

Организация C#-системы ввода-вывода

C#-программы выполняют операции ввода-вывода посредством потоков, которые построены на иерархии классов. *Поток* (stream) — это абстракция, которая генерирует и принимает данные. С помощью потока можно читать данные из различных источников (клавиатура, файл) и записывать в различные источники (принтер, экран, файл). Несмотря на то, что потоки связываются с различными физическими устройствами, характер поведения всех потоков одинаков. Поэтому классы и методы ввода-вывода можно применить ко многим типам устройств.

На самом низком уровне иерархии потоков ввода-вывода находятся потоки, оперирующие байтами. Это объясняется тем, что многие устройства при выполнении операций ввода-вывода ориентированы на байты. Однако для человека привычнее оперировать символами, поэтому разработаны символьные потоки, которые фактически представляют собой оболочки, выполняющие преобразование байтовых потоков в символьные и наоборот. Кроме этого, реализованы потоки для работы с int-, double-, short- значениями, которые также представляют оболочку для байтовых потоков, но работают не с самими значениями, а с их внутренним представлением в виде двоичных кодов.

Центральную часть потоковой C#-системы занимает класс Stream пространства имен System.IO. Класс Stream представляет байтовый поток и является базовым для всех остальных потоковых классов. Из класса Stream выведены такие байтовые классы потоков как:

- 1) FileStream - байтовый поток, разработанный для файлового ввода-вывода
- 2) BufferedStream - заключает в оболочку байтовый поток и добавляет буферизацию, которая во многих случаях увеличивает производительность программы;
- 3) MemoryStream - байтовый поток, который использует память для хранения данных.

Программист может вывести собственные потоковые классы. Однако для подавляющего большинства приложений достаточно встроенных потоков.

Подробно мы рассмотрим класс FileStream, классы StreamWriter и StreamReader, представляющие собой оболочки для класса FileStream и позволяющие преобразовывать байтовые потоки в символьные, а также классы BinaryWriter и BinaryReader, представляющие собой оболочки для класса FileStream и позволяющие преобразовывать байтовые потоки в двоичные для работы с int-, double-, short- и т.д. значениями.

Байтовый поток

Чтобы создать байтовый поток, связанный с файлом, создается объект класса FileStream. При этом в классе определено несколько конструкторов. Чаще всего используется конструктор, который открывает поток для чтения и/или записи:

`FileStream(string filename, FileMode mode)`

где:

- 1) параметр filename определяет имя файла, с которым будет связан поток ввода-вывода данных; при этом filename определяет либо полный путь к файлу, либо имя файла, который находится в папке bin/debug вашего проекта.
- 2) параметр mode определяет режим открытия файла, который может принимать одно из возможных значений, определенных перечислением FileMode:
 - a) FileMode.Append - предназначен для добавления данных в конец файла;
 - b) FileMode.Create — предназначен для создания нового файла, при этом если существует файл с таким же именем, то он будет предварительно удален;
 - c) FileMode.CreateNew - предназначен для создания нового файла, при этом файл с таким же именем не должен существовать;
 - d) FileMode.Open - предназначен для открытия существующего файла;

- e) `FileMode.OpenOrCreate` - если файл существует, то открывает его, в противном случае создает новый
- f) `FileMode.Truncate` - открывает существующий файл, но усекает его длину до нуля

Если попытка открыть файл оказалась неуспешной, то генерируется одно из исключений: `FileNotFoundException` - файл невозможно открыть по причине его отсутствия, `IOException` - файл невозможно открыть из-за ошибки ввода-вывода, `ArgumentNullException` - имя файла представляет собой null-значение, `ArgumentException` - некорректен параметр `mode`, `SecurityException` - пользователь не обладает правами доступа, `DirectoryNotFoundException` - некорректно задан каталог.

