ЛАБОРАТОРНАЯ РАБОТА 14

Тема: «Разработка, отладка и испытание программ с использованием файлов»

Цель: Сформировать умение разрабатывать программы создания и обработки файлов.

Время выполнения: 4 часа.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

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

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

Работа с дисками

Работу с файловой системой начнем с самого верхнего уровня - дисков. Для представления диска в пространстве имен 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 объем доступного свободного места на диске в байтах

DriveFormat: получает имя файловой системы

DriveType: представляет тип диска

IsReady: готов ли диск (например, DVD-диск может быть не вставлен в дисковод)

Name: получает имя диска

TotalFreeSpace: получает общий объем свободного места на диске в байтах

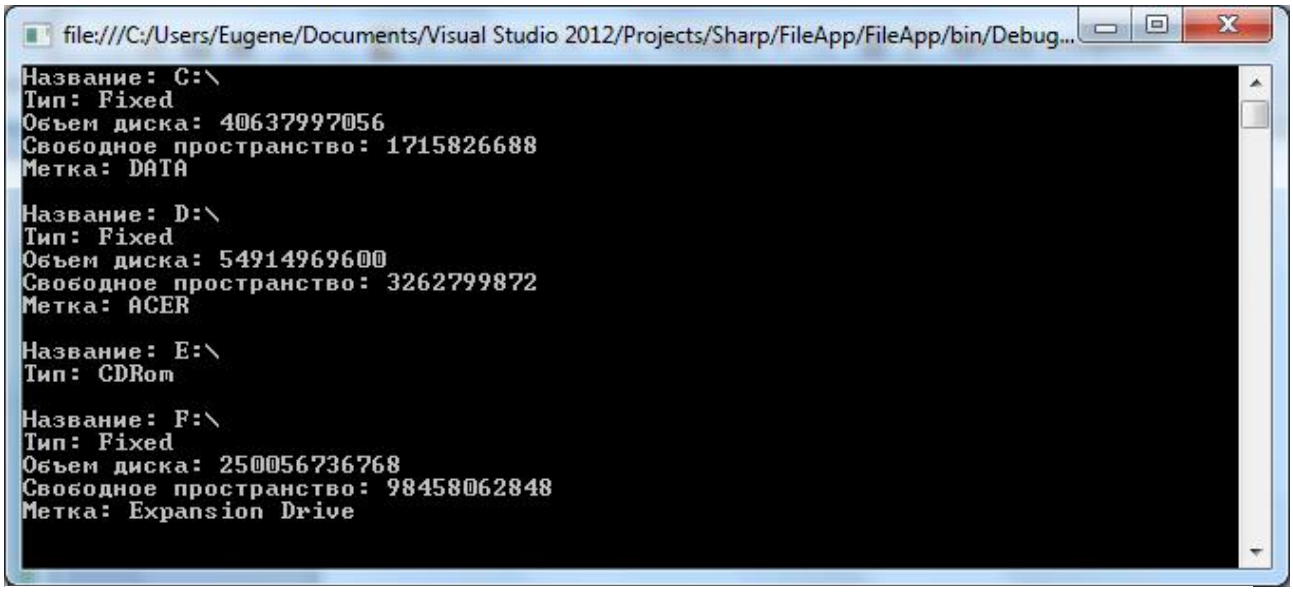
TotalSize: общий размер диска в байтах

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("Объем диска:
{0}", drive.TotalSize);
                    Console.WriteLine("Свободное
пространство: {0}", drive.TotalFreeSpace);
                    Console.WriteLine("Метка: {0}",
drive.VolumeLabel);
                }
                Console.WriteLine();
            }
            Console.ReadLine();
        }
    }
}
```



Работа с каталогами

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

Класс Directory

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

CreateDirectory(path): создает каталог по указанному пути path

Delete(path): удаляет каталог по указанному пути path

Exists(path): определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false

GetDirectories(path): получает список каталогов в каталоге path

GetFiles(path): получает список файлов в каталоге path

Move(sourceDirName, destDirName): перемещает каталог

GetParent(path): получение родительского каталога

Класс DirectoryInfo

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

Create(): создает каталог

CreateSubdirectory(path): создает 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

Свойство Parent: получение родительского каталога

Свойство Root: получение корневого каталога

Посмотрим 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. Либо мы используем двойной слеш: "C:\\", либо одинарный, но тогда перед всем путем ставим знак @: @"C:\Program Files"

Создание каталога

