```
    try {
      result = a / b; // сгенерировать исключение при попытке деления на нуль
    } catch(DivideByZeroException exc) {
       Console.Error.WriteLine(exc.Message);
    }
  }
}
```

При выполнении этой программы получается следующий результат.

```
Деление на нуль приведет к исключительной ситуации.
Попытка деления на нуль.
```

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. Мы еще вернемся к вопросу переадресации после рассмотрения файлового ввода-вывода.

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.

Таблица 14.4. Значения из перечисления `FileMode`

Значение	Описание
`FileMode.Append`	Добавляет выводимые данные в конец файла
`FileMode.Create`	Создает новый выходной файл. Существующий файл с таким же именем будет разрушен
`FileMode.CreateNew`	Создает новый выходной файл. Файл с таким же именем не должен существовать
`FileMode.Open`	Открывает существующий файл
`FileMode.OpenOrCreate`	Открывает файл, если он существует. В противном случае создает новый файл
`FileMode.Truncate`	Открывает существующий файл, но сокращает его длину до нуля

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.

Ради простоты в примерах, представленных в этой книге, перехватывается только исключение `IOException`, но в реальной программе, скорее всего, потребуется перехватывать и другие вероятные исключения, связанные с вводом-выводом, в зависимости от обстоятельств. Кроме того, в обработчиках исключений, приводимых в качестве примера в этой главе, просто уведомляется об ошибке, но зачастую в них должны быть запрограммированы конкретные меры по исправлению ошибок, если это вообще возможно. Например, можно предложить пользователю еще раз ввести имя файла, если указанный ранее файл не был найден. Возможно, также потребуется сгенерировать исключение повторно.

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
```

Например, в следующем примере кода файл `test.dat` открывается только для чтения.

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

По завершении работы с файлом его следует закрыть, вызвав метод `Close()`. Ниже приведена общая форма обращения к этому методу.

```
void 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.

ПРИМЕЧАНИЕ

Оператор `using`, рассматриваемый в главе 20, предоставляет еще один способ закрытия файла, который больше не нужен. Такой способ оказывается удобным во многих случаях обращения с файлами, поскольку гарантирует закрытие ненужного больше файла простыми средствами. Но исключительно в целях демонстрации основ обращения с файлами, в том числе и того момента, когда файл может быть закрыт, во всех примерах, представленных в этой главе, используются явные вызовы метода `Close()`.

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`. К числу других вероятных исключений, которые генерируются при этом, относится `NotSupportedException`. Это исключение генерируется в том случае, если чтение из файла не поддерживается в потоке.

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.

```
/* Отобразить содержимое текстового файла.

  Чтобы воспользоваться этой программой, укажите имя того файла,
  содержимое которого требуется отобразить. Например, для просмотра
  содержимого файла TEST.CS введите в командной строке следующее:

  ShowFile TEST.CS
*/

using System;
using System.IO;

class ShowFile {
  static void Main(string[] args) {
    int i;
    FileStream fin;

    if(args.Length != 1) {
    Console.WriteLine("Применение: ShowFile Файл");
    return;
    }

    try {
      fin = new FileStream(args[0], FileMode.Open);
    } catch(IOException exc) {
      Console.WriteLine("Не удается открыть файл");
      Console.WriteLine(exc.Message);
      return; // Файл не открывается, завершить программу
```

```
    }

    // Читать байты до конца файла.
      try {

        do {
          i = fin.ReadByte();
          if(i != -1) Console.Write((char) i);
        } while(i != -1);

      } catch(IOException exc) {
        Console.WriteLine("Ошибка чтения файла");
        Console.WriteLine(exc.Message);
      } finally {
        fin.Close();
      }
  }
}
```

Обратите внимание на то, что в приведенной выше программе применяются два блока `try`. В первом из них перехватываются исключения, возникающие при вводе-выводе и способные воспрепятствовать открытию файла. Если произойдет ошибка ввода-вывода, выполнение программы завершится. В противном случае во втором блоке `try` будет продолжен контроль исключений, возникающих в операциях ввода-вывода. Следовательно, второй блок `try` выполняется только в том случае, если в переменной `fin` содержится ссылка на открытый файл. Обратите также внимание на то, что файл закрывается в блоке `finally`, связанном со вторым блоком `try`. Это означает, что независимо от того, как завершится цикл `do-while` (нормально или аварийно из-за ошибки), файл все равно будет закрыт. И хотя в данном конкретном примере это и так важно, поскольку программа все равно завершится в данной точке, преимущество такого подхода, вообще говоря, заключается в том, что файл закрывается в завершающем блоке `finally` в *любом* случае — даже если выполнение кода доступа к этому файлу завершается преждевременно из-за какого-нибудь исключения.

