**Организация С#-системы ввода-вывода**

Цель: изучить принципы организации ввода-вывода C#. Рассмотреть байтовый, символьный и двоичный потоки. Изучить принципы работы с файловой системой. Рассмотреть классы для работы с каталогами и файлами.

С#-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 С#-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 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.

1. 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:

* FileMode.Append - предназначен для добавления данных в конец файла;
* FileMode.Create - предназначен для создания нового файла, при этом если существует
* файл с таким же именем, то он будет предварительно удален;
* FileMode.CreateNew - предназначен для создания нового файла, при этом файл с таким
* же именем не должен существовать;
* FileMоde.Open - предназначен для открытия существующего файла;
* FileMode.ОpenOrCreate - если файл существует, то открывает его, в противном случае
* создает новый
* 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:

1. FileAccess.Read - только чтение;

2. FileAccess.Write - только запись;

3. 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 достинут 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.

Пример использования класса FileStream, для копирования одного файла в другой. 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);

}

}

}

}

***2. Символьный поток***

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

Класс 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, 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.

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();

}

Задание. Выясните, для чего предназначен метод ReadToEnd() и когда имеется смысл его

применять.

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();

}

**3. 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 |
| BaseStream | Определяет базовый поток, с которым работает объект BinaryWriter |
| Close | Закрывает поток |
| Flush | Очищает буфер |
| Seek | Устанавливает позицию в текущем потоке |
| Write | Записывает значение в текущий поток |

Наиболее важные методы выходного потока BinaryReader:

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

Двоичный поток открывается на основе базового потока (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.

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();

f=new FileStream("t.dat",FileMode.Open); //чтение данных из двоичного потока

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 потоке

//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();

}**4. 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 Setln(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);

**5. Работа с файловой системой: классы Directory и Filе и классы DirectoryInfo и FileInfo**

В пространстве имен System.IO предусмотрено четыре класса, которые предназначены для работы

с файловой системой компьютера, т.е для создания, удаления переноса и т.д. файлов и каталогов.

Первые два типа - Directory и Fi1е реализуют свои возможности с помощью статических методов,

поэтому данные классы можно использовать без создания соответствующих объектов (экземпляров классов).

Следующие типы - DirectoryInfo и FileInfo обладают схожими функциональными возможностями

c Directory и Fi1е, но порождены от класса FileSystemInfo и поэтому реализуются путем создания

соответствующих экземпляров классов.

**Работа с каталогами**

**Абстрактный класс FileSystemInfo**

Значительная часть членов FileSystemInfo предназначена для работы с общими характеристиками файла или каталога (метками времени, атрибутами и т. п.). Рассмотрим некоторые свойства FileSystemInfo:

**Attributes**- Позволяет получить или установить атрибуты для данного объекта файловой

системы. Для этого свойства используются значения и перечисления FileAttributes

**CreationTime** - Позволяет получить или установить время создания объекта файловой системы

**Exists-** Может быть использовано для того, чтобы определить, существует ли данный

объект файловой системы

**Extension** - Позволяет получить расширение для файла

**FullName** - Возвращает имя файла или каталога с указанием пути к нему в файловой системе

**LastAccessTime** - Позволяет получить или установить время последнего обращения к объекту

файловой системы

**LastWriteTime** - Позволяет получить или установить время последнего внесения изменений в

объект файловой системы

**Name** - Возвращает имя указанного файла. Это свойство доступно только для чтения. Для

каталогов возвращает имя последнего каталога в иерархии, если это возможно. Если

нет, возвращает полностью определенное имя

В FileSystemInfo предусмотрено и несколько методов. Например, метод Delete() - позволяет

удалить объект файловой системы с жесткого диска, a Refresh() - обновить информацию об объекте файловой системы.

**Класс DirectoryInfo**

Данный класс наследует члены класса FileSystemInfo и содержит дополнительный набор членов, которые предназначены для создания, перемещения, удаления, получения информации о каталогах иподкаталогах в файловой системе. Наиболее важные члены класса содержатся в следующей таблице:

|  |  |
| --- | --- |
| Член | Описание |
| Create()  CreateSubDirectory() | Создают каталог (или подкаталог) по указанному пути в файловой  системе |
| Delete() | Удаляет пустой каталог |
| GetDirectories() | Позволяет получить доступ к подкаталогам текущего каталога (в виде  массива объектов DirectoryInfo) |
| GetFiles() | Позволяет получить доступ к файлам текущего каталога (в виде массива  объектов FileInfo) |
| MoveTo() | Перемещает каталог и все его содержимое на новый адрес в файловой  системе |
| Parent | Возвращает родительский каталог в иерархии файловой системы |

Работа с типом DirectoryInfo начинается с того, что мы создаем экземпляр класса (объект),

указывая при вызове конструктора в качестве параметра путь к нужному каталогу. Если мы хотим

обратиться к текущему каталогу (то есть каталогу, в котором в настоящее время производится

выполнение приложения), вместо параметра используется обозначение ".".