```
string path = @"C:\SomeDir";
string subpath = @"program\avalon";
DirectoryInfo dirInfo = new DirectoryInfo(path);
if (!dirInfo.Exists)
{
    dirInfo.Create();
}
dirInfo.CreateSubdirectory(subpath);
```

Вначале проверяем, а нету ли такой директории, так как если она существует, то ее создать будет нельзя, и приложение выбросит ошибку. В итоге у нас получится 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);
```

Удаление каталога

Если мы просто применим метод Delete к непустой папке, в которой есть какие-нибудь файлы или подкаталоги, то приложение нам выбросит ошибку. Поэтому нам 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 так:

```
1    string dirName = @"C:\SomeFolder";
2    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 надо учитывать, что новый каталог, в который мы хотим перемесить все содержимое старого каталога, не должен существовать.

Работа с файлами. Классы File и FileInfo

одобно паре Directory/DirectoryInfo для работы с файлами предназначена пара классов File и FileInfo. С их помощью мы можем 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(): создает 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(): создает 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

```
1       string path = @"C:\apache\hta.txt";
2       string newPath = @"C:\SomeDir\hta.txt";
3       FileInfo fileInf = new FileInfo(path);
4       if (fileInf.Exists)
5       {
6         fileInf.CopyTo(newPath, true);
7         // альтернатива с помощью класса File
8         // File.Copy(path, newPath, true);
9       }
```

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, как в случае выше, файл при копировании перезаписывается). Если же 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, путь, по которому файл будет копироваться, и булевое значение, указывающее, будет ли файл перезаписываться.

Чтение и запись файла. Класс FileStream

Класс FileStream представляет возможности по считыванию из файла и записи в файл. 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 Read: считывает данные из файла в массив байтов. Принимает три параметра: int Read(byte[] array, int offset, int count) и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:

array - массив байтов, куда будут помещены считываемые из файла данные

offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены

count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.

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, откуда данные будут записываться в файла

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 определенную структуру FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

Посмотрим на примере считывания-записи 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();
```

Разберем этот пример. И при чтении, и при записи используется оператор using. Не надо путать данный оператор с директивой using, которая подключает пространства имен 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 создается двумя разными способами: через конструктор и через один из статических методов класса File.

Здесь в конструктор передается 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 файл. Если файла нет, то он создается. Файл открывается только для записи.

Create: создается новый файл. Если такой файл уже существует, то он перезаписывается

CreateNew: создается новый файл. Если такой файл уже существует, то он приложение выбрасывает ошибку

Open: открывает файл. Если файл не существует, выбрасывается исключение

OpenOrCreate: если файл существует, он открывается, если нет - создается 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 msdn.

И при записи, и при чтении применяется объект кодировки Encoding.Default из пространства имен System.Text. В данном случае мы используем два его метода: GetBytes для получения массива байтов из строки и GetString для получения строки из массива байтов.

В итоге введенная нами строка записывается в файл note.txt. По сути это бинарный файл (не текстовый), хотя если мы в него запишем только строку, то сможем посмотреть в удобочитаемом виде этот файл, открыв его в текстовом редакторе. Однако если мы в него запишем случайные байты, например:

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

То у нас могут возникнуть проблемы с его пониманием. Поэтому для работы непосредственно с текстовыми файлами предназначены отдельные классы - StreamReader и StreamWriter.

