

Лекция 22

Организация файлового ввода/вывода

Потоки

Поток — это некая абстракция производства или потребления информации.

С физическим устройством поток связывает **система ввода-вывода.**

Все потоки действуют одинаково — даже если они связаны с разными физическими устройствами (дисковый файл, сетевой канал, место в памяти или любой другой объект, поддерживающий чтение и запись в линейном режиме).

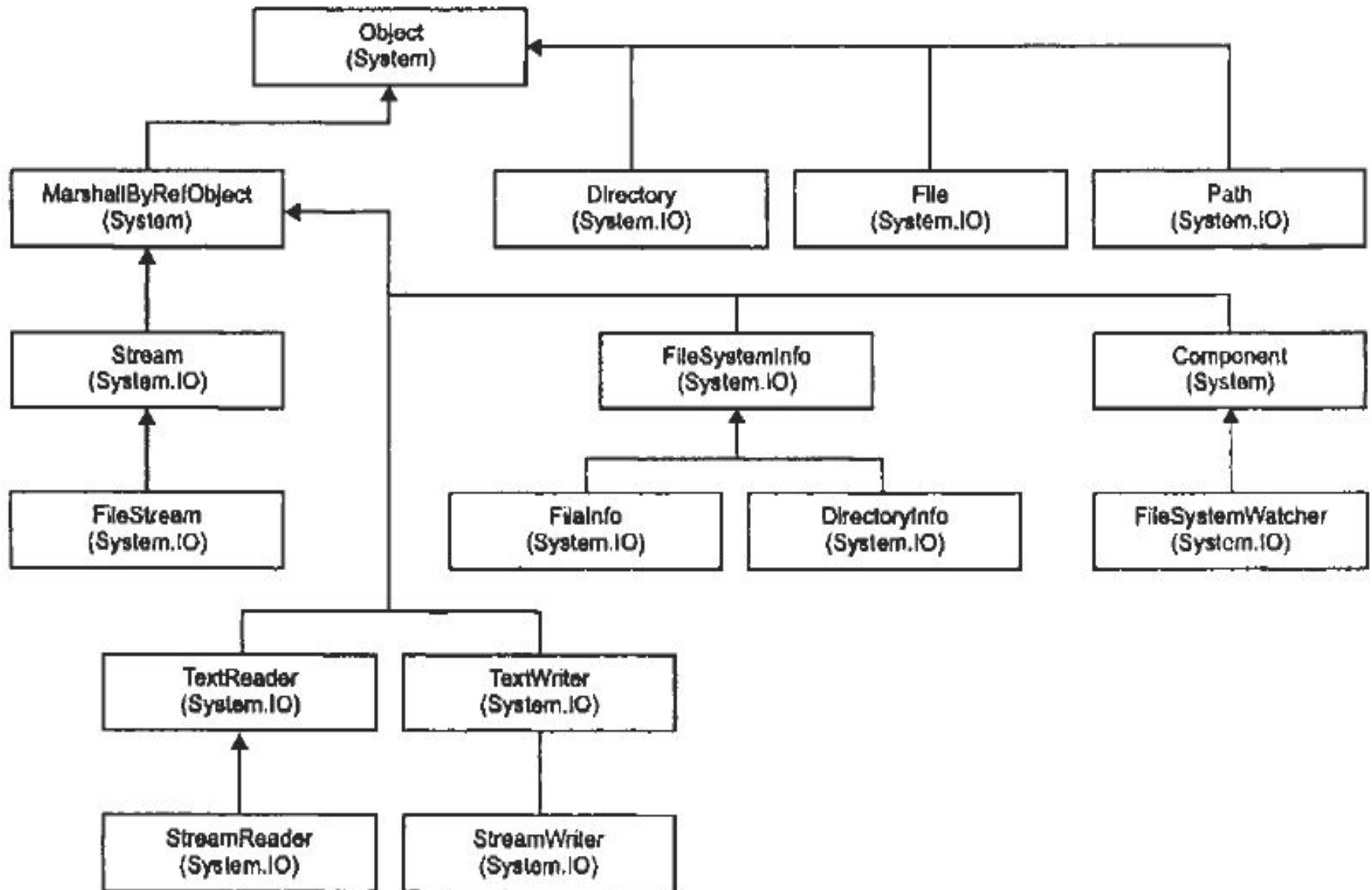
Существуют два типа потоков:

- **Выходные**
- **Входные**

Байтовые и символьные потоки

В среде .NET Framework определены классы как для **байтовых**, так и для **символьных потоков**. Но на самом деле **классы символьных потоков служат лишь оболочками для превращения заключенного в них байтового потока в символьный**, автоматически выполняя любые требующиеся преобразования типов данных. Следовательно, **символьные потоки основываются на байтовых, хотя они и разделены логически**. Основные классы потоков определены в пространстве имен **System.IO**.

Некоторые классы System.IO



Некоторые классы System.IO

Класс	Описание
File	Статический служебный класс, предоставляющий множество статических методов для перемещения, копирования и удаления файлов
Directory	Статический служебный класс, предоставляющий множество статических методов для перемещения, копирования и удаления каталогов
Path	Служебный класс, используемый для манипулирования путевыми именами
FileInfo	Представляет физический файл на диске, имеет методы для манипулирования этим файлом. Для любого, кто читает или пишет в этот файл, должен быть создан объект stream
DirectoryInfo	Представляет физический каталог на диске и имеет методы для манипулирования этим каталогом

Некоторые классы System.IO

Класс	Описание
FileSystemInfo	Служит базовым классом для FileInfo и DirectoryInfo, обеспечивая возможность работы с файлами и каталогами одновременно, используя полиморфизм
Stream	Представляет байтовый поток и является базовым для всех остальных классов потоков. Абстрактный класс, поддерживающий чтение и запись байтов
FileStream	Представляет файл, который может быть записан, прочитан или то и другое. Этот файл может быть записан или прочитан как синхронно, так и асинхронно