Например: // Создаем объект DirectoryInfo, которому будет обращаться к текущему каталогу

DirectoryInfo dir1 = new DirectoryInfo(".");

// Создаем объект DirectoryInfo, которому будет обращаться к каталогу d:\prim

DirectoryInfo dir2 = new DirectoryInfo(@"d:\prim");

Если мы попытаемся создать объект DirectoryInfo, связав его с несуществующим каталогом, то

будет сгенерировано исключение System.IO.DirectoryNotFoundException. Если же все нормально, то сможем получить доступ к данному каталогу. В примере, который приведен ниже, мы создаем

объект DlrectoryInfo, который связан с каталогом d:\prim, и выводим информацию о данном каталоге:

using System;

using System.Text;

using System.IO;

namespace MyProgram

{

class Program

{

static void Main(string[] args)

{

DirectoryInfo dir = new DirectoryInfo(@"d:\prim");

Console.WriteLine("\*\*\*\*\* "+dir.Name+" \*\*\*\*\*");

Console.WriteLine("FullName: {0}", dir.FullName);

Console.WriteLine("Name: {0}", dir.Name);

Console.WriteLine("Parent: {0}", dir.Parent);

Console.WriteLine("Creation: {0}", dir.CreationTime);

Console.WriteLine("Attributes: {0}", dir.Attributes.ToString());

Console.WriteLine("Root: {0}", dir.Root);

}

}

}

Свойство Attributes позволяет получить информацию об атрибутах объекта файловой системы.

Возможные значения данного свойства приведены ниже:

Archive- Этот атрибут используется приложениями при проведении резервного копирования, а в некоторых случаях - удаления старых файлов

Compressed - Определяет, что файл является сжатым

Directory - Определяет, что объект файловой системы является каталогом

Encrypted- Определяет, что файл является зашифрованным

Hidden - Определяет, что файл является скрытым (такой файл не будет выводиться при

обычном просмотре каталога)

Normal - Определяет, что файл находится в обычном состоянии и для него установлены

любые другие атрибуты. Этот атрибут не может использоваться с другими атрибутами

Offline- Файл (расположенный на сервере) кэширован в хранилище off-line на клиентском

компьютере. Возможно, что данные этого файла уже устарели

Readonly - Файл доступен только для чтения

System - Файл является системным (то есть файл является частью операционной системы

или используется исключительно операционной системой)

Через DirectoryInfo можно не только получать доступ к информации о текущем каталоге, но

получить доступ к информации о его подкаталогах:

class Program

{

static void printDirect( DirectoryInfo dir)

{

Console.WriteLine("\*\*\*\*\* "+dir.Name+" \*\*\*\*\*");

Console.WriteLine("FullName: {0}", dir.FullName);

Console.WriteLine("Name: {0}", dir.Name);

Console.WriteLine("Parent: {0}", dir.Parent);

Console.WriteLine("Creation: {0}", dir.CreationTime);

Console.WriteLine("Attributes: {0}", dir.Attributes.ToString());

Console.WriteLine("Root: {0}", dir.Root);

}

static void Main(string[] args)

{

DirectoryInfo dir = new DirectoryInfo(@"d:\prim");

printDirect(dir);

DirectoryInfo[] subDirects = dir.GetDirectories();

Console.WriteLine("Найдено {0} подкаталогов", subDirects.Length);

foreach (DirectoryInfo d in subDirects)

{

printDirect(d);

}

}

}

Метод CreateSubdirectory() позволяет создать в выбранном каталоге как единственный подкаталог,

так и множество подкаталогов (в том числе, и вложенных друг в друга). Создадим в каталоге

несколько дополнительных подкаталогов:

DirectoryInfo dir = new DirectoryInfo(@"d:\prim");

dir.CreateSubdirectory("doc"); //создали подкаталог

dir.CreateSubdirectory(@"book\2008"); //создали вложенный подкаталог

Метод MoveTo() позволяет переместить текущий каталог по заданному в качестве параметра

адресу. При этом возможно произвести переименование каталога. Например:

DirectoryInfo dir = new DirectoryInfo(@"d:\prim\bmp");

dir.MoveTo(@"d:\prim\letter\bmp");

В данном случае каталог bmp перемещается в по адресу d:\prim\letter\bmp. Так как имя

перемещаемого каталога совпадает с крайним правым именем в адресе нового местоположения

каталога, то переименования не происходит. Следующий пример позволит нам переименовать

текущий каталог:

DirectoryInfo dir = new DirectoryInfo(@"d:\prim\letter");

dir.MoveTo(@"d:\prim\archive");

**Класс Directory**

Работать с каталогами файловой системы компьютера можно и при помощи класса Directory,

функциональные возможности которого во многом совпадают с возможностями DirectoryInfo. Но

члены данного класса реализованы статически, поэтому для их использования нет необходимости

создавать объект.

