Система ввода-вывода основана в C# на иерархии классов, поэтому ее функции и особенности нельзя было представлять до тех пор, пока не были рассмотрены классы, наследование и исключения. А теперь настал черед и для ввода-вывода. В C# применяется система ввода-вывода и классы, определенные в среде .NET Framework, и поэтому рассмотрение ввода-вывода в этом языке относится ко всей системе ввода-вывода среды .NET в целом.

**Организация системы ввода- вывода в C# на потоках**

Ввод-вывод в программах на C# осуществляется посредством потоков. *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. Поэтому классы и методы ввода-вывода могут применяться к самым разным типам устройств. Например, методами вывода на консоль можно пользоваться и для вывода в файл на диске.

**Байтовые и символьные потоки**

На самом низком уровне ввод-вывод в С# осуществляется байтами. И делается это

потому, что многие устройства ориентированы на операции ввода-вывода отдельными

байтами. Но человеку больше свойственно общаться символами. Напомним, что в C#

тип char является 16-разрядным, а тип byte — 8-разрядным. Так, если в целях ввода-

вывода используется набор символов в коде ASCII, то для преобразования типа char

в тип byte достаточно отбросить старший байт значения типа char. Но это не годится

для набора символов в уникоде (Unicode), где символы требуется представлять двумя,

а то и больше байтами. Следовательно, байтовые потоки не совсем подходят для организации

ввода-вывода отдельными символами. С целью разрешить это затруднение

в среде .NET Framework определено несколько классов, выполняющих превращение

байтового потока в символьный с автоматическим преобразованием типа byte в тип

char и обратно.

**Встроенные потоки**

Для всех программ, в которых используется пространство имен System, доступны

встроенные потоки, открывающиеся с помощью свойств Console.In, Console.Out

и Console.Error. В частности, свойство Console.Out связано со стандартным потоком

вывода. По умолчанию это поток вывода на консоль. Так, если вызывается метод

Console.WriteLine(), информация автоматически передается свойству Console.

Out. Свойство Console.In связано со стандартным потоком ввода, который по умолчанию

осуществляется с клавиатуры. А свойство 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. Стандартные потоки являются символьными. Поэтому в эти

потоки выводятся и вводятся из них символы.

**Классы потоков**

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. Для того

чтобы воспользоваться этими классами, как правило, достаточно ввести приведенный

ниже оператор в самом начале программы.

***using System.IO;***

Пространство имен System.IO не указывается для консольного ввода-вывода потому,

что для него определен класс Console в пространстве имен System.

# Класс Stream

Основным для потоков является класс System.IO.Stream. Он представляет байтовый

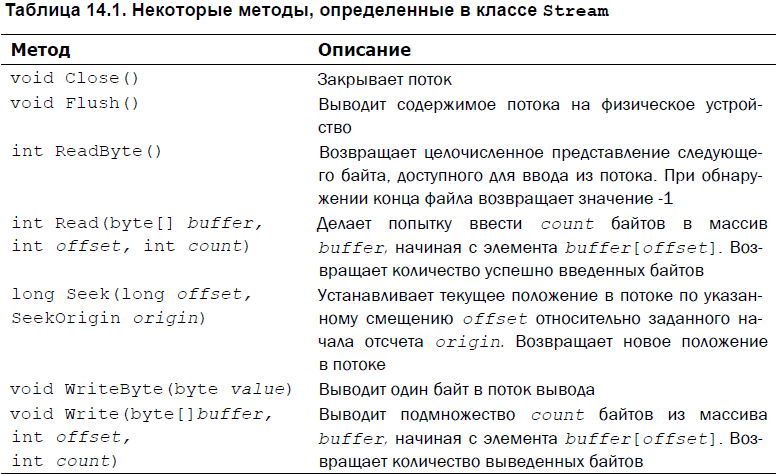
поток и является базовым для всех остальных классов потоков. Кроме того, он

является абстрактным классом, а это означает, что получить экземпляр объекта класса

Stream нельзя. В классе Stream определяется ряд операций со стандартными потоками,

представленных соответствующими методами. В табл. 14.1 перечислен ряд наиболее

часто используемых методов, определенных в классе Stream.



Некоторые из методов, перечисленных в табл. 14.1, генерируют исключение

IOException при появлении ошибки ввода-вывода. Если же предпринимается попытка

выполнить неверную операцию, например вывести данные в поток, предназначенный

только для чтения, то генерируется исключение NotSupportedException.

Кроме того, могут быть сгенерированы и другие исключения — все зависит от конкретного

метода.

Следует заметить, что в классе Stream определены методы для ввода (или чтения)

и вывода (или записи) данных. Но не все потоки поддерживают обе эти операции,

поскольку поток можно открывать только для чтения или только для записи. Кроме

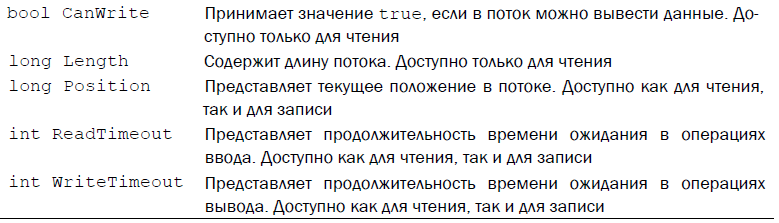
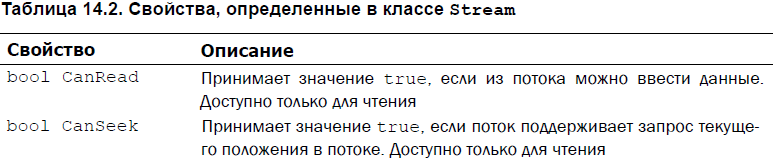
того, не все потоки поддерживают запрос текущего положения в потоке с помощью

метода Seek(). Для того чтобы определить возможности потока, придется воспользоваться

одним, а то и несколькими свойствами класса Stream. Эти свойства перечислены

в табл. 14.2 наряду со свойствами Length и Position, содержащими длину потока

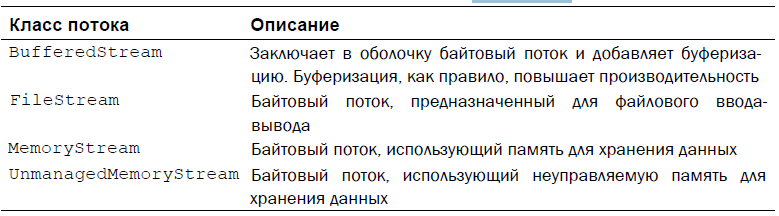
и текущее положение в нем.



**Классы байтовых потоков**

Производными от класса Stream являются несколько конкретных классов байтовых

потоков. Эти классы определены в пространстве имен System.IO и перечислены ниже.



В среде NET Framework поддерживается также ряд других конкретных классов потоков,

в том числе для ввода-вывода в сжатые файлы, сокеты и каналы. Кроме того,

можно создать свои собственные производные классы потоков, хотя для подавляющего

числа приложений достаточно и встроенных потоков.

# Классы-оболочки символьных потоков

Для создания символьного потока достаточно заключить байтовый поток в один

из классов-оболочек символьных потоков. На вершине иерархии классов символьных

потоков находятся абстрактные классы TextReader и TextWriter. Так, класс

TextReader организует ввод, а класс TextWriter — вывод. Методы, определенные

в обоих этих классах, доступны для всех их подклассов. Они образуют минимальный

набор функций ввода-вывода, которыми должны обладать все символьные потоки.

В табл. 14.3 перечислены методы ввода, определенные в классе ***TextReader***. В целом,

эти методы способны генерировать исключение IOException при появлении

ошибки ввода, а некоторые из них — исключения других типов. Особый интерес вызывает

метод ReadLine(), предназначенный для ввода целой текстовой строки, возвращая

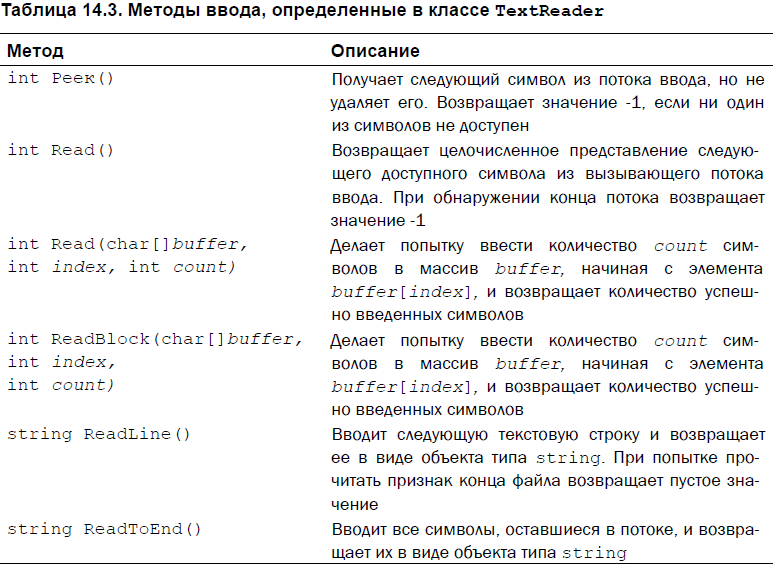
ее в виде объекта типа string. Этот метод удобен для чтения входных данных,

содержащих пробелы. В классе TextReader имеется также метод Close(), определяемый

следующим образом.