Некоторые классы System.IO

Класс	Описание
StreamReader	Читает символьные данные из потока и может быть создан с использованием класса FileStream в качестве базового
StreamWriter	Пишет символьные данные в поток и может быть создан с использованием класса FileStream в качестве базового
FileSystemWatcher	Наиболее развитый класс, который используется для мониторинга файлов и каталогов и представляет события, которые приложение может перехватить, когда в этих объектах происходят какие-то изменения. Этой функциональности всегда недоставало в программировании для Windows, но теперь .NET Framework значительно облегчает задачу реагирования на события файловой системы

Класс Stream

Класс	Описание
Stream	
BinaryReader/ BinaryWriter	Чтение и запись в поток закодированных строк и базовых типов данных
FileSystemInfo	
File, FileInfo, Directory, DirectoryInfo	Предоставляют реализацию абстрактных классов FileSystemInfo, в том числе создание, перемещение, переименование и удаление файлов и каталогов
FileStream	
TextReader, TextWriter, StringReader, StringWriter	Предназначен для чтения/записи объектов класса File; поддерживает произвольный доступ к файлам. По умолчанию открывает файлы синхронно, поддерживает асинхронный доступ к файлам. TextReader и TextWriter являются абстрактными классами, предназначенными для ввода/вывода символов Unicode. Классы StringReader и StringWriter выполняют чтение/запись в строки, что позволяет при вводе/выводе пользоваться либо потоком данных, либо строкой

Класс Stream

Класс	Описание
BufferedStream	Поток данных, «добавляющий» буферизацию к другому потоку, например NetworkStream. Обратите внимание, что класс File Stream имеет встроенную буферизацию. Классы BufferedStream могут повысить производительность потоков, к которым они прикреплены
MemoryStream	Небуферизованный поток, инкапсулированные данные которого непосредственно доступны в памяти. Класс MemoryStream не имеет внешней памяти и наиболее полезен в качестве временного буфера
NetworkStream	Поток данных в сетевом соединении

Класс Stream

Метод	Описание
void Close ()	Закрывает поток
void Flush ()	Выводит содержимое потока на физическое устройство
int ReadByte ()	Возвращает целочисленное представление следующего байта, доступного для ввода из потока. При обнаружении конца файла возвращает значение -1
int Read (byte [] buffer, int offset, int count)	Делает попытку ввести count байтов в массив buffer, начиная с элемента buffer [offset]. Возвращает количество успешно введенных байтов
long Seek (long offset, SeekOrigin origin)	Устанавливает текущее положение в потоке по указанному смещению offset относительно заданного начала отсчета origin. Возвращает новое положение в потоке
void WriteByte (byte value)	Выводит один байт в поток вывода
void Write (byte [] buffer, int offset, int count)	Выводит подмножество count байтов из массива buffer, начиная с элемента buffer[offset]. Возвращает количество выведенных байтов

Класс Stream

Свойство	Описание
bool CanRead	Принимает значение true, если из потока можно ввести данные. Доступно только для чтения
bool CanSeek	Принимает значение true, если поток поддерживает запрос текущего положения в потоке. Доступно только для чтения
bool CanWrite	Принимает значение true, если в поток можно вывести данные. Доступно только для чтения
long Length	Содержит длину потока в байтах. Доступно только для чтения
long Position	Представляет текущее положение в потоке. Доступно как для чтения, так и для записи
int ReadTimeout	Представляет продолжительность времени ожидания в операциях ввода. Доступно как для чтения, так и для записи
int WriteTimeout	Представляет продолжительность времени ожидания в операциях вывода. Доступно как для чтения, так и для записи

Порядок работы с файлом

1. Подключить пространство имен, в котором описываются стандартные классы для работы с файлами.
2. Объявить *файловую переменную* и связать ее с файлом на диске.
3. Выполнить операции ввода-вывода.
4. Закрывать файл.

Класс FileStream

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы.

Следовательно, для работы с текстовыми файлами применяются другие классы.

Однако, некоторые операции, например, произвольный доступ к файлу, могут выполняться только посредством объекта **FileStream**.

Класс FileStream

Для формирования байтового потока, привязанного к файлу, создается объект класса **FileStream**, например, с помощью конструктора:

FileStream(string **путь**, FileMode **режим**, FileAccess **доступ**);

где **путь** обозначает имя открываемого файла, включая полный путь к нему;

доступ — конкретный способ доступа к файлу:

Read - для чтения

Write - для записи

ReadWrite - для чтения или записи (по умолчанию)

режим — порядок открытия файла;

Класс FileStream

Член FileMode (режим)	Поведение при существующем файле	Поведение при отсутствии файла
Append	Файл открыт, поток установлен в конец файла - текст добавляется в конец файла. Может использоваться только в сочетании с FileAccess.Write - файл открывается только для записи	Создается новый файл. Может использоваться только в сочетании с FileAccess.Write
Create	Файл уничтожается и на его месте создается новый - файл перезаписывается	Создается новый файл
CreateNew	Генерируется исключение	Создается новый файл

Класс FileStream

Член FileMode	Поведение при существующем файле	Поведение при отсутствии файла
Open	Файл открывается, поток позиционируется в начало файла - текст добавляется в начало файла	Генерируется исключение
OpenOrCreate	файл открывается, поток позиционируется в начало файла	Создается новый файл
Truncate	файл открывается и очищается - перезаписывается. Поток позиционируется в начало файла. Файл открывается только для записи. Исходная дата создания файла остается неизменной	Генерируется исключение

Класс `FileStream`

Например, в следующем примере кода файл **test.dat** открывается ТОЛЬКО для ЧТЕНИЯ:

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

По завершении работы с файлом его следует закрыть, вызвав метод **Close()**.

Класс FileStream

Классы **File** и **FileInfo** предоставляют методы **OpenRead()** и **OpenWrite()**, облегчающие создание объектов **FileStream**, например:

```
FileStream aFile = File.OpenRead("Data.txt");
```

Следующий код выполняет ту же функцию:

```
FileInfo aFileInfo = new FileInfo("Data.txt");  
FileStream aFile = aFileInfo.OpenRead();
```

Класс `FileStream`

В классе `FileStream` определены два метода для чтения байтов из файла: `ReadByte()` и `Read()`.

```
int ReadByte();
```

```
int Read(byte[ ] array, int offset, int count);
```

Метод `Read()` считывает количество `count` байтов в массив `array`, начиная с элемента `array[offset]` - смещение в байтах в массиве `array`].