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`.

```
// Отобразить содержимое текстового файла.

using System;
using System.IO;

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. Подобный механизм препятствует любой попытке вызвать метод `Close()` для переменной `fin`, когда она не ссылается на открытый файл. 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.

В целом, порядок открытия, доступа и закрытия файла зависит от конкретного приложения. То, что хорошо в одном случае, может оказаться неприемлемым в другом. Поэтому данный процесс приходится приспосабливать к конкретным потребностям разрабатываемой программы.

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. Благодаря этому повышается эффективность системы. Так, на диске файлы организованы по секторам величиной от 128 байтов и более. Поэтому выводимые данные обычно буферизуются до тех пор, пока не появится возможность записать на диск сразу весь сектор.

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`.

По завершении вывода в файл следует закрыть его с помощью метода `Close()`. Этим гарантируется, что любые выведенные данные, оставшиеся в дисковом буфере, будут записаны на диск. В этом случае отпадает необходимость вызывать метод `Flush()` перед закрытием файла.

Ниже приведен простой пример программы, в котором демонстрируется порядок записи данных в файл.

```
// Записать данные в файл.

using System;
using System.IO;

class WriteToFile {
  static void Main(string[] args) {
    FileStream fout = null;

    try {
      // Открыть выходной файл.
      fout = new FileStream("test.txt", FileMode.CreateNew);

      // Записать весь английский алфавит в файл.
      for(char c = 'A'; c <= 'Z'; C++)
        fout.WriteByte((byte) c);
    } catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      if (fout != null) fout.Close();
    }
  }
}
```

В данной программе сначала создается выходной файл под названием `test.txt` с помощью перечисляемого значения `FileMode.CreateNew`. Это означает, что файл с таким же именем не должен уже существовать. (В противном случае генерируется исключение `IOException`.) После открытия выходного файла в него записываются

прописные буквы английского алфавита. По завершении данной программы содержимое файла `test.txt` оказывается следующим.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

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.

```
/* Копировать файл.
   Чтобы воспользоваться этой программой, укажите имена исходного и выходного
   файлов. Например, для копирования файла FIRST.DAT в файл SECOND.DAT
   введите в командной строке следующее:

   CopyFile FIRST.DAT SECOND.DAT
*/

using System;
using System.IO;

class CopyFile {
  static void Main(string[] args) {
    int i;
    FileStream fin = null;
    FileStream fout = null;

    if(args.Length != 2) {
      Console.WriteLine("Применение: CopyFile Откуда Куда");
      return;
    }

    try {
       // Открыть файлы.
       fin = new FileStream(args[0], FileMode.Open);
       fout = new FileStream(args[1], FileMode.Create);

       // Скопировать файл.
       do {
         i = fin.ReadByte();
         if(i != -1) fout.WriteByte((byte)i);
       } while (i != —1);
    } catch(IOException exc) {
        Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      if(fin != null) fin.Close();
      if(fout != null) fout.Close();
    }
  }
}
```

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:

```
StreamWriter(Stream поток)
```

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`.

```
// Простая сервисная программа ввода с клавиатуры и вывода на диск,
// демонстрирующая применение класса StreamWriter.

using System;
using System.IO;

class KtoD {
  static void Main() {
    string str;
    FileStream fout;

    // Открыть сначала поток файлового ввода-вывода.
    try {
      fout = new FileStream("test.txt", FileMode.Create);
    }
```

```
    catch(IOException exc) {
      Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
      return;
    }

    // Заключить поток файлового ввода-вывода в оболочку класса StreamWriter.
    StreamWriter fstr_out = new StreamWriter(fout);

    try {
      Console.WriteLine("Введите текст, а по окончании — 'стоп'.");
      do {
        Console.Write(": ");
        str = Console.ReadLine();