Произвольный доступ к файлам

Нередко бинарные файлы представляют определенную стрктуру. И, зная эту структуру, мы можем взять из файла нужную порцию информации или наоброт записать в определенном месте файла определенный набор байтов. Например, в wav-файлах непосредственно звуковые данные начинаются с 44 байта, а до 44 байта 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 позиция в файле
```

Курсор потока, с которого начинается чтение или запись, смещается вперед на значение offset относительно позиции, указанной в качестве второго параметра. Смещение может отрицательным, тогда курсор сдвигается назад, если положительное - то вперед.

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 файл: worl
Текст из файла: hello house
```

Вызов fstream.Seek(-5, SeekOrigin.End) перемещает курсор потока в конец файлов назад на пять символов:

То есть 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 1 символ будет представлять 1 байт. Поэтому чтение 4 байтов будет эквивалентно чтению четырех сиволов: "worl".

Затем опять же перемещаемся в конец файла, не доходя до конца пять символов (то есть опять же с позиции символа "w"), и осуществляем запись строки "house". Таким образом, строка "house" заменяет строку "world".

Закрытие потока

В примерах выше дл закрытия потока применяется конструкция using. После того как все операторы и выражения в блоке using отработают, объект FileStream уничтожается. Однако мы можем 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 мы не используем конструкцию using, то нам надо явным образом вызвать метод Close(): fstream.Close()

Чтение и запись текстовых файлов. StreamReader и StreamWriter

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.

Чтение из файла и StreamReader

Класс StreamReader позволяет нам 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

Read: считывает и возвращает следующий символ в численном представлении. 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, и count - максимальное количество считываемых символов

ReadLine: считывает одну строку в файле

ReadToEnd: считывает весь текст из файла

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: while ((line = sr.ReadLine()) != null) - сначала присваиваем переменной line результат функции sr.ReadLine(), а затем проверяем, не равна ли она null. Когда объект sr дойдет 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 мы можем указать другие кодировки. Если кодировка не указана, то при чтении используется UTF8. Иногда важно указывать кодировку, так как она может отличаться от UTF8, и тогда мы получим некорректный вывод.

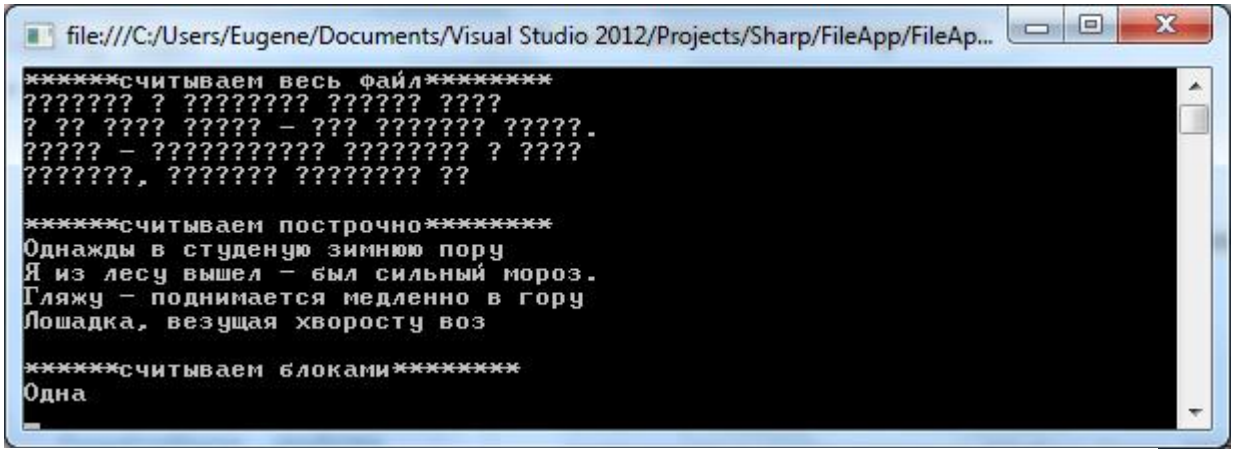


Рисунок 7

Запись в файл и StreamWriter

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: также записывает данные, только после записи добавляет в файл символ окончания строки

Рассмотрим запись в файл на примере:

```
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);
}
```

Здесь сначала мы считываем файл в переменную text, а затем записываем эту переменную в файл, а затем через объект StreamWriter записываем в новый файл.

