**Модель потоков в C#. Пространство System.IO**

**Класс Stream**

В зависимости от базового источника или хранилища данных потоки могут поддерживать только некоторые из возможностей (файл, устройство ввода-вывода, канал взаимодействия процессов, поток шифрования и др.). Приложение может запрашивать возможности потока с помощью свойств **CanRead**, **CanWrite** и **CanSeek**.

Методы **Read** и **Write** позволяют выполнять чтение и запись данных в различных форматах. Если поток поддерживает поиск, для отправки запросов и изменения текущего положения и длины потока рекомендуется использовать методы **Seek**, **SetLength**, а также свойства **Position**, **Length**.

Удаления внутренних буферов - метод **Flush**.

Конструктор **FileStream**:

*FileStream fs = new FileStream(filePath, FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite);*

Первым параметром передается путь к файлу

Переменная типа **FileMode** (перечисление).

|  |  |
| --- | --- |
| FileMode.Append | Открывает файл, если он существует, и перемещает курсор в конец файл. Если файл не существует – создает новый файл. FileMode.Append можно использовать только вместе с FileAccess.Write. |
| FileMode.Create | Создает новый файл, если файл уже существует, то он будет переписан. Курсор устанавливается на начало файла. |
| FileMode.CreateNew | Создает новый файл, если файл уже существует, то будет вызвано исключение IOException. Курсор устанавливается на начало файла. |
| FileMode.Open | Открывает существующий файл, если файла не существует, то будет вызвано исключение FileNotFoundException. Курсор устанавливается на начало файла. |
| FileMode.OpenOrCreate | Открывает существующий файл, если файла не существует, то будет создан новый файл. Курсор устанавливается на начало файла. |
| FileMode.Truncate | Открывает существующий файл и усекает его размер до нуля. |

Переменная типа *FileAccess* (перечисление, имеющее атрибут *FlagsAttribute*, т.е. поддерживающее побитовое соединение составляющих его значений с помощью оператора |). Он описывает, каким образом осуществляется доступ к файлу: запись (*FileAccess.Write*), чтение (*FileAccess.Read*) или запись и чтение (*FileAccess.ReadWrite*).

Переменная типа *FileShare* (тоже перечисление, имеющее атрибут *FlagsAttribute*). Этот параметр позволяет управлять доступом.

|  |  |
| --- | --- |
| FileShare.Delete | Разрешает удаление файла. |
| FileShare.Inheritable | Разрешает наследование дескриптора файла дочерними процессами. В Win32 непосредственная поддержка этого свойства не обеспечена. |
| FileShare.None | Отклоняет совместное использование текущего файла. Любой запрос на открытие файла (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. |
| FileShare.Read | Разрешает последующее открытие файла для чтения. Если этот флаг не задан, любой запрос на открытие файла для чтения (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. |
| FileShare.ReadWrite | Разрешает последующее открытие файла для чтения или записи. Если этот флаг не задан, любой запрос на открытие файла для записи или чтения (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. |
| FileShare.Write | Разрешает последующее открытие файла для записи. Если этот флаг не задан, любой запрос на открытие файла для записи (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. |

**Пример**

using System;

using System.IO;

using System.Text;

namespace FileStreamExample

{

class Program

{

// Демонстрационный пример

// работы с классом FileStream

static void Main(string[] args)

{

// Получаем путь к файлу

Console.WriteLine("Введите путь к файлу:");

string filePath = Console.ReadLine();

// Создаем объект FileStream -

// коментарии к конструктору см. ниже в уроке

// Использование блока using позволяет правильно

// разрушить FileStream

using (FileStream fs = new FileStream(filePath, FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite))

{

// Получаем данные для записи в файл

Console.WriteLine("Введите строку для записи в файл:");

string writeText = Console.ReadLine();

// Преобразуем строку в массив байт,

// т.к. FileStream умеет работать только с байтами

// Как вам известно, строку можно по-разному преобразовать

// в массив байт - все зависит от выбранной кодировки.