***void Close()***

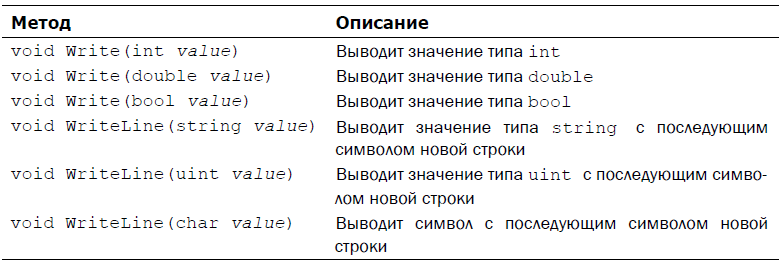
Этот метод закрывает считывающий поток и освобождает его ресурсы.



В классе ***TextWriter*** определены также варианты методов Write() и WriteLine(),

предназначенные для вывода данных всех встроенных типов. Ниже в качестве примера

перечислены лишь некоторые из перегружаемых вариантов этих методов.



Все эти методы генерируют исключение IOException при появлении ошибки

вывода.

Кроме того, в классе TextWriter определены методы Close() и Flush(), приведенные

ниже.

***virtual void Close()***

***virtual void Flush()***

Метод Flush() организует вывод в физическую среду всех данных, оставшихся

в выходном буфере. А метод Close() закрывает записывающий поток и освобождает

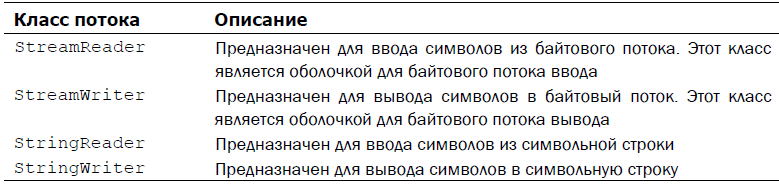
его ресурсы.

Классы TextReader и TextWriter реализуются несколькими классами символьных

потоков, включая и те, что перечислены ниже. Следовательно, в этих классах

потоков предоставляются методы и свойства, определенные в классах TextReader

и TextWriter.



# Консольный ввод-вывод

Следует еще раз подчеркнуть, что большинство реальных приложений

C# ориентированы не на консольный ввод-вывод в текстовом виде, а на графический

оконный интерфейс для взаимодействия с пользователем, или же они представляют

собой программный код, используемый на стороне сервера. Поэтому часть системы

ввода-вывода, связанная с консолью, не находит широкого практического применения.

И хотя программы, ориентированные на текстовый ввод-вывод, отлично подходят в

качестве учебных примеров, коротких сервисных программ или определенного рода

программных компонентов, для большинства реальных приложений они не годятся.

# Чтение данных из потока ввода с консоли

Поток Console.In является экземпляром объекта класса TextReader, и поэтому

для доступа к нему могут быть использованы методы и свойства, определенные в классе TextReader.

В классе Console определены три метода ввода. Два первых метода,

Read() и ReadLine(), были доступны еще в версии .NET Framework 1.0. А третий

метод, ReadKey(), был добавлен в версию 2.0 этой среды.

Для чтения одного символа служит приведенный ниже метод Read().

***static int Read()***

Метод Read() возвращает очередной символ, считанный с консоли. Возвращаемый

символ относится к типу int и поэтому должен быть приведен к типу

char. Если при вводе возникает ошибка, то метод Read() возвращает значение -1.

Этот метод сгенерирует исключение IOException при неудачном исходе операции

ввода. Ввод с консоли с помощью метода Read() буферизуется построчно, поэтому

пользователь должен нажать клавишу <Enter>, прежде чем программа получит любой

символ, введенный с консоли.

(***glava14\_1***)

class KbIn

{

static void Main()

{

char ch;

Console.Write("Press button, and - <ENTER>: ");

ch = (char)Console.Read();//get ch value

Console.WriteLine("You pressed: " + ch);

}

}

Для того чтобы считать в переменную int:

int x, y, z, q;

x = int.Parse(Console.ReadLine());

y = int.Parse(Console.ReadLine());

z = int.Parse(Console.ReadLine());

Необходимость буферизировать построчно ввод, осуществляемый с консоли посредством

метода Read(), иногда может быть досадным препятствием. Ведь при нажатии

клавиши <Enter> в поток ввода передается последовательность символов перевода

каретки и перевода строки. Более того, эти символы остаются во входном буфере до тех

пор, пока они не будут считаны. Следовательно, в некоторых приложениях приходится

удалять эти символы (путем их считывания), прежде чем приступать к следующей операции

ввода. Впрочем, для чтения введенных с клавиатуры символов без построчной

буферизации, можно воспользоваться рассматриваемым далее методом ReadKey().

Для считывания строки символов служит приведенный ниже метод ReadLine().

***static string ReadLine()***

Символы считываются методом ReadLine() до тех пор, пока пользователь не

нажмет клавишу <Enter>, а затем этот метод возвращает введенные символы в виде объекта типа string. Кроме того, он сгенерирует исключение IOException при неудачном исходе операции ввода.

(***glava14\_1.1***)

class ReadString

{

static void Main()

{

string str;

Console.WriteLine("Type few symbols.");

str = Console.ReadLine();

Console.WriteLine("You typed: " + str);

}

}

Итак, для чтения данных из потока Console.In проще всего воспользоваться методами

класса Console. Но для этой цели можно обратиться и к методам базового класса

TextReader. В качестве примера ниже приведен переделанный вариант предыдущего

примера программы, в котором используется метод Rea.dLine(), определенный

в классе TextReader.

(***glava14\_1.2***)

class ReadString2

{

static void Main()

{

string str;

Console.WriteLine("Type few symbols.");

str = Console.In.ReadLine();//call method ReadLine() class TextReadaer

Console.WriteLine("You typed: " + str);

}

}

Обратите внимание на то, что метод ReadLine() теперь вызывается непосредственно

для потока Console.In. Поэтому если требуется доступ к методам, определенным

в классе TextReader, который является базовым для потока Console.In, то подобные

методы вызываются так, как было показано в приведенном выше примере.

# Применение метода ReadKey()

В состав среды .NET Framework включен метод, определяемый в классе Console

и позволяющий непосредственно считывать отдельно введенные с клавиатуры

символы без построчной буферизации. Этот метод называется ReadKey(). При нажа тии клавиши метод ReadKey() немедленно возвращает введенный с клавиатуры символ.

Ниже приведены две формы объявления метода ReadKey().

***static ConsoleKeyInfo ReadKey()***

***static ConsoleKeyInfo ReadKey(bool intercept)***

В первой форме данного метода ожидается нажатие клавиши. Когда оно происходит,

метод возвращает введенный с клавиатуры символ и выводит его на экран.

Во второй форме также ожидается нажатие клавиши, и затем возвращается введенный

с клавиатуры символ. Но если значение параметра *intercept* равно true, то введенный

символ не отображается. А если значение параметра *intercept* равно false,

то введенный символ отображается.

Метод ReadKey() возвращает информацию о нажатии клавиши в объекте типа

ConsoleKeyInfo, который представляет собой структуру, состоящую из приведенных

ниже свойств, доступных только для чтения.

***char KeyChar***

***ConsoleKey Key***

***ConsoleModifiers Modifiers***

Свойство KeyChar содержит эквивалент char введенного с клавиатуры символа,

свойство Key — значение из перечисления ConsoleKey всех клавиш на клавиатуре,

а свойство Modifiers — описание одной из модифицирующих клавиш (<Alt>, <Ctrl>

или <Shift>), которые были нажаты, если это действительно имело место, при формировании

ввода с клавиатуры. Эти модифицирующие клавиши представлены в

перечислении ConsoleModifiers следующими значениями: Control, Shift и Alt.

В свойстве Modifiers может присутствовать несколько значений нажатых модифицирующих

клавиш.

Главное преимущество метода ReadKey() заключается в том, что он предоставляет

средства для организации ввода с клавиатуры в диалоговом режиме, поскольку этот

ввод не буферизуется построчно.