Другая версия конструктора позволяет ограничить доступ только чтением или только записью:

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

где:

- 1) параметры `filename` и `mode` имеют то же назначение, что и в предыдущей версии конструктора;
- 2) параметр `how`, определяет способ доступа к файлу и может принимать одно из значений, определенных перечислением `FileAccess`:
 - a) `FileAccess.Read` – только чтение;
 - b) `FileAccess.Write` – только запись;
 - c) `FileAccess.ReadWrite` - и чтение, и запись.

После установления связи байтового потока с физическим файлом внутренний указатель потока устанавливается на начальный байт файла.

Для чтения очередного байта из потока, связанного с физическим файлом, используется метод `ReadByte()`. После прочтения очередного байта внутренний указатель перемещается на следующий байт файла. Если достигнут конец файла, то метод `ReadByte()` возвращает значение - 1.

Для побайтовой записи данных в поток используется метод `WriteByte()`.

По завершении работы с файлом его необходимо закрыть. Для этого достаточно вызвать метод `Close ()`. При закрытии файла освобождаются системные ресурсы, ранее выделенные для этого файла, что дает возможность использовать их для работы с другими файлами.

Рассмотрим пример использования класса `FileStream`, для копирования одного файла в другой. Но вначале создадим текстовый файл `text.txt` в папке `bin/debug` текущего проекта. И внесем в него произвольную информацию, например:

```
12 456
Hello!
23,67 4: Message
```

```
using System;
using System.Text;
using System.IO;    //для работы с потоками
```

```
namespace MyProgram
{
    class Program
    {
        static void Main()
        {
            try
            {
                FileStream fileIn = new FileStream("text.txt", FileMode.Open, FileAccess.Read);
```

```

FileStream fileOut = new FileStream("newText.txt", FileMode.Create, FileAccess.Write);
int i;
while ((i = fileIn.ReadByte())!=-1)
{
    //запись очередного файла в поток, связанный с файлом fileOut
    fileOut.WriteByte((byte)i);
}
fileIn.Close();
fileOut.Close();
}
catch (Exception EX)
{
    Console.WriteLine(EX.Message);
}
}
}
}

```

Задание. Подумайте, почему для переменной i указан тип int. Можно было бы указать тип byte?

Символьный поток

Чтобы создать символьный поток нужно поместить объект класса Stream (например, FileStream) «внутри» объекта класса StreamWriter или объекта класса StreamReader. В этом случае байтовый поток будет автоматически преобразовываться в символьный.

Классе StreamWriter предназначен для организации выходного символьного потока. В нем определено несколько конструкторов. Один из них записывается следующим образом:

```
StreamWriter(Stream stream);
```

где параметр stream определяет имя уже открытого байтового потока.

Например, создать экземпляр класса StreamWriter можно следующим образом:

```
StreamWriter fileOut=new StreamWriter(new FileStream("text.txt", FileMode.Create, FileAccess.Write));
```

Этот конструктор генерирует исключение типа ArgumentException, если поток stream не открыт для вывода, и исключение типа ArgumentNullException, если он (поток) имеет null-значение.

Другой вид конструктора позволяет открыть поток сразу через обращения к файлу:

```
StreamWriter(string name);
```

где параметр name определяет имя открываемого файла.

Например, обратиться к данному конструктору можно следующим образом:

```
StreamWriter fileOut=new StreamWriter("c:\\temp\\t.txt");
```

И еще один вариант конструктора StreamWriter:

```
StreamWriter(string name, bool appendFlag);
```

где параметр name определяет имя открываемого файла;

параметр appendFlag может принимать значение true – если нужно добавлять данные в конец файла, или false – если файл необходимо перезаписать.

Например:

```
StreamWriter fileOut=new StreamWriter("t.txt", true);
```

Теперь для записи данных в поток fileOut можно обратиться к методу WriteLine. Это можно сделать следующим образом:

```
fileOut.WriteLine("test");
```

В данном случае в конец файла t.txt будет дописано слово test.