Метод возвращает количество байтов, успешно считанных из файла.

Пример 1

```
using System;  
using System.IO; // 1  
class ShowFile {  
    static void Main(string[] args) {  
        int i;  
        FileStream fin = null; // 2  
        try {  
            fin = new FileStream("filebyte",  
FileMode.Open); // 3
```

Пример 1

```
do { // читать байты до конца файла
    i = fin.ReadByte(); // 4
    if (i != -1) {
        Console.Write("код = " + i + " символ = ");
        Console.WriteLine((char)i);
    }
} while (i != -1);
} catch(IOException exc) {
    Console.WriteLine("Ошибка ввода-
вывода:\n" + exc.Message);
    return;
}
```

Пример 1

```
catch(Exception exc) {  
    Console.WriteLine(exc.Message);  
    // обработать ошибку, если это возможно  
    // еще раз сгенерировать необрабатываемые ИС  
}  
    finally {  
        if (fin != null) fin.Close();           // 5  
    }  
}  
}
```

Порядок работы с файлом

1. Подключить пространство имен, в котором описываются стандартные классы для работы с файлами (оператор 1).
2. Объявить *файловую переменную* и связать ее с файлом на диске (операторы 2 и 3).
3. Выполнить операции ввода-вывода (оператор 4).
4. Закрыть файл (оператор 5).

Класс `FileStream`

Для записи байта в файл служит метод `WriteByte()`:

```
void WriteByte(byte value);
```

Этот метод выполняет запись в файл байта, обозначаемого параметром `value`.

Для записи в файл целого массива байтов может

быть вызван метод `Write()`:

```
void Write(byte[] array, int offset, int count);
```

Метод `Write()` записывает в файл количество

`count` байтов из массива `array`, начиная с

элемента `array [offset` - смещение в байтах в

массиве `array]`. Метод возвращает количество

байтов, успешно записанных в файл.

Пример 2

```
using System;
using System.IO;
class CopyFile {
    static void Main(string[] args) {
        int i;        string si, sr;
        FileStream fin = null;
        FileStream fout = null;
        Console.Write("Введите имя исходного файла,
который следует копировать:  ");
        si = Console.ReadLine();
        Console.Write("Введите имя выходного файла:  ");
        sr = Console.ReadLine();
```

Пример 2

```
try { // открыть файлы
    fin = new FileStream(si , FileMode.Open);
    fout = new FileStream(sr, FileMode.Create);
    do { // скопировать файл
        i = fin.ReadByte();
        if (i != -1) fout.WriteByte((byte)i);
    } while (i != -1);
} catch(IOException exc) {
    Console.WriteLine("Ошибка ввода-вывода:\n"+exc.Message);
} finally {
    if (fin != null) fin.Close();
    if (fout != null) fout.Close ();
}
}
```

Пример 3

```
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)) {
```

```
// преобразование строки в байты (Default – UTF8)
```

```
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");
```

```
}
```

Пример 3

// чтение из файла

```
using (FileStream fstream =  
    File.OpenRead(@"C:\SomeDir\noname\note.txt")) {  
    // преобразование строки в байты  
    byte[] array = new byte[fstream.Length];  
    // считывание данных  
    fstream.Read(array, 0, array.Length);  
    // декодирование байтов в строку  
    string textFromFile =  
        System.Text.Encoding.Default.GetString(array);  
    Console.WriteLine("Текст из файла: {0}",  
        textFromFile);  
}  
Console.ReadLine();
```

Файлы с произвольным доступом

метод **Seek()** позволяет установить указатель файла на любое место в файле:

```
long Seek(long offset, SeekOrigin origin);
```

Курсор потока, с которого начинается чтение или

запись, смещается вперед на значение **offset**

относительно позиции, указанной в качестве второго

параметра **origin**. Смещение может отрицательным,

тогда курсор сдвигается назад, если положительное -

то вперед.

В качестве **origin** может быть указано одно из

приведенных ниже значений:

SeekOrigin.Begin

Поиск от начала файла

SeekOrigin.Current

Поиск от текущего положения

SeekOrigin.End

Поиск от конца файла

Пример 4

```
using System;
using System.IO;
class RandomAccessDemo {
    static void Main() {
        FileStream f = null;
        char ch;
        try {
            f = new FileStream("random.dat", FileMode.Create);
            // записать английский алфавит в файл
            for (int i=0; i < 26; i++) f.WriteByte((byte)('A'+i));
            // считать отдельные буквы английского алфавита
            f.Seek(0, SeekOrigin.Begin); // найти первый байт
            ch = (char) f.ReadByte ();
            Console.WriteLine("Первая буква: " + ch);
        }
    }
}
```

Пример 4

```
f.Seek(1, SeekOrigin.Begin); // найти второй байт
ch = (char) f.ReadByte ();
Console.WriteLine("Вторая буква: " + ch);
f.Seek(4, SeekOrigin.Begin); // найти пятый байт
ch = (char) f.ReadByte();
Console.WriteLine("Пятая буква: " + ch);
// прочитать буквы английского алфавита через одну
Console.WriteLine("Буквы алфавита через одну: ");
for (int i=0; i < 26; i += 2) {
    f.Seek(i, SeekOrigin.Begin); // найти i-й символ
    ch = (char) f.ReadByte();
    Console.Write(ch + " ");
}
}
```

Пример 4

```
catch(IOException exc) {  
    Console.WriteLine("Ошибка ввода-вывода\n" +  
exc.Message);  
}  
    finally {  
        if (f != null) f.Close();  
        Console.WriteLine();  
    }  
}  
}
```


Пример 4

```
// прочитать буквы английского алфавита через одну
Console.WriteLine("Буквы алфавита через одну: ");
for (int i=0; i < 26; i += 2) {
    f.Position = i; // найти i-й символ
                    // посредством свойства Position
    ch = (char) f.ReadByte ();
    Console.Write(ch + " ");
}
```

Пример 5

```
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)) {
        // преобразование строки в байты
        byte[] input = Encoding.Default.GetBytes(text);
```