Класс StreamWriter имеет несколько конструкторов. Здесь мы использовали один из них: new StreamWriter(writePath, false, System.Text.Encoding.Default). В качестве первого параметра передается путь к записываемому файлу. Второй параметр представляет булевую переменную, которая определяет, будет файл дозаписываться или перезаписываться. Если этот параметр равен true, то новые данные добавляются в конце к уже имеющимся данным. Если false, то файл перезаписывается. И если в первом случае файл перезаписывается, то во втором делается дозапись в конец файла.

Третий параметр указывает кодировку, в которой записывается файл.

Работа с бинарными файлами. BinaryWriter и BinaryReader

Для работы с бинарными файлами предназначена пара классов BinaryWriter и BinaryReader. Эти классы позволяют читать и записывать данные в двоичном формате.

Основные метода класса BinaryWriter

Close(): закрывает поток и освобождает ресурсы

Flush(): очищает буфер, дописывая из него оставшиеся данные в файл

Seek(): устанавливает позицию в потоке

Write(): записывает данные в поток

Основные метода класса BinaryReader

Close(): закрывает поток и освобождает ресурсы

ReadBoolean(): считывает значение bool и перемещает указатель на один байт

ReadByte(): считывает один байт и перемещает указатель на один байт

ReadChar(): считывает значение char, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке

ReadDecimal(): считывает значение decimal и перемещает указатель на 16 байт

ReadDouble(): считывает значение double и перемещает указатель на 8 байт

ReadInt16(): считывает значение short и перемещает указатель на 2 байта

ReadInt32(): считывает значение int и перемещает указатель на 4 байта

ReadInt64(): считывает значение long и перемещает указатель на 8 байт

ReadSingle(): считывает значение float и перемещает указатель на 4 байта

ReadString(): считывает значение string. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

С чтением бинарных данных все просто: соответствующий метод считывает данные определенного типа и перемещает указатель на размер этого типа в байтах, например, значение типа int занимает 4 байта, поэтому BinaryReader считает 4 байта и переместит указать на эти 4 байта.

Посмотрим на реальной задаче применение этих классов. Попробуем с их помощью записывать и считывать из файла массив структур:

```
struct State
{
    public string name;
    public string capital;
    public int area;
    public double people;

    public State(string n, string c, int a, double p)
    {
        name = n;
        capital = c;
        people = p;
        area = a;
    }
}
class Program
{
    static void Main(string[] args)
    {
        State[] states = new State[2];
        states[0] = new State("Германия",
"Берлин",  357168,  80.8);
        states[1] = new State("Франция", "Париж",
640679, 64.7);

        string path= @"C:\SomeDir\states.dat";

        try
        {
            // создаем объект BinaryWriter
```

```
            using (BinaryWriter writer = new
BinaryWriter(File.Open(path, FileMode.OpenOrCreate)))
            {
                // записываем в файл значение каждого
поля структуры
                foreach (State s in states)
                {
                    writer.Write(s.name);
                    writer.Write(s.capital);
                    writer.Write(s.area);
                    writer.Write(s.people);
                }
            }
            // создаем объект BinaryReader
            using (BinaryReader reader = new
BinaryReader(File.Open(path, FileMode.Open)))
            {
                // пока не достигнут конец файла
                // считываем каждое значение из файла
                while (reader.PeekChar() > -1)
                {
                    string name = reader.ReadString();
                    string capital =
reader.ReadString();
                    int area = reader.ReadInt32();
                    double population =
reader.ReadDouble();

                    Console.WriteLine("Страна:
{0}  столица: {1}  площадь {2} кв. км   численность
населения: {3} млн. чел.",
                        name, capital, area,
population);
                }
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        Console.ReadLine();
    }
}
```

Итак, у нас есть структура State с некоторым набором полей. В основной программе создаем массив структур и записываем с помощью

BinaryWriter. Этот класс в качестве параметра в конструкторе принимает объект Stream, который создается вызовом File.Open(path, FileMode.OpenOrCreate).