Рассмотрим работу с методами данного класса на примерах.

Замечание. Удалите с диска d измененную папку prim. И еще раз скопируйте ее исходную версию

из раздела 12 данного электронного учебника.

Directory.CreateDirectory(@"d:\prim\2008"); //создали подкаталог 2008

Directory.Move(@"d:\prim\bmp", @"d:\prim\2008\bmp");//перенесли каталог bmp в каталог 2008

Directory.Move(@"d:\prim\letter", @"d:\prim\archives");//переименовали каталог letter в archives

Замечания.

1. Удаление каталога возможно только тогда, когда он пуст.

2. На практике комбинируют использование классов Directory и DirectoryInfo.

**Работа с файлами. Класс Filelnfo**

Класс Filelnfo предназначен для организации доступа к физическому файлу, который содержится

на жестком диске компьютера. Он позволяет получать информацию об этом файле (например, о

времени его создания, размере, атрибутах и т. п.), а также производить различные операции,

например, по созданию файла или его удалению. Класс FileInfo наследует члены класса

FileSystemInfo и содержит дополнительный набор членов, который приведен ниже:

AppendText() -0Создает объект StreamWriter для добавления текста к файлу

CopyTo() - Копирует уже существующий файл в новый файл

Create() - Создает новый файл и возвращает объект FileStream для взаимодействия с этим файлом

CreateText() - Создает объект StreamWriter для записи текстовых данных в новый файл

Delete() - Удаляет файл, которому соответствует объект FileInfo

Directory - Возвращает каталог, в котором расположен данный файл

DirectoryName - Возвращает полный путь к данному файлу в файловой системе

Length - Возвращает размер файла

MoveTo() - Перемещает файл в указанное пользователем место (этот метод позволяет

одновременно переименовать данный файл)

Name - Позволяет получить имя файла

Ореn() - Открывает файл с указанными пользователем правами доступа на чтение, запись

или совместное использование с другими пользователями

OpenRead() - Создает объект FileStream, доступный только для чтения

OpenText() - Создает объект StreamReader (о нем также будет рассказано ниже), который

позволяет считывать информацию из существующего текстового файла

OpenWrite() - Создает объект FileStream, доступный для чтения и записи

Большинство методов FileInfo возвращает объекты (FIleStream, StreamWriter, StreamReader и т. п.), которые позволяют различным образом взаимодействовать с файлом, например, производить чтение или запись в него. Приемы работы с данными потоками уже известны. Поэтому рассмотрим другие возможности класса FileInfo.

using System;

using System.Text;

using System.IO; //для работы с файловым вводом-выводом

using System.Text.RegularExpressions;

namespace MyProgram

{

class Program

{

static void Main()

{

//создаем новый файл и связываем с ним строковый поток

FileInfo f = new FileInfo("text.txt");

StreamWriter fOut = new StreamWriter(f.Create());

//записываем в файл данные и закрываем строковый поток,

// при этом связь с физическим файлом для f не рвется

fOut.WriteLine("ОДИН ДВА ТРИ...");

fOut.Close();

//получаем информацию о файле

Console.WriteLine("\*\*\*\*\*\*\*"+f.Name+"\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Console.WriteLine("File size: {0}", f.Length);

Console.WriteLine("Creation: {0}", f.CreationTime);

Console.WriteLine("Attributes: {0}", f.Attributes.ToString());

}

}

}

Замечание. Еще раз удалите с диска d измененную папку prim, затем скопирует ее исходную

версию из раздела 12 данного электронного учебника.

Рассмотрим следующий пример:

static void Main()

{

FileInfo f = new FileInfo(@"d:\prim\letter\letter1.txt");

f.CopyTo(@"d:\prim\bmp\letter.txt");

Directory.CreateDirectory(@"d:\prim\archives");

f.MoveTo(@"d:\prim\archives\letter1.txt");

f = new FileInfo(@"d:\prim\letter\letter2.txt");

f.Delete();

}