Пример 5

```
// запись массива байтов в файл
fstream.Write(input, 0, input.Length);
Console.WriteLine("Текст записан в файл");
// перемещение указателя в конец файла,
// до конца файла- пять байт
fstream.Seek(-5, SeekOrigin.End);
    // минус 5 символов с конца потока
// считывание четырех символов с текущей позиции
byte[] output = new byte[4];
fstream.Read(output, 0, output.Length);
// декодирование байтов в строку
string textFromFile =
Encoding.Default.GetString(output);
```

Пример 5

```
Encoding.Default.GetString(output);  
    Console.WriteLine("Текст из файла: {0}",  
textFromFile); // worl  
// замена в файле слова 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);  
// СЧИТЫВАНИЕ ВСЕГО ФАЙЛА,  
// ВОЗВРАЩЕНИЕ УКАЗАТЕЛЯ В НАЧАЛО ФАЙЛА  
    fstream.Seek(0, SeekOrigin.Begin);
```

Пример 5

```
output = new byte[fstream.Length];
fstream.Read(output, 0, output.Length);
// декодирование байтов в строку
textFromFile =
Encoding.Default.GetString(output);
    Console.WriteLine("Текст из файла: {0}",
textFromFile); // hello house
    }
    Console.Read();
}
}
```

Пример 5

```
fstream.Seek(-5, SeekOrigin.End)
```



Пример 6

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

```
using System.Text;
```

```
using System.IO;
```

```
...
```

```
static void Main(string[] args) {
```

```
    byte[] byData = new byte[200];
```

```
    char[] charData = new Char [200];
```

```
    try {
```

```
        FileStream aFile = new
```

```
        FileStream("../..//Program.cs", FileMode.Open)
```

```
        aFile.Seek(113, SeekOrigin.Begin);
```

```
        aFile.Read(byData, 0, 200);
```

```
    }
```

Пример 6

```
catch(IOException e) {  
    Console.WriteLine("Сгенерировано  
ИСКЛЮЧЕНИЕ ВВОДА-ВЫВОДА!");  
    Console.WriteLine(e.ToString ());  
    Console.ReadKey();  
    return;  
}  
Decoder d = Encoding.UTF8.GetDecoder();  
d.GetChars(byData, 0, byData.Length, charData, 0);  
Console.WriteLine(charData);  
Console.ReadKey();  
}
```


Пример 7

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

```
using System.Text;
```

```
using System. IO;
```

```
...
```

```
static void Main(string[] args) {
```

```
    byte[] byData;
```

```
    char[] charData;
```

```
    try {
```

```
        FileStream aFile = new FileStream("Temp.txt",  
        FileMode.Create);
```

```
        charData="Скоро НОВЫЙ  
год!!!".ToCharArray();
```

```
        byData = new byte[charData.Length*2];
```

Пример 7

```
e.GetBytes(charData, 0, charData.Length,
byData, 0, true);
// переместить файловый указатель в начало файла
aFile.Seek(0, SeekOrigin.Begin);
aFile.Write(byData, 0, byData.Length);
} catch (IOException ex) {
    Console.WriteLine("Сгенерировано исключение
ВВОДА-ВЫВОДА!");
    Console.WriteLine (ex.ToString());
    Console.ReadKey();
    return;
}
}
```

Символьный ввод-вывод в файл

Несмотря на то что файлы часто обрабатываются побайтово, для этой цели можно воспользоваться также символьными потоками.

Преимущество символьных потоков заключается в том, что они оперируют символами непосредственно в уникоде.

На вершине иерархии классов символьных потоков находятся абстрактные классы **TextReader** (организует ввод) и **TextWriter** (организует вывод).

Класс TextReader

Метод	Описание
int Peek()	Получает следующий символ из потока ввода, но не удаляет его. Возвращает значение -1, если ни один из символов не доступен
int Read()	Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца потока возвращает значение -1
int Read(char[]buffer, int index, int count)	Делает попытку ввести количество count символов в массив buffer, начиная с элемента buffer [index], и возвращает количество успешно введенных символов

Класс TextReader

Метод	Описание
int ReadBlock (char[]buffer, int index, int count)	Делает попытку ввести количество count символов в массив buffer, начиная с элемента buffer [index], и возвращает количество успешно введенных символов
string ReadLine ()	Вводит следующую текстовую строку и возвращает ее в виде объекта типа string. При попытке прочесть признак конца файла возвращает пустое значение
string ReadToEnd ()	Вводит все символы, оставшиеся в потоке, и возвращает их в виде объекта типа string
void Close ()	Закрывает считывающий поток и освобождает его ресурсы

Класс `TextWriter`

Для записи в текстовый файл используется класс `TextWriter`. 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.

СИМВОЛЬНЫЙ ВВОД-ВЫВОД В ФАЙЛ

Классы **TextReader** и **TextWriter** реализуются несколькими классами символьных потоков:

StreamReader Предназначен для ввода символов из байтового потока. Этот класс является оболочкой для байтового потока ввода

StreamWriter Предназначен для вывода символов в байтовый поток. Этот класс является оболочкой для байтового потока вывода

StringReader Предназначен для ввода символов из символьной строки

StringWriter Предназначен для вывода символов в символьную строку

СИМВОЛЬНЫЙ ВВОД-ВЫВОД В ФАЙЛ

Для выполнения операций символьного ввода-вывода в файлы объект класса **FileStream** заключается в оболочку класса **StreamReader** или **StreamWriter**. В этих классах выполняется автоматическое преобразование байтового потока в символьный и наоборот.

Класс **StreamWriter** является производным от класса **TextWriter**, а класс **StreamReader** — производным от класса **TextReader**.

Следовательно, в классах **StreamReader** и **StreamWriter** доступны методы и свойства, определенные в их базовых классах.

Применение класса `StreamWriter`

Существует много способов создания объекта `StreamWriter`.