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

Ниже приведен вариант представленной ранее сервисной программы ввода с клавиатуры и вывода на диск, измененный таким образом, чтобы открывать выходной файл средствами самого класса `StreamWriter`.

```
// Открыть файл средствами класса StreamWriter.

using System;
using System.IO;

class KtoD {
  static void Main() {
    string str;
    StreamWriter fstr_out = null;

    try {
      // Открыть файл, заключенный в оболочку класса StreamWriter.
```

```
      fstr_out = new StreamWriter("test.txt");

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

      do {
        Console.Write(" : ");
        str = Console.ReadLine();

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

```
// Простая сервисная программа ввода с диска и вывода на экран,
// демонстрирующая применение класса StreamReader.

using System;
using System.IO;

class DtoS {
  static void Main() {
    FileStream fin;
    string s;

    try {
      fin = new FileStream("test.txt", FileMode.Open);
    }
```

```
    catch(IOException exc) {
      Console.WriteLine("Ошибка открытия файла:\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("Ошибка ввода-вывода:\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);
}
```

В данном случае код немного упрощается благодаря свойству `EndOfStream`, хотя общий порядок выполнения операции ввода из файла не меняется. Иногда такое применение свойства `EndOfStream` позволяет несколько упростить сложную ситуацию, внося ясность и улучшая структуру кода.

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.

```
using System;

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 C#, 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.

```
// Переадресовать поток Console.Out.

using System;
using System.IO;

class Redirect {
  static void Main() {
    StreamWriter log_out = null;
```

```
    try {
      log_out = new StreamWriter("logfile.txt");

      // Переадресовать стандартный вывод в файл logfile.txt.
      Console.SetOut(log_out);

      Console.WriteLine("Это начало файла журнала регистрации.");

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

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

При выполнении этой программы на экран ничего не выводится, но файл `logfile.txt` будет содержать следующее.

```
Это начало файла журнала регистрации.
0
1
2
3
4
5
6
7
8
9
Это конец файла журнала регистрации.
```

Попробуйте сами поупражняться в переадресации других встроенных потоков.

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.

Таблица 14.5. Наиболее часто используемые методы, определенные в классе `BinaryWriter`

Метод	Описание
`void Write(sbyte value)`	Записывает значение типа `sbyte` со знаком
`void Write(byte value)`	Записывает значение типа `byte` без знака
`void Write(byte[] buffer)`	Записывает массив значений типа `byte`
`void Write(short value)`	Записывает целочисленное значение типа `short` (короткое целое)
`void Write(ushort value)`	Записывает целочисленное значение типа `ushort` (короткое целое без знака)
`void Write(int value)`	Записывает целочисленное значение типа `int`
`void Write(uint value)`	Записывает целочисленное значение типа `uint` (целое без знака)
`void Write(long value)`	Записывает целочисленное значение типа `long` (длинное целое)
`void Write(ulong value)`	Записывает целочисленное значение типа `ulong` (длинное целое без знака)
`void Write(float value)`	Записывает значение типа `float` (с плавающей точкой одинарной точности)
`void Write(double value)`	Записывает значение типа `double` (с плавающей точкой двойной точности)
`void Write(decimal value)`	Записывает значение типа `decimal` (с двумя десятичными разрядами после запятой)
`void Write(char ch)`	Записывает символ
`void Write(char[] buffer)`	Записывает массив символов
`void Write(string value)`	Записывает строковое значение типа `string`, представленное во внутреннем формате с указанием длины строки

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.

Таблица 14.6. Наиболее часто используемые методы, определенные в классе `BinaryReader`

Метод	Описание
`bool ReadBoolean()`	Считывает значение логического типа `bool`
`byte ReadByte()`	Считывает значение типа `byte`
`sbyte ReadSByte()`	Считывает значение типа `sbyte`
`byte[] ReadBytes(int count)`	Считывает количество *count* байтов и возвращает их в виде массива
`char ReadChar()`	Считывает значение типа `char`
`char[] ReadChars(int count)`	Считывает количество *count* символов и возвращает их в виде массива
`decimal ReadDecimal()`	Считывает значение типа `decimal`
`double ReadDouble()`	Считывает значение типа `double`
`float ReadSingle()`	Считывает значение типа `float`
`short ReadInt16()`	Считывает значение типа `short`
`int ReadInt32()`	Считывает значение типа `int`
`long ReadInt64()`	Считывает значение типа `long`
`ushort ReadUIntl6()`	Считывает значение типа `ushort`
`uint ReadUInt32()`	Считывает значение типа `uint`
`ulong ReadUInt64()`	Считывает значение типа `ulong`
`string ReadString()`	Считывает значение типа `string`, представленное во внутреннем двоичном формате с указанием длины строки. Этот метод следует использовать для считывания строки, которая была записана средствами класса `BinaryWriter`

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()`.

Метод	Описание
`int Read()`	Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца файла возвращает значение -1
`int Read(byte[] buffer, int offset, int count)`	Делает попытку прочитать количество *count* байтов в массив *buffer*, начиная с элемента *buffer*[*offset*], и возвращает количество успешно считанных байтов
`int Read(char[]buffer, int offset, int count)`	Делает попытку прочитать количество *count* символов в массив *buffer*, начиная с элемента *buffer*[*offset*], и возвращает количество успешно считанных символов

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.