(***glava14\_2***)

class ReadKeys

{

static void Main()

{

ConsoleKeyInfo keypress;

Console.WriteLine("Type few symbols, <Q> for exit.");

do

{

keypress = Console.ReadKey(); //read data pressed keys

Console.WriteLine(" You pressed: " + keypress.KeyChar);

if ((ConsoleModifiers.Alt & keypress.Modifiers) != 0)

Console.WriteLine("You pressed <ALT>");

if ((ConsoleModifiers.Control & keypress.Modifiers) != 0)

Console.WriteLine("You pressed <CTRL>");

if ((ConsoleModifiers.Shift & keypress.Modifiers) != 0)

Console.WriteLine("You pressed <SHIFT>");

} while (keypress.KeyChar != 'Q');

}

}

Как следует из приведенного выше результата, всякий раз, когда нажимается клавиша,

метод ReadKey() немедленно возвращает введенный с клавиатуры символ. Этим

он отличается от упоминавшегося ранее метода Read(), в котором ввод выполняется

с построчной буферизацией. Поэтому если требуется добиться в программе реакции

на ввод с клавиатуры, то рекомендуется выбрать метод ReadKey().

# Запись данных в поток вывода на консоль

Потоки Console.Out и Console.Error являются объектами типа TextWriter.

Вывод на консоль проще всего осуществить с помощью методов Write()

и WriteLine(), с которыми вы уже знакомы. Существуют варианты этих методов для

вывода данных каждого из встроенных типов. В классе Console определяются его собственные

варианты метода Write() и WriteLine(), и поэтому они могут вызываться

непосредственно для класса Console, как это было уже не раз показано на страницах

данной книги. Но при желании эти и другие методы могут быть вызваны и для класса

TextWriter, который является базовым для потоков Console.Out и Console.Error.

(***glava14\_3***)

class ErrOut

{

static void Main()

{

int a = 10, b = 0;

int result;

Console.Out.WriteLine("Divide by zero will folow to exception");

try

{

result = a / b;

}

catch(DivideByZeroException exc)

{

Console.Error.WriteLine(exc.Message);

}

}

}

Начинающие программисты порой испытывают затруднения при использовании

потока Console.Error. Перед ними невольно встает вопрос: если оба потока,

Console.Out и Console.Error, по умолчанию выводят результат на консоль, то зачем

нужны два разных потока вывода? Ответ на этот вопрос заключается в том, что

стандартные потоки могут быть переадресованы на другие устройства. Так, поток

Console.Error можно переадресовать в выходной файл на диске, а не на экран. Это,

например, означает, что сообщения об ошибках могут быть направлены в файл журнала

регистрации, не мешая выводу на консоль. И наоборот, если вывод на консоль переадресуется,

а вывод сообщений об ошибках остается прежним, то на консоли появятся

сообщения об ошибках, а не выводимые на нее данные.

# Класс FileStream и байтовый ввод-вывод в файл

В среде .NET Framework предусмотрены классы для организации ввода-вывода

в файлы. Безусловно, это в основном файлы дискового типа. На уровне операционной

системы файлы имеют байтовую организацию. И, как следовало ожидать, для ввода

и вывода байтов в файлы имеются соответствующие методы. Поэтому ввод и вывод

в файлы байтовыми потоками весьма распространен. Кроме того, байтовый поток ввода

или вывода в файл может быть заключен в соответствующий объект символьного

потока. Операции символьного ввода-вывода в файл находят применение при обработке

текста. О символьных потоках речь пойдет далее в этой главе, а здесь рассматривается

байтовый ввод-вывод.

Для создания байтового потока, привязанного к файлу, служит класс FileStream.

Этот класс является производным от класса Stream и наследует всего его функции.

Напомним, что классы потоков, в том числе и FileStream, определены в пространстве

имен System.IO. Поэтому в самом начале любой использующей их программы

обычно вводится следующая строка кода.

using System.IO;

# Открытие и закрытие файла

Для формирования байтового потока, привязанного к файлу, создается объект

класса FileStream. В этом классе определено несколько конструкторов. Ниже приведен

едва ли не самый распространенный среди них:

***FileStream(string путь, FileMode режим)***

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

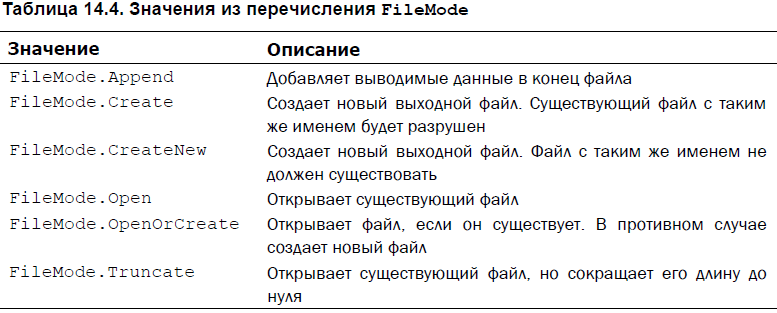
— порядок открытия файла. В последнем случае указывается одно из значений,

определяемых в перечислении FileMode и приведенных в табл. 14.4. Как правило,

этот конструктор открывает файл для доступа с целью чтения или записи. Исключением

из этого правила служит открытие файла в режиме FileMode.Append, когда

файл становится доступным только для записи.



Если попытка открыть файл оказывается неудачной, то генерируется исключение.

Если же файл нельзя открыть из-за того что он не существует, генерируется

исключение FileNotFoundException. А если файл нельзя открыть из-за какой-

нибудь ошибки ввода-вывода, то генерируется исключение IOException. К числу

других исключений, которые могут быть сгенерированы при открытии файла,

относятся следующие: ArgumentNullException (указано пустое имя файла),

ArgumentException (указано неверное имя файла), ArgumentOutOfRangeException

(указан неверный режим), SeaurityException (у пользователя нет прав доступа

к файлу), PathTooLongException (слишком длинное имя файла или путь к нему),

NotSupportedException (в имени файла указано устройство, которое не поддерживается),

а также DirectoryNotFoundException (указан неверный каталог).

Исключения PathTooLongException, DirectoryNotFoundException

и FileNotFoundException относятся к подклассам класса исключений IOException.

Поэтому все они могут быть перехвачены, если перехватывается исключение

IOException.

Ниже в качестве примера приведен один из способов открытия файла test.dat

для ввода.

FileStream fin;

try {

fin = new FileStream("test", FileMode.Open);

}

catch(IOException exc) { // перехватить все исключения, связанные с вводом-выводом

Console.WriteLine(exc.Message);

// Обработать ошибку.

}

catch(Exception exc { // перехватить любое другое исключение.

Console.WriteLine(exc.Message);

// Обработать ошибку, если это возможно.

// Еще раз сгенерировать необрабатываемые исключения.

}

В первом блоке catch из данного примера обрабатываются ошибки, возникающие

в том случае, если файл не найден, путь к нему слишком длинен, каталог не существует,

а также другие ошибки ввода-вывода. Во втором блоке catch, который является

*"*универсальным*"* для всех остальных типов исключений, обрабатываются другие вероятные

ошибки (возможно, даже путем повторного генерирования исключения). Кроме

того, каждую ошибку можно проверять отдельно, уведомляя более подробно о ней

и принимая конкретные меры по ее исправлению.

Как упоминалось выше, конструктор класса FileStream открывает файл, доступный

для чтения или записи. Если же требуется ограничить доступ к файлу только для чтения

или же только для записи, то в таком случае следует использовать такой конструктор.

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

Как и прежде, *путь* обозначает имя открываемого файла, включая и полный путь

к нему, а *режим* — порядок открытия файла. В то же время *доступ* обозначает конкретный

способ доступа к файлу. В последнем случае указывается одно из значений,

определяемых в перечислении FileAccess и приведенных ниже.

FileAccess.Read - FileAccess.Write - FileAccess.ReadWrite

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

При закрытии файла высвобождаются системные ресурсы, распределенные для

этого файла, что дает возможность использовать их для другого файла. Любопытно,

что метод Close() вызывает, в свою очередь, метод Dispose(), который, собственно,

и высвобождает системные ресурсы.

# Чтение байтов из потока файлового ввода-вывода

В классе FileStream определены два метода для чтения байтов из файла:

ReadByte() и Read(). Так, для чтения одного байта из файла используется метод

ReadByte(), общая форма которого приведена ниже.

***int ReadByte()***

Всякий раз, когда этот метод вызывается, из файла считывается один байт, который

затем возвращается в виде целого значения. К числу вероятных исключений, которые

генерируются при этом, относятся NotSupportedException (поток не открыт для

ввода) и ObjectDisposedException (поток закрыт).

Для чтения блока байтов из файла служит метод Read(), общая форма которого

выглядит так.

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

В методе Read() предпринимается попытка считать количество *count* байтов

в массив *array,* начиная с элемента *array*[*offset*]*.* Он возвращает количество байтов,

успешно считанных из файла. Если же возникает ошибка ввода-вывода, то генерируется

исключение IOException.

В приведенном ниже примере программы метод ReadByte() используется для

ввода и отображения содержимого текстового файла, имя которого указывается в качестве

аргумента командной строки. Обратите внимание на то, что в этой программе

проверяется, указано ли имя файла, прежде чем пытаться открыть его.