Если объект `FileStream` уже имеется, можно использовать следующее для создания `StreamWriter` (**1 способ**):

```
FileStream aFile = new FileStream("Log.txt",  
    FileMode.CreateNew);  
StreamWriter sw = new StreamWriter(aFile);
```

Объект `StreamWriter` можно также создать непосредственно из файла (**2 способ**):

```
StreamWriter sw = new StreamWriter("Log.txt",  
    true);
```

Пример 8 (1 способ)

```
using System;
using System.IO;
class KtoD {
    static void Main() {
        string str;
        FileStream fout; // 1
        // открыть сначала поток файлового ввода-вывода
        try {
            fout = new FileStream("test.txt", FileMode.Create); // 2
        } catch(IOException exc) {
            Console.WriteLine("Ошибка открытия файла:\n" +
                exc.Message);
            return;
        }
    }
}
```

Пример 8 (1 способ)

// заключить поток файлового ввода-вывода в

// оболочку класса StreamWriter

```
StreamWriter fstr_out = new StreamWriter(fout); // 3
```

```
try {
```

```
    Console.WriteLine("Введите текст, а по окончании - 'стоп'.");
```

```
    do {
```

```
        Console.Write(": ");
```

```
        str = Console.ReadLine();
```

```
        if (str != "стоп") {
```

```
            str = str + "\r\n"; // добавить новую строку
```

```
            fstr_out.Write(str);
```

```
        }
```

```
    } while (str != "стоп");
```

```
}
```

Пример 8 (1 способ)

```
catch(IOException exc) {  
    Console.WriteLine("Ошибка ввода-  
вывода:\n" + exc.Message);  
} finally {  
    fstr_out.Close();  
}  
}  
}
```

Пример 9 (2 способ)

```
using System;
using System.IO;
class KtoD {
    static void Main() {
        string str;
        StreamWriter fstr_out = null; // 1
        try {
            // открыть файл, заключенный в оболочку
            // класса StreamWriter
            fstr_out = new StreamWriter("test.txt"); // 2
            Console.WriteLine("Введите текст, а по
окончании - 'стоп.'");
```

Пример 9 (2 способ)

```
do {
    Console.Write(": ");
    str = Console.ReadLine();
    if (str != "стоп") {
        str = str + "\r\n"; // добавить новую строку
        fstr_out.Write(str);
    }
} while(str != "стоп");
} catch(IOException exc) {
    Console.WriteLine("Ошибка ввода-вывода:\n" +
exc.Message);
} finally { if (fstr_out != null) fstr_out.Close(); }
}
```

Применение класса `StreamReader`

Чтобы создать экземпляр класса `StreamReader`, можно сначала создать объект `FileInfo` и затем вызвать его метод `OpenText()` (**0 способ**):

```
FileInfo theSourceFile = new FileInfo (@"C:\test\test1.cs");  
StreamReader stream = theSourceFile.OpenText();
```

Наиболее распространенный способ его создания — применение ранее созданного объекта `FileStream` (**1 способ**):

```
FileStream aFile = new FileStream("Log.txt", FileMode.Open);  
StreamReader sr = new StreamReader(aFile);
```

Подобно `StreamWriter`, объект класса `StreamReader` может быть создан непосредственно из строки, содержащей путь к определенному файлу (**2 способ**):

```
StreamReader sr = new StreamReader("Log.txt");
```

Пример 10

```
using System;
using System.IO;
class DtoS {
    static void Main() {
        StreamReader fstr_in;
        string s;
        try {
            // открыть файл, заключенный в оболочку класса StreamReader
            fstr_in = new StreamReader("test.txt");
        } catch (IOException exc) {
            Console.WriteLine("Ошибка открытия файла:\n" +
                exc.Message);
            return;
        }
    }
}
```


Пример 10

```
try {  
    while ((s = fstr_in.ReadLine()) != null) {  
        Console.WriteLine(s);  
    }  
} catch (IOException exc) {  
    Console.WriteLine("Ошибка ввода-вывода:\n" +  
exc.Message);  
} finally {  
    fstr_in.Close();  
}  
}  
}
```

```
while(!fstr_in.EndOfStream) {  
    s = fstr_in.ReadLine();  
    Console.WriteLine(s);  
}
```

Способы считывания текста из файла

1. Построчный ввод можно организовать с помощью оператора **using**.

```
Console.WriteLine("*****считывается файл  
построчно + using*****");
```

```
    string s;
```

```
    using (StreamReader sr = new  
StreamReader("test.txt",  
System.Text.Encoding.Default)) {  
        while ((s = fstr_in.ReadLine()) != null) {  
            Console.WriteLine(s);  
        }  
    }  
}
```

Способы считывания текста из файла

```
2. Console.WriteLine("*****считывается файл  
    ПОСИМВОЛЬНО*****");  
StreamReader fstr_in = new StreamReader(fin);  
int nChar;  
try { nChar = fstr_in.Read();  
    while(nChar != -1) {  
        Console.Write(Convert.ToChar(nChar));  
        nChar = fstr_in.Read();  
    }  
} catch (IOException exc) {  
    Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);  
} finally {  
    fstr_in.Close();  
}
```

Способы считывания текста из файла

3. СИМВОЛЫ МОЖНО СЧИТЫВАТЬ БЛОКАМИ.

```
Console.WriteLine("*****считывается файл  
    блоками*****");
```

```
using (StreamReader fstr_in = new  
    StreamReader("test.txt",  
    System.Text.Encoding.Default)) {  
    char[] array = new char[4];  
    // считывается 4 символа  
    fstr_in.Read(array, 0, 4);  
    Console.WriteLine(array);  
}
```

Способы считывания текста из файла

4. Очень удобен при работе с **небольшими** файлами метод **ReadToEnd()**. Он читает весь файл целиком и возвращает его содержимое в виде строки.

```
Console.WriteLine("*****считывается весь  
файл*****");  
using (StreamReader fstr_in = new  
    StreamReader(fin)) {  
    Console.WriteLine(fstr_in.ReadToEnd());  
}
```

Хотя это может показаться простым и удобным, следует соблюдать осторожность, читая все данные в строковый объект и помещая их все в память. В зависимости от размера этих данных это может быть крайне нежелательно.