Затем в цикле пробегаемся по массиву структур и записываем каждое поле структуры в поток. В том порядке, в каком эти значения полей записываются, в том порядке они и будут размещаться в файле.

Затем считываем из записанного файла. Конструктор класса BinaryReader также в качестве параметра принимает объект потока, только в данном случае устанавливаем в качестве режима FileMode.Open: new BinaryReader(File.Open(path, FileMode.Open))

В цикле while считываем данные. Чтобы узнать окончание потока, вызываем метод PeekChar(). Этот метод считывает следующий символ и возвращает его числовое представление. Если символ отсутствует, то метод возвращает -1, что будет означать, что мы достигли конца файла.

В цикле последовательно считываем значения поле структур в том же порядке, в каком они записывались.

Таким образом, классы BinaryWriter и BinaryReader очень удобны для работы с бинарными файлами, особенно когда нам известна структура этих файлов. В то же время для хранения и считывания более комплексных объектов, например, объектов классов, лучше подходит другое решение - сериализация.

Основная функциональность по работе с JSON сосредоточена в пространстве имен **System.Text.Json**.

Ключевым типом является класс **JsonSerializer**, который и позволяет сериализовать объект в json и, наоборот, десериализовать код json в объект C#.

Для сохранения объекта в json в классе JsonSerializer определен статический метод **Serialize()**, который имеет ряд перегруженных версий. Некоторые из них:

`string Serialize(Object obj, Type type, JsonSerializerOptions options)`: сериализует объект obj типа type и возвращает код json в виде строки. Последний необязательный параметр options позволяет задать дополнительные опции сериализации

`string Serialize<T>(T obj, JsonSerializerOptions options)`: типизированная версия сериализует объект obj типа T и возвращает код json в виде строки.

`Task SerializeAsync(Object obj, Type type, JsonSerializerOptions options)`: сериализует объект obj типа type и возвращает код json в виде строки. Последний необязательный параметр options позволяет задать дополнительные опции сериализации

`Task SerializeAsync<T>(T obj, JsonSerializerOptions options)`: типизированная версия сериализует объект obj типа T и возвращает код json в виде строки.

object Deserialize(string json, Type type, JsonSerializerOptions options): десериализует строку json в объект типа type и возвращает десериализованный объект. Последний необязательный параметр options позволяет задать дополнительные опции десериализации

T Deserialize<T>(string json, JsonSerializerOptions options): десериализует строку json в объект типа T и возвращает его.

ValueTask<object> DeserializeAsync(Stream utf8Json, Type type, JsonSerializerOptions options, CancellationToken token): десериализует текст UTF-8, который представляет объект JSON, в объект типа type. Последние два параметра необязательны: options позволяет задать дополнительные опции десериализации, а token устанавливает CancellationToken для отмены задачи. Возвращается десериализованный объект, обернутый в ValueTask

ValueTask<T> DeserializeAsync<T>(Stream utf8Json, JsonSerializerOptions options, CancellationToken token): десериализует текст UTF-8, который представляет объект JSON, в объект типа T. Возвращается десериализованный объект, обернутый в ValueTask

Рассмотрим применение класса на простом примере. Сериализуем и десериализуем простейший объект:

```
using System;
using System.Text.Json;
namespace HelloApp
{
    class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Person tom = new Person { Name = "Tom", Age = 35 };
            string json = JsonSerializer.Serialize<Person>(tom);
            Console.WriteLine(json);
            Person restoredPerson = JsonSerializer.Deserialize<Person>(json);
            Console.WriteLine(restoredPerson.Name);
        }
    }
}
```

Здесь вначале сериализуем с помощью метода `JsonSerializer.Serialize()` объект типа Person в стоку с кодом json. Затем обратно получаем из этой строки объект Person посредством метода `JsonSerializer.Deserialize()`.