В некоторых случаях оказывается проще заключить те части программы, где осуществляется

открытие и доступ к файлу, внутрь блока try, вместо того чтобы разделять

обе эти операции. В качестве примера ниже приведен другой, более краткий вариант

написания представленной выше программы ShowFile.

(***glava14\_4***)

class ShowFile

{

static void Main(string[] args)

{

int i;

FileStream fin = null;

if (args.Length != 1)

{

Console.WriteLine("Применение: ShowFile File");

return;

}

// Использовать один блок try для открытия файла и чтения из него

try

{

fin = new FileStream(args[0], FileMode.Open);

// Читать байты до конца файла.

do

{

i = fin.ReadByte();

if (i != -1) Console.Write((char)i);

} while (i != -1);

}

catch (IOException exc)

{

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

}

finally

{

if (fin != null) fin.Close();

}

}

}

Обратите внимание на то, что в данном варианте программы переменная fin

ссылки на объект класса FileStream инициализируется пустым значением. Если

файл удастся открыть в конструкторе класса FileStream, то значение переменной

fin окажется непустым, а иначе — оно так и останется пустым. Это очень важно, поскольку

метод Close() вызывается внутри блока finally только в том случае, если

значение переменной fin оказывается непустым.

Благодаря своей компактности такой подход часто применяется

во многих примерах организации ввода-вывода, приведенных далее в этой книге. Следует,

однако, иметь в виду, что он не пригоден в тех случаях, когда ситуацию, возникающую

в связи с невозможностью открыть файл, нужно обрабатывать отдельно. Так,

если пользователь неправильно введет имя файла, то на экран, возможно, придется

вывести приглашение правильно ввести имя файла, прежде чем входить в блок try,

где осуществляется проверка правильности доступа к файлу.

# Запись в файл

Для записи байта в файл служит метод WriteByte(). Ниже приведена его простейшая

форма.

void WriteByte(byte value)

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

Если базовый поток не открывается для вывода, то генерируется исключение

NotSupportedException. А если поток закрыт, то генерируется исключение

ObjectDisposedException.

Для записи в файл целого массива байтов может быть вызван метод Write(). Ниже

приведена его общая форма.

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

В методе Write() предпринимается попытка записать в файл количество *count*

байтов из массива *array,* начиная с элемента *array*[*offset*]*.* Он возвращает количе ство байтов, успешно записанных в файл. Если во время записи возникает ошибка, то

генерируется исключение IOException. А если базовый поток не открывается для вывода,

то генерируется исключение NotSupportedException. Кроме того, может быть

сгенерирован ряд других исключений.

Вам, вероятно, известно, что при выводе в файл выводимые данные зачастую записываются

на конкретном физическом устройстве не сразу. Вместо этого они буферизуются

на уровне операционной системы до тех пор, пока не накопится достаточный

объем данных, чтобы записать их сразу одним блоком.

Но если данные требуется записать на физическое устройство без предварительного

накопления в буфере, то для этой цели можно вызвать метод Flush.

void Flush()

При неудачном исходе данной операции генерируется исключение IOException.

Если же поток закрыт, то генерируется исключение ObjectDisposedException.

(***glava14\_5***)

class WriteToFile

{

static void Main(string[] args)

{

FileStream fout = null;

try

{

//open file

fout = new FileStream("test.txt", FileMode.CreateNew);

//write ABC

for (char c = 'A'; c <= 'Z'; c++)

{

fout.WriteByte((byte)c);

}

}

catch (IOException exc)

{

Console.WriteLine("IO Error:\n " + exc.Message);

}

finally

{

if (fout != null) fout.Close();

}

}

}

По умолчанию файл сохраняется в проекте в папке Debug.

# Использование класса FileStream для копирования файла

Преимущество байтового ввода-вывода средствами класса FileStream заключается,

в частности, в том, что его можно применить к файлам практически любого типа,

а не только к текстовым файлам. В качестве примера ниже приведена программа, позволяющая

копировать файл любого типа, в том числе исполняемый. Имена исходного

и выходного файлов указываются в командной строке.

(***glava14\_6***)

class CopyFile

{

static void Main(string[] args)

{

int i;

FileStream fout = null;

FileStream fin = null;

if (args.Length != 2)

{

Console.WriteLine("Do: CopyFile From To");

return;

}

try

{

fin = new FileStream(args[0], FileMode.Open);

fout = new FileStream(args[1], FileMode.Create);

do

{

i = fin.ReadByte();

if (i != -1) fout.WriteByte((byte)i);

} while (i != -1);

}

catch (IOException exc)

{

Console.WriteLine("IO Error:\n" + exc.Message);

}

finally

{

if (fin != null) fin.Close();

if (fout != null) fout.Close();

}

}

}

# 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. Так,

если требуется сохранить текст в уникоде, то для этого лучше всего подойдут именно

символьные потоки. В целом, 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.

Не следует, однако, забывать, что на уровне операционной системы файл представляет

собой набор байтов. И применение класса StreamReader или StreamWriter

никак не может этого изменить.

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

Для создания символьного потока вывода достаточно заключить объект класса

Stream, например FileStream, в оболочку класса StreamWriter. В классе

StreamWriter определено несколько конструкторов. Ниже приведен едва ли не самый

распространенный среди них:

FileStream fout;

StreamWriter fstr\_out = new StreamWriter(fout);

где *поток* обозначает имя открытого потока. Этот конструктор генерирует исключение

ArgumentException, если *поток* не открыт для вывода, а также исключение

ArgumentNullException, если *поток* оказывается пустым. После создания объекта

класс StreamWriter выполняет автоматическое преобразование символов в байты.

Ниже приведен простой пример сервисной программы ввода с клавиатуры и вывода

на диск набранных текстовых строк, сохраняемых в файле test.txt. Набираемый

тест вводится до тех пор, пока в нем не встретится строка *"*стоп*"*. Для символьного вывода

в файл в этой программе используется объект класса FileStream, заключенный

в оболочку класса StreamWriter.

(***glava14\_7***)

class KtoD

{

static void Main()

{

string str;

FileStream fout;

//first open IO stream

try

{

fout = new FileStream("test.txt", FileMode.Create);

}

catch (IOException exc)

{

Console.WriteLine("Could't open file:\n" + exc.Message);

return;

}

//get IO stream in StreamWriter class

StreamWriter fstr\_out = new StreamWriter(fout);

try

{

Console.WriteLine("Type in text (<Q> to quit).");

do

{

Console.Write(": ");

str = Console.ReadLine();

if (str != "Q")

{

str = str + "\r\n";//add a new line

fstr\_out.Write(str);

}

} while (str != "Q");

}

catch (IOException exc)

{

Console.WriteLine("IO Error:\n" + exc.Message);

}

finally

{

fstr\_out.Close();

}

}

}

В некоторых случаях файл удобнее открывать средствами самого класса

StreamWriter. Для этого служит один из следующих конструкторов:

***StreamWriter(string путь)***

***StreamWriter(string путь, bool append)***

где *путь* — это имя открываемого файла, включая полный путь к нему. Если во второй

форме этого конструктора значение параметра *append* равно true, то выводимые

данные присоединяются в конец существующего файла. В противном случае эти

данные перезаписывают содержимое указанного файла. Но независимо от формы

конструктора файл создается, если он не существует. При появлении ошибок ввода-

вывода в обоих случаях генерируется исключение IOException. Кроме того, могут

быть сгенерированы и другие исключения.

(***glava14\_7.1***)

class KtoD

{

static void Main()

{

string str;

StreamWriter fout = null;

//first open IO stream

try

{

fout = new StreamWriter("test.txt");

}

catch (IOException exc)

{

Console.WriteLine("Could't open file:\n" + exc.Message);

return;

}

try

{

Console.WriteLine("Type in text (<Q> to quit).");

do

{

Console.Write(": ");

str = Console.ReadLine();

if (str != "Q")

{

str = str + "\r\n";//add a new line

fout.Write(str);

}

} while (str != "Q");

}

catch (IOException exc)

{

Console.WriteLine("IO Error:\n" + exc.Message);

}

finally

{

fout.Close();

}

}

}

# Применение класса StreamReader

Для создания символьного потока ввода достаточно заключить байтовый поток в

оболочку класса StreamReader. В классе StreamReader определено несколько конструкторов.

Ниже приведен наиболее часто используемый конструктор:

StreamReader(Stream *поток*)

где *поток* обозначает имя открытого потока. Этот конструктор генерирует исключение

ArgumentNullException, если *поток* оказывается пустым, а также исключение

ArgumentException, если *поток* не открыт для ввода. После своего создания объект

класса StreamReader выполняет автоматическое преобразование байтов в символы.

По завершении ввода из потока типа StreamReader его нужно закрыть. При этом закрывается

и базовый поток.

В приведенном ниже примере создается простая сервисная программа ввода с диска

и вывода на экран содержимого текстового файла test.txt. Она служит дополнением

к представленной ранее сервисной программе ввода с клавиатуры и вывода на

диск.