Способы считывания текста из файла

5. С применением функций преобразования типов

```
StreamReader f = new StreamReader(
```

```
    "d:\\C#\\input.txt" );
```

```
string s = f.ReadLine();
```

```
Console.WriteLine( "s = " + s );
```

```
char c = (char)f.Read();
```

```
f.ReadLine();
```

```
Console.WriteLine( "c = " + c );
```

```
string buf;
```

```
buf = f.ReadLine();
```

```
int i = Convert.ToInt32( buf );
```

```
Console.WriteLine( i );
```

Способы считывания текста из файла

5. С применением функций преобразования типов

```
buf = f.ReadLine();
```

```
double x = Convert.ToDouble( buf );
```

```
Console.WriteLine( x );
```

```
buf = f.ReadLine();
```

```
double y = double.Parse( buf );
```

```
Console.WriteLine( y );
```

```
buf = f.ReadLine();
```

```
decimal z = decimal.Parse( buf );
```

```
Console.WriteLine( z );
```

```
f.Close();
```

Применение класса `MemoryStream`

Иногда оказывается полезно читать вводимые данные из массива или записывать выводимые данные в массив, а не вводить их непосредственно из устройства или выводить прямо на него. Для этой цели служит класс `MemoryStream`. Один из конструкторов класса: `MemoryStream(byte[] buffer);` где **buffer** обозначает массив байтов, используемый в качестве источника или адресата в запросах ввода-вывода. Массив **buffer** должен быть достаточно большим для хранения направляемых в него данных.

Пример 11

```
using System;
using System.IO;
class MemStrDemo {
    static void Main() {
        byte[] storage = new byte[255];
        // создать запоминающий поток
        MemoryStream memstrm = new
        MemoryStream(storage);
        // заключить объект memstrm в оболочки классов
        // чтения и записи данных в потоки
        StreamWriter memwtr = new StreamWriter(memstrm);
        StreamReader memrdr = new StreamReader(memstrm);
```

Пример 11

```
try {  
    // записать данные в память, используя объект memwtr  
    for (int i=0; i < 10; i++)  
        memwtr.WriteLine("byte [" + i + "]:"+ i);  
    // поставить в конце точку  
    memwtr.WriteLine(".");  
    memwtr.Flush();  
    Console.WriteLine("Чтение прямо из массива storage: ");  
    // отобразить содержимое массива storage непосредственно  
    foreach (char ch in storage) {  
        if (ch == '.') break;  
        Console.Write (ch);  
    }  
    Console.WriteLine("\nЧтение из потока с помощью  
объекта memrdr: ");
```

Пример 11

```
memstrm.Seek(0, SeekOrigin.Begin);
```

```
string str = memrdr.ReadLine();
```

```
while(str != null) {
```

```
    str = memrdr.ReadLine();
```

```
    if (str [0] == '.') break;
```

```
    Console.WriteLine (str);
```

```
}
```

```
} catch(IOException exc) {
```

```
    Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
```

```
} finally {
```

```
// освободить ресурсы считывающего и записывающего потоков
```

```
    memwtr.Close();    memrdr.Close();
```

```
}
```

```
}
```

```
}
```

Применение классов **StringReader** и **StringWriter**

Для выполнения операций ввода-вывода с запоминанием в некоторых приложениях в качестве базовой памяти иногда лучше использовать массив типа **string**, чем массив типа **byte**. Именно для таких случаев и предусмотрены классы **StringReader** и **StringWriter**. В частности, класс **StringReader** наследует от класса **TextReader**, а класс **StringWriter** — от класса **TextWriter**.

Пример 12

```
using System;
using System.IO;
class StrRdrWtrDemo {
    static void Main() {
        StringWriter strwtr = null;
        StringReader str rdr = null;
        try {
            // создать объект класса StringWriter
            strwtr = new StringWriter();
            // вывести данные в записывающий поток типа StringWriter
            for (int i=0; i < 10; i++)
                strwtr.WriteLine("Значение i равно: " + i);
            // создать объект класса StringReader
            str rdr = new StringReader(strwtr.ToString());
        }
    }
}
```

Пример 12

```
// ввести данные из считывающего потока типа StringReader
string str = strdr.ReadLine();
while (str != null) {
    str = strdr.ReadLine();
    Console.WriteLine(str);
}
} catch (IOException exc) {
    Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
} finally {
// освободить ресурсы считывающего и записывающего потоков
if (strdr != null) strdr.Close();
if (strwtr != null) strwtr.Close();
}
}
}
```

Переадресация стандартных потоков

Для всех программ, в которых используется пространство имен **System**, доступны встроенные потоки, открывающиеся с помощью свойств **Console.In** - связано с потоком ввода, **Console.Out** - связано с потоком вывода и **Console.Error** - связано со стандартным потоком сообщений об ошибках, которые по умолчанию также выводятся на консоль.

Но эти потоки могут быть переадресованы на любое другое совместимое устройство ввода-вывода. И чаще всего они переадресовываются в файл.

Переадресацию стандартных потоков можно осуществлять под управлением самой программы с помощью методов **SetIn()**, **SetOut()** и **SetError()**, являющихся членами класса **Console**.

Пример 13

```
using System;
using System.IO;
class Redirect {
    static void Main() {
        StreamWriter log_out = null;
        try {
            log_out = new StreamWriter("logfile.txt");
            // переадресовать стандартный вывод в файл logfile.txt
            Console.SetOut(log_out);
            Console.WriteLine("Это начало файла
журнала регистрации.");
            for (int i=0; i<10; i++) Console.WriteLine(i);
        }
    }
}
```


Пример 13

```
Console.WriteLine("Это конец файла журнала  
регистрации.");  
} catch(IOException exc) {  
    Console.WriteLine("Ошибка ввода-  
вывода\n" + exc.Message);  
} finally {  
    if (log_out != null) log_out.Close();  
}  
}  
}
```

Чтение и запись двоичных данных

В C# имеется также возможность читать и записывать другие типы данных. Например, можно создать файл, содержащий данные типа **int**, **double** или **short**. Для чтения и записи двоичных значений встроенных типов данных служат классы потоков **BinaryReader** и **BinaryWriter**.

Используя эти потоки, следует иметь в виду, что данные считываются и записываются во внутреннем двоичном формате, а не в удобочитаемой текстовой форме.

Класс **BinaryWriter**

Класс **BinaryWriter** служит оболочкой, в которую заключается байтовый поток, управляющий выводом двоичных данных. Наиболее часто употребляемый конструктор этого класса:

BinaryWriter(Stream output);

где **output** обозначает поток, в который выводятся записываемые данные. Для записи в выходной файл в качестве параметра **output** может быть указан объект, создаваемый средствами класса **FileStream**.