// В данном случае мы используем кодировку UTF-8 (Unicode),

// следовательно каждый символ будет преобразован

// в 2 байта согласно правилам UTF-8.

byte[] writeBytes = Encoding.UTF8.GetBytes(writeText);

// Запишем данные в файл

fs.Write(writeBytes, 0, writeBytes.Length);

// Сохраним данные из буфера на диск

fs.Flush();

// Теперь прочитаем данные из файла

// для этого нужно сначала установить курсор

// на начало файла

fs.Seek(0, SeekOrigin.Begin);

// Теперь прочитаем данные в другой массив байт

byte[] readBytes = new byte[(int)fs.Length];

fs.Read(readBytes, 0, readBytes.Length);

// Преобразуем байты в строку

// Для этого используем ту же кодировку, что и для записи,

// иначе нет гарантии, что мы получим правильную строку

string readText = Encoding.UTF8.GetString(readBytes);

// Выведем ее на консоль

Console.WriteLine("Данные, прочитанные из файла: {0}",

readText);

}

Console.Read();

}

}

}

**Использование класса StreamWriter и StreamReader для файловых операций**

**StreamReader**

Также у них определено несколько методов для чтения:

1) **Read** – читает следующий символ, либо массив символов;

2) **ReadBlock** – читает массив символов;

3) **ReadLine** – читает строку от текущей позиции до символа перехода на новую строчку.

4) **ReadToEnd** – читает поток от текущей позиции до конца.

StreamReader sr = new StreamReader(filePath, Encoding.UTF8);

// Прочитаем данные

string readText = sr.ReadToEnd();

**Пример 2**

namespace StreamWriterReaderExample

{

class Program

{

// Демонстрационный пример

// работы с классами StreamWriter

// и StreamReader

static void Main(string[] args)

{

// Получаем путь к файлу

Console.WriteLine("Введите путь к файлу:");

string filePath = Console.ReadLine();

// Создаем объект StreamWriter для

// записи строк в различных кодировках в файл.

// Создадим его как обертку для

// другого файлового потока

FileStream fs = new FileStream(filePath, FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite);

StreamWriter sw = new StreamWriter(fs, Encoding.UTF8);

// Получаем данные для записи в файл

Console.WriteLine("Введите строку для записи в файл:");

string writeText = Console.ReadLine();

// Запишем данные в поток

sw.Write(writeText);

// Сохраним данные из буфера на диск и закроем поток

sw.Dispose();

// Теперь прочитаем данные из файла

// для этого используем StreamReader.

// На этот раз воспользуемся другим конструктором

StreamReader sr = new StreamReader(filePath, Encoding.UTF8);

// Прочитаем данные

string readText = sr.ReadToEnd();

sr.Dispose();

// Выведем их на консоль

Console.WriteLine("Данные, прочитанные из файла: {0}", readText);

Console.Read();

}

}

}

**Использование класса BinaryWriter и BinaryReader для файловых операций**

Данные классы предназначены для записи простых типов данных в поток в виде двоичных значений, а также строк в определенной кодировке.

using System;

using System.IO;

namespace BinaryWriterReaderExample

{

class Program

{

static void Main(string[] args)

{

// Получаем путь к файлу

Console.WriteLine("Введите путь к файлу:");

string filePath = Console.ReadLine();

using (FileStream fs = new FileStream(filePath, FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite))

{

// Создаем объект BinaryWriter для

// записи простых типов данных в поток.

BinaryWriter bw = new BinaryWriter(fs);

// Получаем данные для записи в файл

Console.WriteLine("Введите строку для записи в файл:");

string writeText = Console.ReadLine();

int writeNum = -1;

while (true)

{

try

{

Console.WriteLine("Введите целое число " + "для записи в файл:");

writeNum = Convert.ToInt32(Console.ReadLine());

break;

}

catch

{

Console.WriteLine("Ошибка! " + "Не удалось преобразовать строку в целое число!");

}

}

// Запишем данные в поток

bw.Write(writeText);

bw.Write(writeNum);

// Сохраним данные из буфера на диск

bw.Flush();

// Установим курсор на начало файла

fs.Seek(0, SeekOrigin.Begin);

// Теперь прочитаем данные из файла

// для этого используем BinaryReader.