(***glava14\_8***)

class DtoS

{

static void Main()

{

FileStream fin;

string s;

try

{

fin = new FileStream("test.txt", FileMode.Open);

}

catch(IOException exc)

{

Console.WriteLine("Error oppen file:\n" + exc.Message);

return;

}

StreamReader fstr\_in = new StreamReader(fin);

try

{

while((s = fstr\_in.ReadLine()) != null)

{

Console.WriteLine(s);

}

}

catch(IOException exc)

{

Console.WriteLine("IO Error:\n" + exc.Message);

}

finally

{

fstr\_in.Close();

}

}

}

Обратите внимание на то, как в этой программе определяется конец файла. Когда

метод ReadLine() возвращает пустую ссылку, это означает, что достигнут конец

файла. Такой способ вполне работоспособен, но в классе StreamReader предоставляется

еще одно средство для обнаружения конца потока — EndOfStream. Это доступное

для чтения свойство имеет логическое значение true, когда достигается конец

потока, в противном случае — логическое значение false. Следовательно, свойство

EndOfStream можно использовать для отслеживания конца файла. В качестве примера

ниже представлен другой способ организации цикла while для чтения из файла.

while(!fstr\_in.EndOfStream) {

s = fstr\_in.ReadLine();

Console.WriteLine(s);

}

Иногда файл проще открыть, используя непосредственно класс StreamReader,

аналогично классу StreamWriter. Для этой цели служит следующий конструктор:

StreamReader(string путь)

где *путь* — это имя открываемого файла, включая полный путь к нему. Указываемый

файл должен существовать. В противном случае генерируется исключение

FileNotFoundException. Если *путь* оказывается пустым, то генерируется исключение

ArgumentNullException. А если *путь* содержит пустую строку, то генерируется

исключение ArgumentException. Кроме того, могут быть сгенерированы исключения

IOException и DirectoryNotFoundException.

# Переадресация стандартных потоков

Как упоминалось ранее, стандартные потоки, например Console.In, могут быть

переадресованы. И чаще всего они переадресовываются в файл. Когда стандартный поток переадресовывается, то вводимые или выводимые данные направляются в новый

поток в обход устройств, используемых по умолчанию. Благодаря переадресации стандартных

потоков в программе может быть организован ввод команд из дискового файла,

создание файлов журнала регистрации и даже чтение входных данных из сетевого

соединения.

Переадресация стандартных потоков достигается двумя способами. Прежде всего,

это делается при выполнении программы из командной строки с помощью операторов

< и >, переадресовывающих потоки Console.In и Console.Out соответственно.

Допустим, что имеется следующая программа.

class Test

{

static void Main()

{

Console.WriteLine("Это тест.");

}

}

Если выполнить эту программу из командной строки

**Test > log**

то символьная строка "Это тест." будет выведена в файл log. Аналогичным образом

переадресуется ввод. Но для переадресации ввода указываемый источник входных данных

должен удовлетворять требованиям программы, иначе она "зависнет".

Операторы < и >, выполняющие переадресацию из командной строки, не являются

составной частью С#, а предоставляются операционной системой. Поэтому если в рабочей

среде поддерживается переадресация ввода-вывода, как, например, в Windows,

то стандартные потоки ввода и вывода можно переадресовать, не внося никаких изменений

в программу. Тем не менее существует другой способ, позволяющий осуществлять

переадресацию стандартных потоков под управлением самой программы. Для

этого служат приведенные ниже методы SetIn(), SetOut() и SetError(), являющиеся

членами класса Console.

static void SetIn(TextReader новый\_поток\_ввода)

static void SetOut(TextWriter новый\_поток\_вывода)

static void SetError(TextWriter новый\_поток\_сообщений\_об\_ошибках)

Таким образом, для переадресации ввода вызывается метод SetIn() с указанием

требуемого потока. С этой целью может быть использован любой поток ввода, при

условии, что он является производным от класса TextReader. А для переадресации

вывода вызывается метод SetOut() с указанием требуемого потока вывода, который

должен быть также производным от класса TextReader. Так, для переадресации вывода

в файл достаточно указать объект класса FileStream, заключенный в оболочку

класса StreamWriter. Соответствующий пример программы приведен ниже.

Короче говоря, весь поток тупо выводит в указанное место(файл), вместо окна например.

(***glava14\_9***)

class Redirect

{

static void Main()

{

StreamWriter log\_out = null;

try

{

log\_out = new StreamWriter("logFile.txt");

//redirect standart output in file logFile.txt

Console.SetOut(log\_out);

Console.WriteLine("This is the begining of registration log");

for (int i = 0; i < 10; i++) Console.WriteLine(i);

Console.WriteLine("This is the end of registration log");

}

catch (IOException exc)

{

Console.WriteLine("IO Error\n:" + exc.Message);

}

finally

{

if (log\_out != null) log\_out.Close();

}

}

}

# Чтение и запись двоичных данных

В приведенных ранее примерах демонстрировались возможности чтения и записи

байтов или символов. Но ведь имеется также возможность (и ею пользуются часто) читать

и записывать другие типы данных. Например, можно создать файл, содержащий

данные типа int, double или short. Для чтения и записи двоичных значений встроенных

в C# 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. Ниже приведен 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. Если же параметр *output* оказывается пустым,

то генерируется исключение ArgumentNullException. А если поток, определяемый

параметром *output,* не был открыт для записи данных, то генерируется исключение

ArgumentException. По завершении вывода в поток типа BinaryWriter его нужно

закрыть. При этом закрывается и базовый поток.

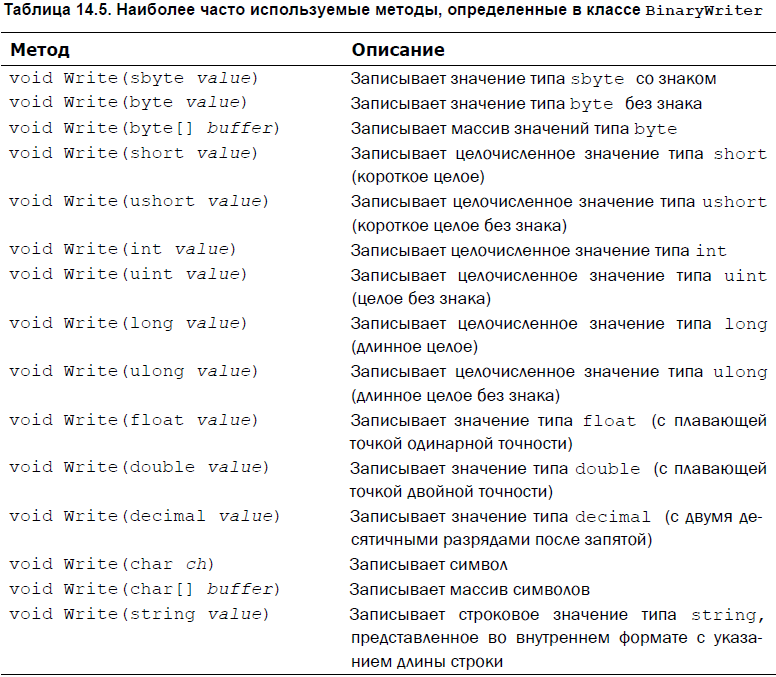
В классе BinaryWriter определены методы, предназначенные для записи данных всех

встроенных в C# типов. Некоторые из этих методов перечислены в табл. 14.5. Обратите

внимание на то, что строковые данные типа string записываются во внутреннем формате

с указанием длины строки. Кроме того, в классе BinaryWriter определены стандартные

методы Close() и Flush(), действующие аналогично описанному выше.



# Класс BinaryReader

Класс BinaryReader служит оболочкой, в которую заключается байтовый поток,

управляющий вводом двоичных данных. Ниже приведен 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. Если же поток, определяемый параметром

*input,* не был открыт для чтения данных или оказался недоступным по иным причинам,

то генерируется исключение ArgumentException. По завершении ввода из

потока типа BinaryReader его нужно закрыть. При этом закрывается и базовый

поток.

В классе BinaryReader определены методы, предназначенные для чтения данных

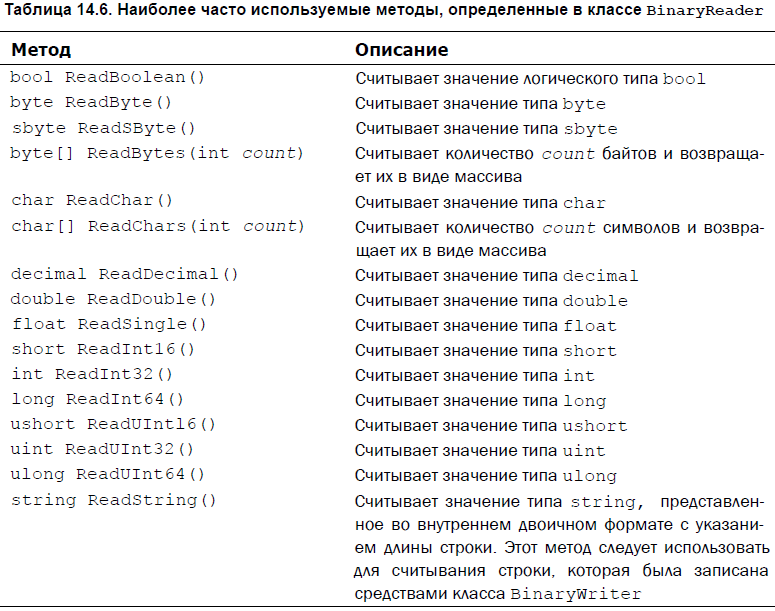
всех встроенных в C# типов. Некоторые из этих методов перечислены в табл. 14.6.