```
// Записать двоичные данные, а затем считать их обратно.

using System;
using System.IO;

class RWData {
  static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    int i = 10;
    double d = 1023.56;
    bool b = true;
    string str = "Это тест";

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

    // Записать данные в файл.
    try {
      Console.WriteLine("Запись " + i);
      dataOut.Write(i);

      Console.WriteLine("Запись " + d);
      dataOut.Write(d);

      Console.WriteLine("Запись " + b);
      dataOut.Write(b);
```

```
      Console.WriteLine("Запись " + 12.2 * 7.4);
      dataOut.Write(12.2 * 7.4);

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

  Console.WriteLine();

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

    try {
      i = dataIn.ReadInt32();
      Console.WriteLine("Чтение " + i);
      d = dataIn.ReadDouble();

      Console.WriteLine("Чтение " + d);
      b = dataIn.ReadBoolean () ;

      Console.WriteLine("Чтение " + b);
      d = dataIn.ReadDouble();

      Console.WriteLine("Чтение " + d);
      str = dataIn.ReadString();

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

Вот к какому результату приводит выполнение этой программы.

```
Запись 10
Запись 1023.56
Запись True
Запись 90.28
Запись Это тест
```

```
Чтение 10
Чтение 1023.56
Чтение True
Чтение 90.28
Чтение Это тест
```

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.

```
/* Использовать классы BinaryReader и BinaryWriter для
   реализации простой программы учета товарных запасов. */

using System;
using System.IO;

class Inventory {
  static void Main() {
    BinaryWriter dataOut;
    BinaryReader detaIn;

    string item; // наименование предмета
    int onhand;  // имеющееся в наличии количество
    double cost; // цена

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

    // Записать данные о товарных запасах в файл.
    try {
      dataOut.Write("Молотки");
      dataOut.Write(10);
      dataOut.Write(3.95);

      dataOut.Write("Отвертки");
      dataOut.Write(18);
      dataOut.Write(1.50);

      dataOut.Write("Плоскогубцы");
      dataOut.Write(5);
```

```
    dataOut.Write(4.95);

    dataOut.Write("Пилы");
    dataOut.Write(8);
    dataOut.Write(8.95);
  }
  catch(IOException exc) {
    Console.WriteLine("Ошибка записи в файл товарных запасов");
    Console.WriteLine("Причина: " + exc.Message);
  } finally {
    dataOut.Close();
  }

  Console.WriteLine();

  // А теперь открыть файл товарных запасов для чтения.
  try {
    dataIn = new
    BinaryReader(new FileStream("inventory.dat", FileMode.Open));
  }
  catch(IOException exc) {
    Console.WriteLine("Не удается открыть файл " +
                      "товарных запасов для ввода");
    Console.WriteLine("Причина: " + exc.Message);
    return;
  }

  // Найти предмет, введенный пользователем.
  Console.Write("Введите наименование для поиска: ");
  string what = Console.ReadLine();
  Console.WriteLine();

  try {
    for(;;) {
      // Читать данные о предмете хранения.
      item = dataIn.ReadString();
      onhand = dataIn.ReadInt32();
      cost = dataIn.ReadDouble();

      // Проверить, совпадает ли он с запрашиваемым предметом.
      // Если совпадает, то отобразить сведения о нем.

      if(item.Equals(what, StringComparison.OrdinalIgnoreCase)) {
        Console.WriteLine(item + ": " + onhand + " штук в наличии. " +
                          "Цена: {0:C} за штуку", cost);
        Console.WriteLine("Общая стоимость по наименованию <{0}>: {1:C}.",
                          item, cost * onhand);
        break;
      }
    }
  }
  catch(EndOfStreamException) {
    Console.WriteLine("Предмет не найден.");
  }
```

```
    catch(IOException exc) {
      Console.WriteLine("Ошибка чтения из файла товарных запасов");
      Console.WriteLine("Причина: " + exc.Message);
    } finally {
      dataIn.Glose();
    }
  }
}
```