Класс StreamReader предназначен для организации входного символьного потока. Один из его конструкторов выглядит следующим образом:

```
StreamReader(Stream stream);
```

где параметр stream определяет имя уже открытого байтового потока.

Этот конструктор генерирует исключение типа ArgumentException, если поток stream не открыт для ввода.

Например, создать экземпляр класса StreamWriter можно следующим образом:

```
StreamReader fileIn = new StreamReader(new FileStream("text.txt", FileMode.Open, FileAccess.Read));
```

Как и в случае с классом StreamWriter у класса StreamReader есть и другой вид конструктора, который позволяет открыть файл напрямую:

```
StreamReader (string name);
```

где параметр name определяет имя открываемого файла.

Обратиться к данному конструктору можно следующим образом:

```
StreamReader fileIn=new StreamReader ("c:\temp\t.txt");
```

В C# символы реализуются кодировкой Unicode. Для того, чтобы можно было обрабатывать текстовые файлы, содержащие русский символы, созданные, например, в Блокноте, рекомендуется вызывать следующий вид конструктора StreamReader:

```
StreamReader fileIn=new StreamReader ("c:\temp\t.txt", Encoding.GetEncoding(1251));
```

Параметр Encoding.GetEncoding(1251) говорит о том, что будет выполняться преобразование из кода Windows-1251 (одна из модификаций кода ASCII, содержащая русские символы) в Unicode. Encoding.GetEncoding(1251) реализован в пространстве имен System.Text.

Теперь для чтения данных из потока fileIn можно воспользоваться методом ReadLine. При этом если будет достигнут конец файла, то метод ReadLine вернет значение null.

Рассмотрим пример, в котором данные из одного файла копируются в другой, но уже с использованием классов StreamWriter и StreamReader.

```
static void Main()
{
    StreamReader fileIn = new StreamReader("text.txt", Encoding.GetEncoding(1251));
    StreamWriter fileOut=new StreamWriter("newText.txt", false);
    string line;
    while ((line=fileIn.ReadLine())!=null) //пока поток не пуст
    {
        fileOut.WriteLine(line);
    }
    fileIn.Close();
    fileOut.Close();
}
```

<p>Задание. Выясните, для чего предназначен метод ReadToEnd() и когда имеется смысл его применять.</p>

Таким образом, данный способ копирования одного файла в другой, даст нам тот же результат, что и при использовании байтовых потоков. Однако, его работа будет менее эффективной, т.к. будет тратиться дополнительное время на преобразование байтов в символы. Но у символьных потоков есть свои преимущества. Например, мы можем использовать регулярные выражения для поиска заданных фрагментов текста в файле.

```
static void Main()
{
    StreamReader fileIn = new StreamReader("text.txt");
    StreamWriter fileOut=new StreamWriter("newText.txt", false);
    string text=fileIn.ReadToEnd();
    Regex r= new Regex(@"[-+]?d+");
    Match integer = r.Match(text);
    while (integer.Success)
    {
        fileOut.WriteLine(integer);
        integer = integer.NextMatch();
    }
    fileIn.Close();
    fileOut.Close();
}
```

Двоичные потоки

Двоичные файлы хранят данные в том же виде, в котором они представлены в оперативной памяти, то есть во внутреннем представлении. Двоичные файлы не применяются для просмотра человеком, они используются только для программной обработки.