Следует, однако, иметь в виду, что в методе Readstring() считывается символьная

строка, хранящаяся во внутреннем формате с указанием ее длины. Все методы данного

класса генерируют исключение IOException, если возникает ошибка ввода. Кроме

того, могут быть сгенерированы и другие исключения.



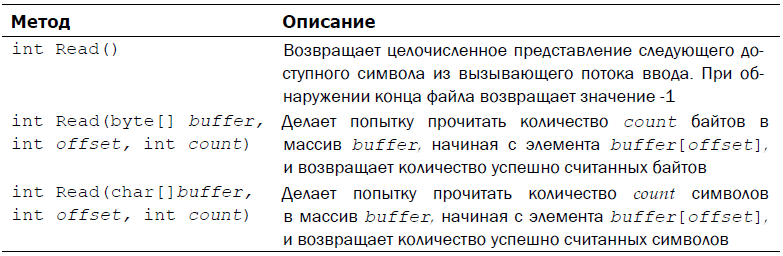
***В классе BinaryWriter определены также три приведенных ниже варианта метода***

Read().

При неудачном исходе операции чтения эти методы генерируют исключение

IOException. Кроме того, в классе BinaryReader определен стандартный метод

Close().



# Демонстрирование двоичного ввода-вывода

Ниже приведен пример программы, в котором демонстрируется применение классов

BinaryReader и BinaryWriter для двоичного ввода-вывода. В этой программе

в файл записываются и считываются обратно данные самых разных типов.

(***glava14\_10***)

class RWData

{

static void Main()

{

BinaryWriter dataOut;

BinaryReader dataIn;

int i = 10;

double d = 1023.56;

bool b = true;

string str = "Test";

//open dataOut file

try

{

dataOut = new BinaryWriter(

new FileStream("testData", FileMode.Create));

}

catch(IOException exc)

{

Console.WriteLine("Read file Error:\n" + exc.Message);

return;

}

//write data in file

try

{

Console.WriteLine("Write " + i);

dataOut.Write(i);

Console.WriteLine("Write " + d);

dataOut.Write(d);

Console.WriteLine("Write " + b);

dataOut.Write(b);

Console.WriteLine("Write " + 12.2 \* 7.4);

dataOut.Write(12.2 \* 7.4);

Console.WriteLine("Write " + str);

dataOut.Write(str);

}

catch(IOException exc)

{

Console.WriteLine("IO Error:\n" + exc.Message);

}

finally

{

dataOut.Close();

}

//Now read data from file

try

{

dataIn = new BinaryReader(new FileStream(

"testData", FileMode.Open));

}

catch(IOException exc)

{

Console.WriteLine("Open file Error:\n" + exc.Message);

return;

}

try

{

i = dataIn.ReadInt32();

Console.WriteLine("Reading " + i);

d = dataIn.ReadDouble();

Console.WriteLine("Reading " + d);

b = dataIn.ReadBoolean();

Console.WriteLine("Reading " + b);

str = dataIn.ReadString();

Console.WriteLine("Reading " + str);

}

catch(IOException exc)

{

Console.WriteLine("IO Error:\n" + exc.Message);

}

finally

{

dataIn.Close();

}

}

}

Если просмотреть содержимое файла testdata, который получается при выполнении

этой программы, то можно обнаружить, что он содержит данные в двоичной,

а не в удобочитаемой текстовой форме.

Далее следует более практический пример, демонстрирующий, насколько эффективным

может быть двоичный ввод-вывод. Для учета каждого предмета хранения на

складе в приведенной ниже программе сначала запоминается наименование предмета,

имеющееся в наличии, количество и стоимость, а затем пользователю предлагается

ввести наименование предмета, чтобы найти его в базе данных. Если предмет найден,

отображаются сведения о его запасах на складе.

(***glava14\_11***)

class Inventory

{

static void Main()

{

BinaryWriter dataOut;

BinaryReader dataIn;

string item; //goods name

int onhand; //qntty

double cost; //price

try

{

dataOut = new BinaryWriter(

new FileStream("inventory.dat", FileMode.Create));

}

catch (IOException exc)

{

Console.WriteLine("Could not open file of stock.");

Console.WriteLine("Reason: " + exc.Message);

return;

}

//writing data about goods in file

try

{

dataOut.Write("Hammers");

dataOut.Write(10);

dataOut.Write(3.95);

dataOut.Write("Screw Drivers");

dataOut.Write(18);

dataOut.Write(1.50);

dataOut.Write("Pliers");

dataOut.Write(5);

dataOut.Write(4.95);

dataOut.Write("Saws");

dataOut.Write(8);

dataOut.Write(8.95);

}

catch (IOException exc)

{

Console.WriteLine("Error writing stock data.");

Console.WriteLine("Reason: " + exc.Message);

}

finally

{

dataOut.Close();

}

//open file of stock data

try

{

dataIn = new BinaryReader(

new FileStream("inventory.dat", FileMode.Open));

}

catch (IOException exc)

{

Console.WriteLine("Could not open file of stock.");

Console.WriteLine("Reason: " + exc.Message);

return;

}

//find goods typed by user

Console.Write("Type name of stuff: ");

string what = Console.ReadLine();

Console.WriteLine();

try

{

for (; ; )

{

//read data of goods

item = dataIn.ReadString();

onhand = dataIn.ReadInt32();

cost = dataIn.ReadDouble();

//check if matches, show if matches

if (item.Equals(what, StringComparison.OrdinalIgnoreCase))

{

Console.WriteLine(item + ": " + onhand

+ " quantity at stock. "

+ "Price: {0:C} for one", cost);

Console.WriteLine("Common price <{0}>: {1:C}.", item, cost \* onhand);

break;

}

}

}

catch(EndOfStreamException)

{

Console.WriteLine("Didnt find any matches.");

}

catch (IOException exc)

{

Console.WriteLine("Read file Error.");

Console.WriteLine("Reason: " + exc.Message);

}

finally

{

dataIn.Close();

}

}

}

Обратите внимание на то, что сведения о товарных запасах сохраняются в этой программе

в двоичном формате, а не в удобной для чтения текстовой форме. Благодаря

этому обработка числовых данных может выполняться без предварительного их преобразования

из текстовой формы.

Обратите также внимание на то, как в этой программе обнаруживается конец файла.

Методы двоичного ввода генерируют исключение EndOfStreamException по достижении

конца потока, и поэтому файл читается до тех пор, пока не будет найден

искомый предмет или сгенерировано данное исключение. Таким образом, для обнаружения

конца файла никакого специального механизма не требуется.

# Файлы с произвольным доступом

В предыдущих примерах использовались *последовательные файлы,* т.е. файлы со

строго линейным доступом, байт за байтом. Но доступ к содержимому файла может

быть и произвольным. Для этого служит, в частности, метод Seek(), определенный

в классе FileStream. Этот метод позволяет установить *указатель положения в файле,*

или так называемый *указатель файла,* на любое место в файле. Ниже приведена общая

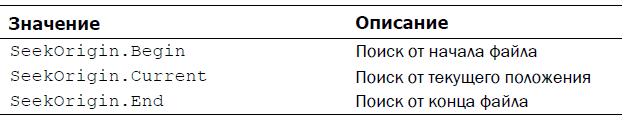
форма метода Seek():

long Seek(long offset, SeekOrigin origin)

где *offset* обозначает новое положение указателя файла в байтах относительно заданного

начала отсчета (*origin*). В качестве *origin* может быть указано одно из приведенных

ниже значений, определяемых в перечислении SeekOrigin.



Следующая операция чтения или записи после вызова метода Seek() будет выполняться,

начиная с нового положения в файле, возвращаемого этим методом. Если во

время поиска в файле возникает ошибка, то генерируется исключение IOException.

Если же запрос положения в файле не поддерживается базовым потоком, то генерируется

исключение NotSupportedException. Кроме того, могут быть сгенерированы

и другие исключения.

В приведенном ниже примере программы демонстрируется ввод-вывод в файл с

произвольным доступом. Сначала в файл записываются прописные буквы английского

алфавита, а затем его содержимое считывается обратно в произвольном порядке.