Выполнение этой программы может привести, например, к следующему результату.

```
Введите наименование для поиска: Отвертки

Отвертки: 18 штук в наличии. Цена: $1.50 за штуку.
Общая стоимость по наименованию <Отвертки>: $27.00.
```

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`.

Значение	Описание
`SeekOrigin.Begin`	Поиск от начала файла
`SeekOrigin.Current`	Поиск от текущего положения
`SeekOrigin.End`	Поиск от конца файла

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.

```
// Продемонстрировать произвольный доступ к файлу.

using System;
using System.IO;

class RandomAccessDemo {
  static void Main() {
    FileStream f = null;
    char ch;

    try {
      f = new FileStream("random.dat", FileMode.Create);
      // Записать английский алфавит в файл.
      for (int i=0; i < 26; i++)
        f.WriteByte((byte)('A'+i));

      // А теперь считать отдельные буквы английского алфавита.
      f.Seek(0, SeekOrigin.Begin); // найти первый байт
      ch = (char) f.ReadByte();
      Console.WriteLine("Первая буква: " + ch);

      f.Seek(1, SeekOrigin.Begin); // найти второй байт
      ch = (char) f.ReadByte();
      Console.WriteLine("Вторая буква: " + ch);

      f.Seek(4, SeekOrigin.Begin); // найти пятый байт
      ch = (char) f.ReadByte();
      Console.WriteLine("Пятая буква: " + ch);

      Console.WriteLine ();

      // А теперь прочитать буквы английского алфавита через одну.
      Console.WriteLine("Буквы алфавита через одну: ");
      for(int i=0; i < 26; i += 2) {
        f.Seek(i, SeekOrigin.Begin); // найти i-й символ
        ch = (char) f.ReadByte();
        Console.Write(ch + " ");
      }
    }
    catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
    } finally {
      if(f != null) f.Close();
    }

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
Первая буква: A
Вторая буква: B
Пятая буква: E

Буквы алфавита через одну:
A C E G I K M O Q S U W Y
```

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.

В качестве примера ниже приведена программа, демонстрирующая применение класса `MemoryStream` в операциях ввода-вывода.

```
// Продемонстрировать применение класса MemoryStream.

using System;
using System.IO;

class MemStrDemo {
  static void Main() {
    byte[] storage = new byte[255];

    // Создать запоминающий поток.
    MemoryStream memstrm = new MemoryStream(storage);

    // Заключить объект memstrm в оболочки классов
```

```
    // чтения и записи данных в потоки.
    StreamWriter memwtr = new StreamWriter(memstrm);
    StreamReader memrdr = new StreamReader(memstrm);

    try {
    // Записать данные в память, используя объект memwtr.
    for(int i=0; i < 10; i++)
      memwtr.WriteLine("byte [" + i + "]: " + i);

      // Поставить в конце точку.
      memwtr.WriteLine(".");

      memwtr.Flush();

      Console.WriteLine("Чтение прямо из массива storage: ");

      // Отобразить содержимое массива storage непосредственно,
      foreach(char ch in storage) {
        if (ch == '.') break;
        Console.Write(ch);
      }

      Console.WriteLine("\nЧтение из потока с помощью объекта memrdr: ");

      // Читать из объекта memstrm средствами ввода данных из потока.
      memstrm.Seek(0, SeekOrigin.Begin); // установить указатель файла
                                         // в исходное положение
      string str = memrdr.ReadLine();
      while(str != null) {
        str = memrdr .ReadLine();
        if (str[0] == '.') break;
        Console.WriteLine(str);
      }
    } catch(IOException exc) {
       Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
    } finally {
       // Освободить ресурсы считывающего и записывающего потоков.
       memwtr.Close();
       memrdr.Close();
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Чтение прямо из массива storage:
byte [0]: 0
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

```
Чтение из потока с помощью объекта memrdr:
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

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()`.