Выходной поток `BinaryWriter` поддерживает произвольный доступ, т.е. имеется возможность выполнять запись в произвольную позицию двоичного файла. Наиболее важные методы потока `BinaryWriter`:

Член класса	Описание
<code>BaseStream</code>	Определяет базовый поток, с которым работает объект <code>BinaryWriter</code>
<code>Close</code>	Закрывает поток
<code>Flush</code>	Очищает буфер
<code>Seek</code>	Устанавливает позицию в текущем потоке
<code>Write</code>	Записывает значение в текущий поток

Наиболее важные методы выходного потока `BinaryReader`:

Член класса	Описание
<code>BaseStream</code>	Определяет базовый поток, с которым работает объект <code>BinaryReader</code>
<code>Close</code>	Закрывает поток
<code>PeekChar</code>	Возвращает следующий символ потока без перемещения внутреннего указателя в потоке
<code>Read</code>	Считывает очередной поток байтов или символов и сохраняет в массиве, передаваемом во входном параметре
<code>ReadBoolean</code> , <code>ReadByte</code> , <code>ReadInt32</code> и т.д	Считывает из потока данные определенного типа

Двоичный поток открывается на основе базового протока (например, `FileStream`), при этом двоичный поток будет преобразовывать байтовый поток в значения `int`-, `double`-, `short`- и т.д.

Рассмотрим пример формирования двоичного файла:

```

static void Main()
{
    //открываем двоичный поток
    BinaryWriter fOut=new BinaryWriter(new FileStream("t.dat",FileMode.Create));
    //записываем данные в двоичный поток
    for (int i=0; i<=100; i+=2)
    {
        fOut.Write(i);
    }
    fOut.Close(); //закрываем двоичный поток
}

```

Попытка просмотреть двоичный файл через текстовый редактор неинформативна. Двоичный файл просматривается программным путем, например следующим образом:

```

static void Main()
{
    FileStream f=new FileStream("t.dat",FileMode.Open);
    BinaryReader fIn=new BinaryReader(f);
    long n=f.Length/4; //определяем количество чисел в двоичном потоке
    int a;
    for (int i=0; i<n; i++)
    {
        a=fIn.ReadInt32();
        Console.Write(a+" ");
    }
    fIn.Close();
    f.Close();
}

```

Двоичные файлы являются файлами с произвольным доступом, при этом нумерация элементов в двоичном файле ведется с нуля. Произвольный доступ обеспечивает метод Seek. Рассмотрим его синтаксис:

Seek(long newPos, SeekOrigin pos)

где параметр newPos определяет новую позицию внутреннего указателя файла в байтах относительно исходной позиции указателя, которая определяется параметром pos. В свою очередь параметр pos должен быть задан одним из значений перечисления SeekOrigin:

Значение	Описание
SeekOrigin.Begin	Поиск от начала файла
SeekOrigin.Current	Поиск от текущей позиции указателя
SeekOrigin.End	Поиск от конца файла

После вызова метода Seek следующие операции чтения или записи будут выполняться с новой позиции внутреннего указателя файла.

Рассмотрим пример организации произвольного доступа к двоичному файлу (на примере файла t.dat):

```

static void Main()
{
    //изменение данных в двоичном потоке
    FileStream f=new FileStream("t.dat",FileMode.Open);
    BinaryWriter fOut=new BinaryWriter(f);
    long n=f.Length; //определяем количество байт в байтовом потоке
}

```

```

int a;
for (int i=0; i<n; i+=8) //сдвиг на две позиции, т.к. тип int занимает 4 байта
{
    fOut.Seek(i,SeekOrigin.Begin);
    fOut.Write(0);
}
fOut.Close();
//чтение данных из двоичного потока
f=new FileStream("t.dat",FileMode.Open);
BinaryReader fIn=new BinaryReader(f);
n=f.Length/4; //определяем количество чисел в двоичном потоке
for (int i=0; i<n; i++)
{
    a=fIn.ReadInt32();
    Console.Write(a+" ");
}
fIn.Close();
f.Close();
}

```

Поток BinaryReader не имеет метода Seek, однако используя возможности потока FileStream можно организовать произвольный доступ при чтении двоичных файлов.