Наиболее часто используемые методы, определенные в классе **BinaryWriter**: **Write(value)**, **Seek()**, **Close()** и **Flush()**.

Класс `BinaryReader`

Класс `BinaryReader` служит оболочкой, в которую заключается байтовый поток, управляющий вводом двоичных данных. Наиболее часто употребляемый конструктор этого класса: `BinaryReader(Stream input);` где **`input`** обозначает поток, из которого вводятся считываемые данные. Для чтения из входного файла в качестве параметра **`input`** может быть указан объект, создаваемый средствами класса **`FileStream`**.

Класс `BinaryReader`

Наиболее часто используемые методы,

определенные в классе `BinaryReader`:

`bool ReadBoolean()`, `byte ReadByte()`, `sbyte ReadSByte()`, `byte [] ReadBytes(int count)`, `char ReadChar()`, `char [] ReadChars(int count)`, `decimal ReadDecimal()`, `double ReadDouble()`, `float ReadSingle()`, `short ReadInt16()`, `int ReadInt32()`, `long ReadInt64()`, `ushort ReadUInt16()`, `uint ReadUInt32()`, `ulong ReadUInt64()`, `string ReadString()`

`int Read()`

`int Read(byte [] buffer, int offset, int count)`

`int Read(char [] buffer, int offset, int count)`

Пример 14

```
using System;
using System.IO;
class RWData {
    static void Main() {
        BinaryWriter dataOut;
        BinaryReader dataIn;
        int i = 10;
        double d = 1023.56;
        bool b = true;
        string str = "ЭТО ТЕСТ";
```

Пример 14

```
// открыть файл для вывода
try {
    dataOut = new BinaryWriter(new
FileStream("testdata", FileMode.Create));
} catch(IOException exc) {
    Console.WriteLine("Ошибка
открытия файла:\n" + exc.Message);
    return;
}
```

Пример 14

```
try { // записать данные в файл
    Console.WriteLine("Запись " + i) ;
    dataOut.Write(i) ;
    Console.WriteLine("Запись " + d);
    dataOut.Write(d);
    Console.WriteLine("Запись " + b);
    dataOut.Write(b);
    Console.WriteLine("Запись " + 12.2 * 7.4);
    dataOut.Write(12.2 * 7.4);
    Console.WriteLine("Запись " + str);
    dataOut.Write(str);
}
```


Пример 14

```
catch(IOException exc) {  
    Console.WriteLine("Ошибка ввода-  
вывода:\n" + exc.Message);  
} finally {  
    dataOut.Close();  
    Console.WriteLine();  
}
```

Пример 14

```
// прочитать данные из файла
try {
    dataIn = new BinaryReader(new
FileStream("testdata", FileMode.Open));
}
catch(IOException exc) {
    Console.WriteLine("Ошибка открытия
файла:\n" + exc.Message) ,
    return;
}
```

Пример 14

```
try {  
    i = dataIn.ReadInt32();  
    Console.WriteLine("Чтение " + i);  
    d = dataIn.ReadDouble();  
    Console.WriteLine("Чтение " + d);  
    b = dataIn.ReadBoolean();  
    Console.WriteLine("Чтение " + b);  
    d = dataIn.ReadDouble();  
    Console.WriteLine("Чтение " + d);  
    str = dataIn.ReadString();  
    Console.WriteLine("Чтение " + str);  
}
```

Пример 14

```
catch(IOException exc) {  
    Console.WriteLine("Ошибка  
ввода-вывода:\n" + exc.Message);  
} finally {  
    dataIn.Close();  
}  
}  
}
```

Пример 15

```
using System;
using System.IO;
class Inventory {
    static void Main() {
        BinaryWriter dataOut;
        BinaryReader dataIn;
        string item = "", st = ""; // наименование предмета
        int onhand, k = 0, ch; // имеющееся в наличии кол-во
        double cost; // цена
```

Пример 15

```
for ( ; ; ) {  
  do {  
    Console.Clear(); // очистка экрана  
    Console.WriteLine( "1. Ввод данных о товаре");  
    Console.WriteLine( "2. Поиск товара");  
    Console.WriteLine( "3. Выход");  
    Console.WriteLine( "Выберите пункт меню");  
    ch = Convert.ToInt32(Console.ReadLine());  
  } while(ch != 1 && ch != 2 && ch != 3);  
}
```

Пример 15

```
switch(ch) {  
    case 1:  
        try {  
            dataOut = new BinaryWriter(new  
FileStream("inventory.dat", FileMode.Create));  
        } catch(IOException exc) {  
            Console.WriteLine("Не удастся открыть  
файл " + "товарных запасов для вывода");  
            Console.WriteLine("Причина: " +  
exc.Message);  
            return;  
        }  
}
```

Пример 15

```
// записать данные о товарных запасах в файл
try {
    Console.WriteLine("Введите данные о
товарах, для выхода нажмите Enter ");
    for ( ; ; ) {
        Console.Write("наименование предмета - ");
        item = Console.ReadLine();
        if (item == "") break;
        Console.Write("имеющееся в наличии
количество - ");
        st = Console.ReadLine();
        if (st == "") break;
    }
}
```


Пример 15

```
onhand = Convert.ToInt32(st);  
Console.Write("цена - ");  
st = Console.ReadLine();  
if (st == "") break;  
cost = Convert.ToDouble (st);  
dataOut.Write(item);  
dataOut.Write(onhand );  
dataOut.Write(cost);  
k++;  
}  
}
```

Пример 15

```
catch(IOException exc) {  
    Console.WriteLine("Ошибка записи в файл  
товарных запасов");  
    Console.WriteLine("Причина: " +  
exc.Message);  
} finally {  
    dataOut.Close();  
}  
break;
```

Пример 15

case 2:

```
if (k == 0) {  
    Console.WriteLine("Вы не ввели данные !!!");  
}  
else {  
    Console.WriteLine();  
// открыть файл товарных запасов для чтения  
try {  
    dataIn = new BinaryReader(new  
FileStream("inventory.dat", FileMode.Open));  
}
```