BinaryReader br = new BinaryReader(fs);

// Данные читать нужно в том же порядке,

// в котором они и записывались в поток

Console.WriteLine("Строка, прочитанная из файла:");

Console.WriteLine(br.ReadString());

Console.WriteLine("Целое число, прочитанное из файла:");

Console.WriteLine(br.ReadInt32());

}

Console.Read();

}

}

}

**Использование классов Directory, DirectoryInfo, File, FileInfo для файловых операций**

Данные классы предназначены для работы с папками и файлами: *Directory* и *DirectoryInfo* – для работы с папками, а *File* и *FileInfo* – с файлами. Отличие этих классов в том, что классы *Directory* и *File* - статические, а *DirectoryInfo* и *FileInfo* позволяют создавать объекты. Назначение же и тех и других одинаковое.

using System;

using System.IO;

namespace DirectoryFileExample

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Введите полный путь для создания новой папки: ");

Directory.CreateDirectory(Console.ReadLine());

}

}

}

using System;

using System.IO;

namespace DirectoryFileExample

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Введите полный путь для создания новой папки: ");

DirectoryInfo di = new DirectoryInfo(Console.ReadLine());

di.Create();

}

}

}

Назначение основных методов и свойств классов Directory и DirectoryInfo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Directory** | **DirectoryInfo** | **File** | **FileInfo** | **Описание** |
| CreateDirectory | Create | Create | Create | Создает папку или файл |
| Delete | Delete | Delete | Delete | Удаляет папку или файл. Для папок есть перегруженный вари-ант, который принимает дополни-тельный булевый параметр, уста-новка которого в true приводит к удалению папки со всем содер-жимым. |
| Move | MoveTo | Move | MoveTo | Перемещает файл или папку в новое место. Если переместить в то же самое место, но с другим именем – переименовывает. |
| - | - | Copy | CopyTo | Копирует файлы в новое место. У папок данный метод отсутствует. |
| Exists | Exists | Exists | Exists | Существует ли файл (папка). |
| GetParent | Parent | - | Directory | Возвращает DirectoryInfo более высокого уровня. |
| GetLogicalDrives | - | - | - | Возвращает логические диски компьютера. Аналогичен методу Environment.GetLogicalDrives() и DriveInfo.GetDrives(). |
| SetCurrentDirectory | - | - | - | Устанавливает текущую папку для всего приложения в целом. |
| GetDirectories | GetDirectories | - | - | Возвращает массив имен или DirectoryInfo подкаталогов. Имеется перегруженный вариант, который принимает поисковый шаблон в формате DOS (?, \*), позволяющий выбирать только те папки, которые соответствуют заданному шаблону. |
| GetFiles | GetFiles | - | - | Возвращает массив имен или FileInfo принадлежащих папке файлов. Имеется перегруженный вариант, который принимает поисковый шаблон в формате DOS (?, \*), позволяющий выбирать только те файлы, которые соответствуют заданному шаблону. |
| GetFileSystemEntries | GetFileSystemInfos | - | - | Возвращает массив имен или FileSystemInfo, т.е. принадлежащих каталогу файлов и подкаталогов. Имеется перегруженный вариант, который принимает поисковый шаблон в формате DOS (?, \*), позволяющий выбирать только те папки и файлы, которые соответствуют заданному шаблону. |
| CreateDirectory | CreateSubdirectory | - | - | Создает папку в текущей папке (поддиректорию). |
| GetCreationTime, GetCreationTimeUtc, SetCreationTime, SetCreationTimeUtc | CreationTime, CreationTimeUtc | GetCreationTime, GetCreationTimeUtc, SetCreationTime, SetCreationTimeUtc | CreationTime, CreationTimeUtc | Возвращает или устанавливает дату и время создания файла или папки в обычном и международном (UTC) формате. |
| GetLastAccessTime, GetLastAccessTimeUtc, SetLastAccessTime, SetLastAccessTimeUtc | LastAccessTime, LastAccessTimeUtc | GetLastAccessTime, GetLastAccessTimeUtc, SetLastAccessTime, SetLastAccessTimeUtc | LastAccessTime, LastAccessTimeUtc | Возвращает или устанавливает дату и время последнего доступа к файлу или папке в обычном и |
| Name | Name | Name | Name | Получает имя текущего файла или каталога |

Один из способов создания дескриптора файла предусматривает использование метода FileInfo.Create():

static void NewFile()

{

// Создаем новый файл

FileInfo f = new FileInfo(@"C:\Test.dat");

FileStream fs = f.Create();

// Закрыть файловый поток

fs.Close();

}

FileInfo f2 = new FileInfo(@"C:\Test1.dat");

using (FileStream fs2 = f2.Open(FileMode.OpenOrCreate,

FileAccess.ReadWrite, FileShare.None)) {………………………………};

Работа с типом **DirectoryInfo** начинается с указания определенного пути в качестве параметра конструктора. Если требуется получить доступ к текущему рабочему каталогу (т.е. каталогу выполняющегося приложения), применяйте нотацию ".". Вот некоторые примеры:

// Привязаться к текущему рабочему каталогу

DirectoryInfo dir1 = new DirectoryInfo(".");

// Привязаться к C:\Windows

DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");

После создания объекта DirectoryInfo можно исследовать его содержимое, используя любое свойство, унаследованное от FileSystemInfo. Например:

using System;

using System.IO;

namespace ConsoleApplication15\_DirectoryInfo

{

class Program

{

static void Main()

{

ShowWindowsDirectoryInfo();

}

// Выводим информацию о каталоге

static void ShowWindowsDirectoryInfo()

{

DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");

Console.WriteLine("\*\*\*\*\* Информация о каталоге \*\*\*\*\*\n");

Console.WriteLine("Полный путь: {0}\nНазвание папки: {1}\nРодительский каталог: {2}\n" +

"Время создания: {3}\nАтрибуты: {4}\nКорневой каталог: {5}",

dir.FullName,dir.Name,dir.Parent,dir.CreationTime,dir.Attributes,dir.Root);

Console.ReadLine();

}

}

}

Метод **GetFiles()** возвращает массив объектов типа **FileInfo**, каждый из которых представляет детальную информацию о конкретном файле. Предположим, что следующий статический метод класса Program вызывается в методе Main():

static void ImageDisplayFiles()

{

DirectoryInfo dir = new DirectoryInfo("C:\\Windows\\Web\\Wallpaper");

// Получить все файлы с расширением .jpg

FileInfo[] imageFiles = dir.GetFiles("\*.jpg",SearchOption.AllDirectories);

// Сколько файлов найдено

Console.WriteLine("Найдено {0} картинок",imageFiles.Length);

// Вывести информацию о каждом файле

foreach (FileInfo f in imageFiles)

{

Console.WriteLine("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

Console.WriteLine("Имя файла: "+f.Name);

Console.WriteLine("Размер файла: "+f.Length);

Console.WriteLine("Время создания файла: " + f.CreationTime);

}

Console.ReadLine();

}

Статический класс .NET, облегчающий работу с путями – Path.

Назначение методов класса Path.

|  |  |
| --- | --- |
| **Название метода** | **Назначение** |
| Path.ChangeExtension | Позволяет изменить расширение файла в пути. |
| Path.Combine | Позволяет объединить две части пути. Например, путь к папке и имя файла, при этом автоматически определяет наличие или отсутствие разделителя / в конце первой части пути и в начале второй части. Другими словами, метод корректно объединяет две части пути в один вида: @“C:\temp\” и @”\filename.dat”. |
| Path.GetDirectoryName | Вычленяет из полного пути путь к папке (без оконечного разделителя \). |
| Path.GetExtension | Вычленяет из полного пути расширение файла (с лидирующей точкой). |
| Path.GetFileName | Вычленяет из полного пути имя файла с расширением. |
| Path.GetFileNameWithoutExtension | Вычленяет из полного пути имя файла без расширения. |
| Path.GetFullPath | Возвращает полный путь по относительному. |
| Path.GetInvalidFileNameChars | Возвращает массив символов, которые не могут быть использованы в имени файла. |
| Path.GetInvalidPathChars | Возвращает массив символов, которые не могут быть использованы в пути. |
| Path.GetInvalidFileNameChars | Возвращает массив символов, которые не могут быть использованы в имени файла. |
| Path.GetPathRoot | Возвращает корень (диск, с оконечным раздели-телем /) данного пути. |
| Path.GetRandomFileName | Генерирует корректное случайное имя файла (с расширением). |
| Path.GetTempFileName | Генерирует уникальное имя файла во временной папке и возвращает полный путь к нему. |
| Path.GetTempPath | Возвращает полный путь к временной папке. |
| Path.HasExtension | Возвращает булевое значение – есть ли расширение у файла (true) или нет (false). |
| Path.IsPathRooted | Возвращает булевое значение – абсолютный ли путь (true) или нет (false). |

**Пример**

using System;

using System.IO;

class Program

{

static void Main()

{

string path = "C:\\tagelist.txt";

string extension = Path.GetExtension(path);

string filename = Path.GetFileName(path);

string filenameNoExtension = Path.GetFileNameWithoutExtension(path);

string root = Path.GetPathRoot(path);

Console.WriteLine("{0}\n{1}\n{2}\n{3}",

extension,

filename,

filenameNoExtension,

root);

}

}