Ниже приведен пример, демонстрирующий применение классов `StringReader` и `StringWriter`.

```
// // Продемонстрировать применение классов StringReader и StringWriter.

using System;
using System.IO;

class StrRdrWtrDemo {
  static void Main() {
    StringWriter strwtr = null;
    StringReader strrdr = null;

    try {
      // Создать объект класса StringWriter.
      strwtr = new StringWriter();

      // Вывести данные в записывающий поток типа StringWriter.
      for (int i=0; i < 10; i++)
        strwtr.WriteLine("Значение i равно: " + i);

      // Создать объект класса StringReader.
      strrdr = new StringReader(strwtr.ToString());

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

Вот к каком результату приводит выполнение этого кода.

```
Значение i равно: 1
Значение i равно: 2
Значение i равно: 3
Значение i равно: 4
Значение i равно: 5
Значение i равно: 6
Значение i равно: 7
Значение i равно: 8
Значение i равно: 9
```

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 Copy(), 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.

СОВЕТ

Ряд методов для работы с файлами определен также в классе FileInfo. Этот класс отличается от класса File одним, очень важным преимуществом: для операций над файлами он предоставляет методы экземпляра и свойства, а не статические методы. Поэтому для выполнения нескольких операций над одним и тем же файлом лучше воспользоваться классом FileInfo.

Копирование файлов с помощью метода Copy()

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 Copy(), 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_целевого_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 `Copy()` 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.

```
/* Скопировать файл, используя метод File.Copy().

   Чтобы воспользоваться этой программой, укажите имя исходного и целевого
   файлов. Например, чтобы скопировать файл FIRST.DAT в файл SECOND.DAT,
   введите в командной строке следующее:

   CopyFile FIRST.DAT SECOND.DAT
*/

using System;
using System.IO;

class CopyFile {
  static void Main(string[] args) {
    if(args.Length != 2) {
     Console.WriteLine("Применение : CopyFile Откуда Куда");
     return;
    }

    // Копировать файлы.
    try {
      File.Copy(args[0], args[1]);
    } catch(IOException exc) {
      Console.WriteLine("Ошибка копирования файла\n" + exc.Message);
    }
  }
}
```

Как видите, в этой программе не нужно создавать поток типа `FileStream` или освобождать его ресурсы. Все это делается в методе `Copy()` 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 `Copy()`.

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. (Структура

DateTime описывается далее в этой книге, но метод ToString() автоматически приводит дату и время к удобочитаемому виду.) 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.

```
// Применить методы Exists() и GetLastAccessTime().

using System;
using System.IO;

class ExistsDemo {
  static void Main() {
    if(File.Exists("test.txt"))
    Console.WriteLine("Файл существует. В последний раз он был доступен " +
                      File.GetLastAccessTime("test.txt"));
     else
    Console.WriteLine("Файл не существует");
  }
}
```

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

Прежде чем завершить обсуждение темы ввода-вывода, рассмотрим еще один способ, который может пригодиться при чтении числовых строк. 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.

Имя структуры в .NET	Имя типа данных в C#
Decimal	decimal
Double	double
Single	float
Int16	short
Int32	int
Int64	long
UInt16	ushort
UInt32	uint
Uint64	ulong
Byte	byte
Sbyte	sbyte

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 C#. 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.

Структура	Метод преобразования
Decimal	`static decimal Parse(string s)`
Double	`static double Parse(string s)`
Single	`static float Parse(string s)`
Int64	`static long Parse(string s)`
Int32	`static int Parse(string s)`
Intl6	`static short Parse(string s)`
UInt64	`static ulong Parse(string s)`
UInt32	`static uint Parse(string s)`
Ulnt16	`static ushort Parse(string s)`
Byte	`static byte Parse(string s)`
Sbyte	`static sbyte Parse(string s)`

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`.

```
// Эта программа усредняет ряд чисел, вводимых пользователем.

using System;
using System.IO;

class AvgNums {
  static void Main() {
    string str;
    int n;
    double sum = 0.0;
    double avg, t;

    Console.Write("Сколько чисел вы собираетесь ввести: ");
    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("Введите " + n + " чисел.");
    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("Среднее равно " + avg);
  }
}
```

Выполнение этой программы может привести, например, к следующему результату.

```
Сколько чисел вы собираетесь ввести: 5
Введите 5 чисел.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Среднее равно 3.3
```

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.