Консольный вывод:

```
{"Name":"Tom","Age": 35}
Tom
```

Хотя в примере выше сериализовался/десериализовался объект класса, но подобным способом мы также можем сериализовать/десериализовать структуры.

Некоторые замечания по сериализации/десериализации

Объект, который подвергается десериализации, должен иметь конструктор без параметров. Например, в примере выше этот конструктор по умолчанию. Но можно также явным образом определить подобный конструктор в классе.

Сериализации подлежат только публичные свойства объекта (с модификатором public).

Запись и чтение файла json

Поскольку методы SerializeAsyc/DeserializeAsync могут принимать поток типа Stream, то соответственно мы можем использовать файловый поток для сохранения и последующего извлечения данных:

```
using System;
using System.IO;
using System.Text.Json;
using System.Threading.Tasks;

namespace HelloApp
{
   class Person
   {
      public string Name { get; set; }
      public int Age { get; set; }
   }
   class Program
   {
      static async Task Main(string[] args)
      {
         // сохранение данных
                        using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))
         {
            Person tom = new Person() { Name = "Tom", Age = 35 };
            await JsonSerializer.SerializeAsync<Person>(fs, tom);
            Console.WriteLine("Data has been saved to file");
         }
```

```
        // чтение данных
                        using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))
        {
                                                Person restoredPerson = await JsonSerializer.DeserializeAsync<Person>(fs);
                        Console.WriteLine($"Name: {restoredPerson.Name} Age: {restoredPerson.Age}");
        }
      }
   }
}
```

В данном случае вначале данные сохраняются в файл user.json и затем считываются из него.

Индивидуальные задания для лабораторной работы 5:

Задание 1 (файлы .txt)

1. В текстовый файл построчно записаны фамилия и имя учащихся класса и его оценка за контрольную. Вывести на экран всех учащихся, чья оценка меньше 3 баллов и посчитать средний балл по классу.
2. Создать текстовый файл, записать в него построчно данные, которые вводит пользователь. Окончанием ввода пусть служит пустая строка.
3. В текстовом файле посчитать количество строк, а также для каждой отдельной строки определить количество в ней символов и слов.
4. Создать текстовый файл и записать в него n вещественных чисел.
5. В текстовом файле хранятся вещественные числа, вывести их на экран и вычислить их количество.
6. Создайте файл numbers.txt и запишите в него натуральные числа от 1 до 500 через запятую.
7. Дан массив строк: "red", "green", "black", "white", "blue". Запишите в файл элементы массива построчно (каждый элемент в новой строке).
8. В любом текстовом файле найдите размер самой длинной его строки.
9. Ввести число и сохранить его в файле s1.txt. Считать число из файла s1.txt, увеличить его на 3 и сохранить в файле s2.txt.
10. Написать программу, в которой из существующего файла считать каждый второй (чётный по порядку) символ и вывести на консоль.
11. Дан текстовый файл f, содержащий произвольный текст. Слова в тексте разделены пробелами и знаками препинания. Получить n наиболее часто встречающихся слов и число их появлений.

12. Даны текстовые файлы f1 и f2. Переписать с сохранением порядка следования компоненты файла f1 в файл f2, а компоненты файла f2 в файл f1.
13. Дан текстовый файл. Вывести количество содержащихся в нем символов и строк.
14. Считать из файла input.txt числа (числа записаны в столбик). Затем записать их произведение в файл output.txt.
15. Дана строка S и текстовый файл. Добавить строку S в конец файла.

Задание 2 (файлы .json)

Организовать чтение и запись данных в json-файл для своего варианта из задания 1.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание по вариантам, представленные в теоретических сведениях, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.