в массив четыре символа.

Обратите внимание, что в последних двух случаях в конструкторе **StreamReader** указывалась кодировка **System.Text.Encoding.Default**. Свойство **Default** класса **Encoding** получает кодировку для текущей кодовой страницы ANSI. Также через другие свойства можно указать другие кодировки. Если кодировка не указана, то при чтении используется UTF8. Но иногда важно указывать кодировку, так как она может отличаться от UTF8, и тогда возможно получить некорректный вывод.

Для записи в текстовый файл используется класс **StreamWriter**. Свою функциональность он реализует через следующие методы:

- **Close** – закрывает записываемый файл и освобождает все ресурсы;
- **Flush** – записывает в файл оставшиеся в буфере данные и очищает буфер;
- **Write** – записывает в файл данные простейших типов, как **int**, **double**, **char**, **string** и т.д.;
- **WriteLine** – записывает данные, только после записи добавляет в файл символ окончания строки.

Рассмотрим пример записи в текстовый файл:

```
string readPath = @"C:\SomeDir\hta.txt";
string writePath = @"C:\SomeDir\ath.txt";
string text = "";
try
{
    using (StreamReader sr = new StreamReader(readPath,
System.Text.Encoding.Default))
    {
        text = sr.ReadToEnd();
    }
```

```
    using (StreamWriter sw = new StreamWriter(writePath, false,
System.Text.Encoding.Default))
    {
        sw.WriteLine(text);
    }
    using (StreamWriter sw = new StreamWriter(writePath, true,
System.Text.Encoding.Default))
    {
        sw.WriteLine("Дозапись");
        sw.Write(4.5);
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Сначала считываем файл в переменную **text**, а затем записываем эту переменную в файл, а затем через объект **StreamWriter** записываем в новый файл, используя один из конструкторов:

```
new StreamWriter(writePath, false, System.Text.Encoding.Default).
```

Здесь первый параметр передаёт путь к записываемому файлу, второй – представляет булевую переменную, определяющую, будет файл дозаписываться или перезаписываться. Если булева переменная равна **true**, то новые данные добавляются в конец к уже имеющимся, а если **false** – файл перезаписывается. Исходя из этого, получаем, что файл перезаписывается в первом случае, а во втором – дозаписывается. Третий параметр указывает кодировку, в которой записывается файл.

Задания на лабораторную работу

Реализуйте запросы, определив:

1) фамилии студентов, у которых две и более двоек за сессию, и удалить их (выведя сообщение);

2) институт, на котором на первом курсе наибольшее количество отличников;

3) курс, на котором исключено большее количество студентов;

4) институт с наибольшим количеством отличников;

5) полный список отличников с указанием института, группы и курса, где они учатся;

6) группу, где нет двоечников;

7) институт и курс, на котором средний бал не меньше 3,5;

8) фамилии студентов, у которых нет троек и двоек;

9) институт и группу, где наибольшее количество отличников;

10) фамилии студентов-отличников на третьем курсе;

11) предметы и перечень кафедр, на которых они присутствуют;

12) фамилии студентов, группу и институт, где средний балл составляет 4,5;

13) студентов первого курса, у которых три двойки и удалите их;

14) группы, в которых нет двоечников;

15) фамилии студентов-отличников на первом и втором курсах по всем институтам, средний балл по каждой группе и упорядочьте группы по нему;

16) институты, на которых нет двоечников;

17) фамилии студентов, которые не явились хотя бы на один экзамен (оценка 0) и удалите тех, у которых средний балл ниже 3;

18) институт, на котором на первом курсе наибольшее количество групп, где нет двоек;

19) курс с наибольшим количеством отличников;

20) институт, на котором на первом курсе наибольшее количество двоечников;

21) группы, в которых нет отличников;

22) полный список двоечников с указанием института, группы и курса, где они учатся;

23) фамилии студентов-отличников на втором курсе с указанием группы и института, где они учатся.

Порядок выполнения лабораторной работы

1. Создайте классы институтов (данные о курсах и группах), предметов и студентов.

2. Реализуйте возможность ввода, удаления, редактирования данных в массив (коллекцию).

3. Реализуйте вывод результата запроса в соответствии с вариантом исполнения в файл.

4. Представьте реализацию перечисленных действий в виде меню программы с возможностью их повторов до выбора пункта «Выход из программы».