Рассмотрим следующий пример:

```

static void Main()
{
    //Записываем в файл t.dat целые числа от 0 до 100
    FileStream f=new FileStream("t.dat",FileMode.Open);
    BinaryWriter fOut=new BinaryWriter(f);
    for (int i=0; i<100; ++i)
    {
        fOut.Write(i);
    }
    fOut.Close();
    //Объекты f и fIn связаны с одним и тем же файлом
    f=new FileStream("t.dat",FileMode.Open);
    BinaryReader fIn=new BinaryReader(f);
    long n=f.Length; //определяем количество байт потока
    //Читаем данные из файла t.dat, перемещая внутренний указатель на 8 байт, т.е. на два целых числа
    for (int i=0; i<n; i+=8)
    {
        f.Seek(i,SeekOrigin.Begin);
        int a=fIn.ReadInt32();
        Console.Write(a+" ");
    }
    fIn.Close();
    f.Close();
}

```

Перенаправление стандартных потоков

Тремя стандартными потоками, доступ к которым осуществляется через свойства Console.Out, Console.In и Console.Error, могут пользоваться все программы, работающие в пространстве имен System. Свойство Console.Out относится к стандартному выходному потоку.

По умолчанию это консоль. Например, при вызове метода `Console.WriteLine()` информация автоматически передается в поток `Console.Out`. Свойство `Console.In` относится к стандартному входному потоку, источником которого по умолчанию является клавиатура. Например, при вводе данных с клавиатуры информация автоматически передается потоку `Console.In`, к которому можно обратиться с помощью метода `Console.ReadLine()`. Свойство `Console.Error` относится к ошибкам в стандартном потоке, источником которого также по умолчанию является консоль. Однако эти потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода, например, на работу с физическими файлами.

Перенаправить стандартный поток можно с помощью методов `SetIn()`, `SetOut()` и `SetError()`, которые являются членами класса `Console`:

```
static void SetIn(TextReader input)
static void SetOut(TextWriter output)
static void SetError(TextWriter output)
```

Пример перенаправления потоков проиллюстрирован следующей программой, в которой двумерный массив вводится из файла `input.txt`, а выводится в файл `output.txt`

```
static void Main()
{
    try
    {
        int[,] MyArray;
        StreamReader file=new StreamReader("input.txt");
        Console.SetIn(file); // перенаправляем стандартный входной поток на file
        string line=Console.ReadLine();
        string []mas=line.Split(' ');
        int n=int.Parse(mas[0]);
            int m=int.Parse(mas[1]);
        MyArray = new int[n,m];
        for (int i = 0; i < n; i++)
        {
            line = Console.ReadLine();
            mas = line.Split(' ');
            for (int j = 0; j < m; j++)
            {
                MyArray[i,j] = int.Parse(mas[j]);
            }
        }
        PrintArray("исходный массив:", MyArray, n, m);
        file.Close();
    }
}

static void PrintArray(string a, int[,] mas, int n, int m)
{
    StreamWriter file=new StreamWriter("output.txt"); // перенаправляем стандартный входной поток на file
    Console.SetOut(file);
    Console.WriteLine(a);
    for (int i = 0; i < n; i++)
    {
        for (int j=0; j<m; j++) Console.Write("{0} ", mas[i,j]);
        Console.WriteLine();
    }
    file.Close();
}
```



```
_____input.txt_____
3 4
1 4 2 8
4 9 0 1
5 7 4 2
```

При необходимости восстановить исходное состояние потока Console.In можно следующим образом:

```
TextWriter str = Console.In; // первоначально сохраняем исходное состояние входного потока
...
Console.SetIn(str); // при необходимости восстанавливаем исходное состояние входного потока
```

Аналогичным образом можно восстановить исходное состояние потока Console.Out:

```
TextWriter str = Console.Out; // первоначально сохраняем исходное состояние выходного потока
...
// при необходимости восстанавливаем исходное состояние выходного потока
Console.SetOut(str);
```

<p>Задание. Подумайте для чего нужно два потока Console.Out и Console.Error, если они оба при стандартной работе выводят информацию на экран.</p>
--