(***glava14\_12***)

class RandomAccessDemo

{

static void Main()

{

FileStream f = null;

char ch;

try

{

f = new FileStream("random.dat", FileMode.Create);

//English ABC

for (int i = 0; i < 26; i++)

f.WriteByte((byte)('A' + i));

//now read specific letters

f.Seek(0, SeekOrigin.Begin); //find first byte

ch = (char)f.ReadByte();

Console.WriteLine("First letter: " + ch);

f.Seek(1, SeekOrigin.Begin); //find second byte

ch = (char)f.ReadByte();

Console.WriteLine("Second letter: " + ch);

f.Seek(4, SeekOrigin.Begin); //find fifth byte

ch = (char)f.ReadByte();

Console.WriteLine("Fifth letter: " + ch);

Console.WriteLine();

//now read letter by two

Console.WriteLine("Letters missing one: ");

for (int i = 0; i < 26; i += 2)

{

f.Seek(i, SeekOrigin.Begin); //find i symbol

ch = (char)f.ReadByte();

Console.Write(ch + " ");

}

}

catch (IOException exc)

{

Console.WriteLine("IO Error\n:" + exc.Message);

}

finally

{

if (f != null) f.Close();

}

Console.WriteLine();

}

}

Несмотря на то что метод Seek() имеет немало преимуществ при использовании

с файлами, существует и другой способ установки текущего положения в файле с помощью

свойства Position. Как следует из табл. 14.2, свойство Position доступно

как для чтения, так и для записи. Поэтому с его помощью можно получить или же

установить текущее положение в файле. В качестве примера ниже приведен фрагмент

кода из предыдущей программы записи и чтения из файла с произвольным доступом

random.dat, измененный с целью продемонстрировать применение свойства

Position.

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 + " ");

}

# 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. Он представляет

собой реализацию класса Stream, в которой массив байтов используется для ввода

и вывода. В классе MemoryStream определено несколько конструкторов. Ниже представлен

один из них:

MemoryStream(byte[] buffer)

где *buffer* обозначает массив байтов, используемый в качестве источника или адресата

в запросах ввода-вывода. Используя этот конструктор, следует иметь в виду,

что массив *buffer* должен быть достаточно большим для хранения направляемых

в него данных.

(***glava14\_13***)

class MemStrDemo

{

static void Main()

{

byte[] storage = new byte[255];

//create memory stream

MemoryStream memstrm = new MemoryStream(storage);

//envelope object memstrm in classes

//of read and write

StreamWriter memwtr = new StreamWriter(memstrm);

StreamReader memrdr = new StreamReader(memstrm);

try

{

//write data in memory to memwtr

for (int i = 0; i < 10; i++)

memwtr.WriteLine("byte [" + i + "]: " + i);

//write in the end Dot

memwtr.WriteLine(".");

memwtr.Flush();

Console.WriteLine("Read from storage array: ");

//show containings of storage

foreach(char ch in storage)

{

if (ch == '.') break;

Console.Write(ch);

}

Console.WriteLine("\nRead from stream with memrdr: ");

//read from memstrm object with tools of input stream

memstrm.Seek(0, SeekOrigin.Begin);//set pointer at the begining

string str = memrdr.ReadLine();

while(str != null)

{

str = memrdr.ReadLine();

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

Console.WriteLine(str);

}

}

catch(IOException exc)

{

Console.WriteLine("IO Error\n" + exc.Message);

}

finally

{

//clear resourses of read and write streams

memwtr.Close();

memrdr.Close();

}

}

}

В этой программе сначала создается массив байтов, называемый storage. Затем

этот массив используется в качестве основной памяти для объекта memstrm класса

MemoryStream. Из объекта memstrm, в свою очередь, создаются объекты memrdr класса

StreamReader и memwtr класса StreamWriter. С помощью объекта memwtr выводимые

данные записываются в запоминающий поток. Обратите внимание на то, что

после записи выводимых данных для объекта memwtr вызывается метод Flush(). Это

необходимо для того, чтобы содержимое буфера этого объекта записывалось непосредственно

в базовый массив. Далее содержимое базового массива байтов отображается

вручную в цикле foreach. После этого указатель файла устанавливается с помощью

метода Seek() в начало запоминающего потока, из которого затем вводятся

данные с помощью объекта потока memrdr.

Запоминающие потоки очень полезны для программирования. С их помощью

можно, например, организовать сложный вывод с предварительным накоплением

данных в массиве до тех пор, пока они не понадобятся. Этот прием особенно полезен

для программирования в такой среде с графическим пользовательским интерфейсом,

как Windows. Кроме того, стандартный поток может быть переадресован

из массива. Это может пригодиться, например, для подачи тестовой информации в

программу.

# 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. Следовательно, они представляют

собой потоки, имеющие доступ к методам, определенным в этих двух базовых

классах, что позволяет, например, вызывать метод ReadLine() для объекта класса

StringReader, а метод WriteLine() — для объекта класса StringWriter.

Ниже приведен конструктор класса StringReader:

StringReader(string s)

где s обозначает символьную строку, из которой производится чтение.

В классе StringWriter определено несколько конструкторов. Ниже представлен

один из наиболее часто используемых.

StringWriter()

Этот конструктор создает записывающий поток, который помещает выводимые

данные в строку. Для получения содержимого этой строки достаточно вызвать метод

ToString().

(***glava14\_14***)

class StrRdrWtrDemo

{

static void Main()

{

StringWriter strwtr = null;

StringReader strrdr = null;

try

{

//create object class StringWriter

strwtr = new StringWriter();

//get data into stream StringWriter

for (int i = 0; i < 10; i++)

strwtr.WriteLine("Value of i: " + i);

//create object of class StringReader

strrdr = new StringReader(strwtr.ToString()); //!!!!!!!!!

//now out data from read stream StringReader

string str = strrdr.ReadLine();

while(str != null)

{

str = strrdr.ReadLine();

Console.WriteLine(str);

}

}

catch(IOException exc)

{

Console.WriteLine("IO Error\n" + exc.Message);

}

finally

{

if (strrdr != null) strrdr.Close();

if (strwtr != null) strwtr.Close();

}

}

}

В данном примере сначала создается объект strwtr класса StringWriter, в который

выводятся данные с помощью метода WriteLine(). Затем создается объект класса

StringReader с использованием символьной строки, содержащейся в объекте strwtr.

Эта строка получается в результате вызова метода ToString() для объекта strwtr.

И наконец, содержимое данной строки считывается с помощью метода ReadLine().

# Класс File

В среде .NET Framework определен класс File, который может оказаться полезным

для работы с файлами, поскольку он содержит несколько статических методов, выполняющих

типичные операции над файлами. В частности, в классе File имеются методы

для копирования и перемещения, шифрования и расшифровывания, удаления файлов,

а также для получения и задания информации о файлах, включая сведения об их

существовании, времени создания, последнего доступа и различные атрибуты файлов

(только для чтения, скрытых и пр.). Кроме того, в классе File имеется ряд удобных методов

для чтения из файлов и записи в них, открытия файла и получения ссылки типа

FileStream на него. В классе File содержится слишком много методов для подробного

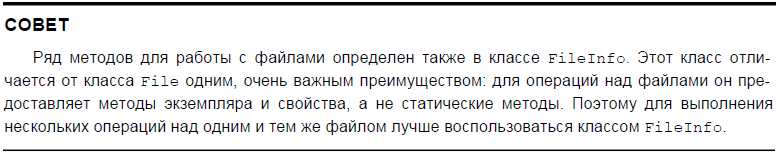
их рассмотрения, поэтому мы уделим внимание только трем из них. Сначала будет

представлен метод Сору(), а затем — методы Exists() и GetLastAccessTime().

На примере этих методов вы сможете получить ясное представление о том, насколько

удобны методы, доступные в классе File. И тогда вам станет ясно, что класс File

определенно заслуживает более тщательного изучения.



# Копирование файлов с помощью метода Сору()

Ранее в этой главе демонстрировался пример программы, в которой файл копировался

вручную путем чтения байтов из одного файла и записи в другой. И хотя задача

копирования файлов не представляет особых трудностей, ее можно полностью автоматизировать

с помощью метода Сору(), определенного в классе File. Ниже представлены

две формы его объявления.

static void Copy(string имя\_исходного\_файла, string имя\_целевого\_файла)

static void Copy(string имя\_исходного\_файла, string имя\_целевого\_файла,

boolean overwrite)

Метод Copy() копирует файл, на который указывает *имя\_исходного\_файла,*

в файл, на который указывает *имя\_целевого\_файла.* В первой форме данный метод

копирует файл только в том случае, если файл, на который указывает *имя\_целево-*

*го\_файла,* еще не существует. А во второй форме копия заменяет и перезаписывает

целевой файл, если он существует и если параметр *overwrite* принимает логическое

значение true. Но в обоих случаях может быть сгенерировано несколько видов исключений,

включая IOException и FileNotFoundException.

В приведенном ниже примере программы метод Сору() применяется для копирования

файла. Имена исходного и целевого файлов указываются в командной строке.

Обратите внимание, насколько эта программа короче демонстрировавшейся ранее.