Пример 15

```
catch(IOException exc) {  
    Console.WriteLine("Не удастся открыть  
файл товарных запасов для ввода");  
    Console.WriteLine("Причина: " +  
exc.Message);  
    return;  
}  
  
// найти предмет, введенный пользователем  
Console.Write("Введите наименование для  
поиска: ");  
string what = Console.ReadLine();  
Console.WriteLine();
```

Пример 15

```
try { // пока не достигнут конец файла
// считывать каждое значение из файла
while (dataIn.PeekChar() > -1){
// читать данные о предмете хранения
    item = dataIn.ReadString();
    onhand = dataIn.ReadInt32();
    cost = dataIn.ReadDouble();

// проверить, совпадает ли он с запрашиваемым предметом,
// если совпадает, то отобразить сведения о нем
    if (item.Equals(what,
StringComparison.OrdinalIgnoreCase)) {
```

Пример 15

```
Console.WriteLine(item+": "+onhand+"  
штук в наличии. "+"Цена: {0:C} за штуку",  
cost);
```

```
Console.WriteLine("Общая стоимость по  
наименованию <{0}>: {1}", item, cost *  
onhand);
```

```
    break;
```

```
    }
```

```
  }
```

```
} catch(EndOfStreamException) {
```

```
    Console.WriteLine("Предмет не найден.");
```

```
}
```

Пример 15

```
catch(IOException exc) {  
    Console.WriteLine("Ошибка чтения из  
файла товарных запасов");  
    Console.WriteLine("Причина: " +  
exc.Message);  
} finally {  
    dataIn.Close();  
}  
} // else  
Console.ReadKey(); // остановка экрана  
break;
```

Пример 15

```
case 3: Environment.Exit(0);
```

```
    break;
```

```
    } // end switch(ch)
```

```
    } // end for ( ; ; )
```

```
    }
```

```
    }
```


Буферизованный поток

Чтобы выполнить чтение и запись двоичного файла, можно создать два объекта класса **Stream**, один для чтения, а другой для записи.

```
Stream inputStream =
```

```
File.OpenRead(@"C:\test\source\test1.cs");
```

```
Stream outputStream =
```

```
File.OpenWrite(@"C:\test\source\test1.bak");
```

Чтение двоичного файла всегда выполняется через буфер. Буфер - это просто байтовый массив, содержащий данные, прочитанные методом **Read()**.

Буферизованный поток

Метод **Read()** читает байты из внешней памяти, сохраняет их в буфере и возвращает количество считанных байт.

Он продолжает читать, пока не прочитает все данные:

```
const int SizeBuff = 1024;
byte[] buffer = new Byte[SizeBuff];
int bytesRead;
while ( (bytesRead = inputStream.Read(buffer, 0,
    SIZE_BUFF)) > 0 ) {
    outputStream.Write(buffer, 0, bytesRead);
}
```

Буферизованный поток

Буферизованный поток - это объект,

позволяющий операционной системе создавать собственный внутренний буфер для обмена данными с внешней памятью и самостоятельно определять число байтов, считываемых или записываемых за один раз.

Программа по-прежнему будет заполнять буфер указанными порциями, но данные в этот буфер будут поступать из внутреннего системного буфера, а не из внешней памяти. В результате ввод/вывод будет происходить эффективнее и, следовательно, быстрее.

Буферизованный поток

Объект **BufferedStream** надстраивается над существующим объектом **Stream**, и, чтобы им воспользоваться, программист должен сначала создать обычный класс потока, как и раньше:

```
Stream inputStream =
```

```
File.OpenRead(@"C:\test\source\folder3.cs");
```

```
Stream outputStream =
```

```
File.OpenWrite(@"C:\test\source\folder3.bak");
```

После этого надо передать объект класса **Stream** конструктору буферизованного потока:

```
BufferedStream bufferedInput = new
```

```
BufferedStream(inputStream);
```

```
BufferedStream bufferedOutput = new
```

```
BufferedStream(outputStream);
```

Буферизованный поток

Затем с объектом класса **BufferedStream** можно обращаться как с обычным потоком, вызывая методы **Read()** и **Write()**, операционная система сама будет выполнять буферизацию:

```
while ( (bytesRead = bufferedInput.Read(buffer, 0,
    SIZE.BUFF)) > 0 ) {
    bufferedOutput.Write(buffer, 0, bytesRead);
}
```

Единственной особенностью применения буферизованных потоков является необходимость принудительно освободить буфер, чтобы гарантировать, что все данные выведены в файл:

```
bufferedOutput.Flush();
```

Пример 16

```
namespace Programming_CSharp {  
using System;  
using System.IO;  
class Tester {  
    const int SizeBuff = 1024;  
    public static void Main() {  
        // создать объект класса и выполнить его  
        Tester t = new Tester();  
        t.Run();  
    }  
}
```

Пример 16

// запустить с именем каталога

```
private void Run() { // создать двоичные потоки
```

```
    Stream inputStream =
```

```
    File.OpenRead(@"C:\test\source\folder3.cs");
```

```
    Stream outputStream =
```

```
    File.OpenWrite(@"C:\test\source\folder3.bak");
```

// надстроить буферизованные потоки над
двоичными

```
    BufferedStream bufferedInput = new  
    BufferedStream(inputStream);
```

```
    BufferedStream bufferedOutput = new  
    BufferedStream(outputStream);
```

Пример 16

```
byte[] buffer = new Byte[SizeBuff];
int bytesRead;
while ( (bytesRead =
bufferedInput.Read(buffer,0,SizeBuff)) > 0 ) {
    bufferedOutput.Write(buffer,0,bytesRead);
}
bufferedOutput.Flush();
bufferedInput.Close();
bufferedOutput.Close();
}
}
}
```


Контрольные вопросы

1. Какие пространства имен нужны приложению для работы с файлами?
2. Когда для записи файла следует использовать объект `FileStream` вместо объекта `StreamWriter`?
3. Какие методы класса `StreamWriter` позволяют читать данные из файлов, и что делает каждый из них?
4. Какой класс желательно использовать для сжатия потока с применением алгоритма `Deflate`?