Кроме того, она более эффективна.

(***glava14\_15***)

class CopyFile

{

static void Main(string[] args)

{

if(args.Length != 2)

{

Console.WriteLine("Do: glava14\_15\_classFile <from> <to>");

return;

}

//copy files

try

{

File.Copy(args[0], args[1]);

}

catch(IOException exc)

{

Console.WriteLine("Copy file Error\n" + exc.Message);

}

}

}

Как видите, в этой программе не нужно создавать поток типа FileStream или

освобождать его ресурсы. Все это делается в методе Сору() автоматически. Обратите

также внимание на то, что в данной программе существующий файл не перезаписывается.

Поэтому если целевой файл должен быть перезаписан, то для этой цели лучше

воспользоваться второй из упоминавшихся ранее форм метода Сору().

# Применение методов Exists() и GetLastAccessTime()

С помощью методов класса File очень легко получить нужные сведения о файле.

Рассмотрим два таких метода: Exists() и GetLastAccessTime(). Метод Exists()

определяет, существует ли файл, а метод GetLastAccessTime() возвращает дату и время

последнего доступа к файлу. Ниже приведены формы объявления обоих методов.

static bool Exists(string путь)

static DateTime GetLastAccessTime(string путь)

В обоих методах *путь* обозначает файл, сведения о котором требуется получить.

Метод Exists() возвращает логическое значение true, если файл существует и доступен

для вызывающего процесса. А метод GetLastAccessTime() возвращает структуру

DateTime, содержащую дату и время последнего доступа к файлу.

С указанием недействительных аргументов

или прав доступа при вызове обоих рассматриваемых здесь методов может быть

связан целый ряд исключений, но в действительности генерируется только исключение

IOException.

В приведенном ниже примере программы методы Exists() и GetLastAccessTime()

демонстрируются в действии. В этой программе сначала определяется, существует ли

файл под названием test.txt. Если он существует, то на экран выводит время последнего

доступа к нему.

(***glava14\_15.1***)

class ExistsDemo

{

static void Main()

{

if (File.Exists("test.txt"))

Console.WriteLine("File exists. Last time he was used " +

File.GetLastAccessTime("test.txt"));

else

Console.WriteLine("File not exists.");

}

}

Кроме того, время создания файла можно выяснить, вызвав метод GetCreationTime(),

а время последней записи в файл, вызвав метод GetLastWriteTime(). Имеются также

варианты этих методов для представления данных о файле в формате всеобщего

скоординированного времени (UTC). Попробуйте поэкспериментировать с ними.

# Преобразование числовых строк в их внутреннее представление

Как вам должно быть уже известно, метод WriteLine() предоставляет удобные средства для вывода различных типов данных на консоль, включая и числовые значения встроенных типов, например

int или double. При этом числовые значения автоматически преобразуются

методом WriteLine() в удобную для чтения текстовую форму. В то же время аналогичный

метод ввода для чтения и преобразования строк с числовыми значениями

в двоичный формат их внутреннего представления не предоставляется. В частности,

отсутствует вариант метода Read() специально для чтения строки "100", введенной с

клавиатуры, и автоматического ее преобразования в соответствующее двоичное значение,

которое может быть затем сохранено в переменной типа int. Поэтому данную

задачу приходится решать другими способами. И самый простой из них — воспользоваться

методом Parse(), определенным для всех встроенных числовых типов данных.

Прежде всего необходимо отметить следующий важный факт: все встроенные в C#

типы данных, например int или double, на самом деле являются не более чем *псевдонимами*

(т.е. другими именами) структур, определяемых в среде .NET Framework.

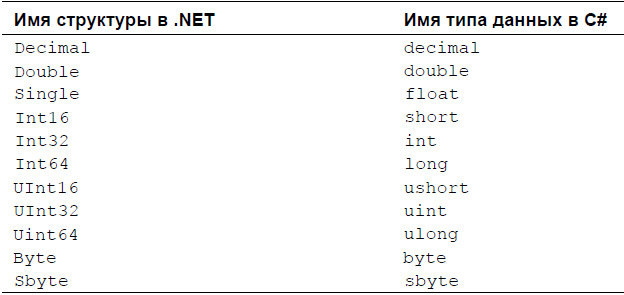
В действительности тип в C# невозможно отличить от типа структуры в среде .NET

Framework, поскольку один просто носит имя другого. В C# для поддержки значений простых типов используются структуры, и поэтому для типов этих значений имеются

специально определенные члены структур.

Ниже приведены имена структур .NET и их эквиваленты в виде ключевых слов C#

для числовых типов данных.



Эти структуры определены в пространстве имен System. Следовательно, имя структуры

Int32 полностью определяется как System.Int32. Эти структуры предоставляют

обширный ряд методов, помогающих полностью интегрировать значения простых

типов в иерархию объектов С#. А кроме того, в числовых структурах определяется статический

метод Parse(), преобразующий числовую строку в соответствующий двоичный

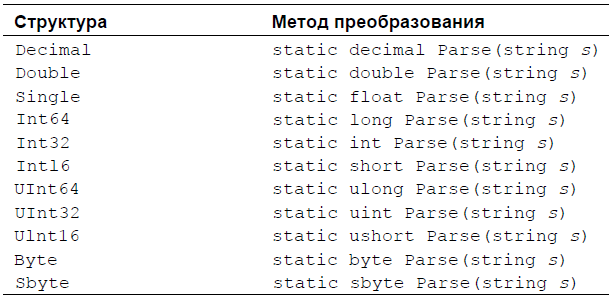
эквивалент.

Существует несколько перегружаемых форм метода Parse(). Ниже приведены его

простейшие варианты для каждой числовой структуры. Они выполняют преобразование

с учетом местной специфики представления чисел. Следует иметь в виду, что каждый

метод возвращает двоичное значение, соответствующее преобразуемой строке.



Приведенные выше варианты метода Parse() генерируют исключение

FormatException, если строка s не содержит допустимое число, определяемое вызывающим

типом данных. А если она содержит пустое значение, то генерируется исключение

ArgumentNullException. Когда же значение в строке s превышает допустимый

диапазон чисел для вызывающего типа данных, то генерируется исключение

OverflowException.

Методы синтаксического анализа позволяют без особого труда преобразовать числовое

значение, введенное с клавиатуры или же считанное из текстового файла в виде

строки, в соответствующий внутренний формат. В качестве примера ниже приведена

программа, в которой усредняется ряд чисел, вводимых пользователем. Сначала пользователю

предлагается указать количество усредняемых значений, а затем это количество

считывается методом ReadLine() и преобразуется из строки в целое число методом

Int32.Parse(). Далее вводятся отдельные значения, преобразуемые методом

Double.Parse() из строки в их эквивалент типа double.

(***glava14\_16***)

class AvgNums

{

static void Main()

{

string str;

int n;

double sum = 0.0;

double avg, t;

Console.Write("How many nums are you gona type?: ");

str = Console.ReadLine();

try

{

n = Int32.Parse(str);

}

catch (FormatException exc)

{

Console.WriteLine(exc.Message);

return;

}

catch (OverflowException exc)

{

Console.WriteLine(exc.Message);

return;

}

Console.WriteLine("Enter " + n + " nums.");

for (int i = 0; i < n; i++)

{

Console.Write(": ");

str = Console.ReadLine();

try

{

t = Double.Parse(str);

}

catch (FormatException exc)

{

Console.WriteLine(exc.Message);

t = 0.0;

}

catch (OverflowException exc)

{

Console.WriteLine(exc.Message);

t = 0;

}

sum += t;

}

avg = sum / n;

Console.WriteLine("Average: " + avg);

}

}

Следует особо подчеркнуть, что для каждого преобразуемого значения необходимо

выбирать подходящий метод синтаксического анализа. Так, если попытаться

преобразовать строку, содержащую значение с плавающей точкой, методом Int32.

Parse(), то искомый результат, т.е. числовое значение с плавающей точкой, получить

не удастся.

Как пояснялось выше, при неудачном исходе преобразования метод Parse() сгенерирует

исключение. Для того чтобы избежать генерирования исключений при преобразовании

числовых строк, можно воспользоваться методом TryParse(), определенным

для всех числовых структур. В качестве примера ниже приведен один из вариантов

метода TryParse(), определяемых в структуре Int32:

static bool TryParse(string s, out int результат)

где *s* обозначает числовую строку, передаваемую данному методу, который возвращает

соответствующий *результат* после преобразования с учетом выбираемой по

умолчанию местной специфики представления чисел. (Конкретную местную специфику

представления чисел с учетом региональных стандартов можно указать в другом

варианте данного метода.) При неудачном исходе преобразования, например, когда

параметр *s* не содержит числовую строку в надлежащей форме, метод TryParse()

возвращает логическое значение false. В противном случае он возвращает логическое

значение true. Следовательно, значение, возвращаемое этим методом, обязательно

следует проверить, чтобы убедиться в удачном (или неудачном) исходе преобразования.