```
Last accessed 26/11/2018 09:08:26
Has readonly set to False
```

Количество байтов может отличаться в вашей операционной системе, поскольку операционные системы могут использовать разные окончания строки.

Контроль работы с файлами

При работе с файлами возникает необходимость контролировать, как именно они открываются. Метод `File.Open` содержит перегрузки для указания дополнительных параметров с помощью значений `enum`. Ниже приведены используемые типы `enum`:

- `FileMode` — определяет, что вы хотите сделать с файлом, например `CreateNew` (Создать новый), `OpenOrCreate` (Открыть файл, если он существует, или создать новый) или `Truncate` (Открыть с удалением текущего содержимого);
- `FileAccess` — определяет, какой уровень доступа вам нужен, например `ReadWrite` (Чтение и запись);
- `FileShare` — управляет блокировками файла, чтобы разрешить другим процессам указанный уровень доступа, например `Read` (Чтение).

Возможно, вы захотите открыть файл и прочитать его, а также разрешить его считывать другим процессам, как показано ниже:

```
FileStream file = File.Open(pathToFile,
  FileMode.Open, FileAccess.Read, FileShare.Read);
```

Существует также тип `enum` для атрибутов файла:

- `FileAttributes` — используется для проверки свойства `Attributes` типов, производных от `FileSystemInfo`. Например, выставлены ли для файла флаги `Archive` и `Encrypted`.

Вы можете проверить атрибуты файла или каталога, как показано ниже:

```
var info = new FileInfo(backupFile);
WriteLine("Is the backup file compressed? {0}",
  info.Attributes.HasFlag(FileAttributes.Compressed));
```

Чтение и запись с помощью потоков

Поток (stream)[1] представляет собой последовательность байтов, которую можно считать или в которую можно записать некие данные. Хотя файлы могут обрабатываться во многом подобно массивам, с произвольным доступом по известной

[1] В русскоязычной терминологии слово «поток» используется для обозначения англоязычных понятий stream (дословно: поток, струя, течение) и thread (дословно: нить). — *Примеч. пер.*

позиции байта в файле, считается полезным обрабатывать файлы как поток, в котором байты могут быть доступны в последовательном порядке.

Кроме того, потоки могут использоваться для обработки входных и выходных данных терминала и сетевых ресурсов, таких как сокеты и порты, которые не обеспечивают произвольный доступ и не могут искать расположение. Вы можете написать код для обработки произвольных байтов, не зная и не заботясь о том, откуда они берутся. Ваш код просто считывает или записывает в поток, а другой фрагмент кода определяет, где байты хранятся фактически.

Существует абстрактный класс `Stream`, представляющий собой поток. Есть множество классов, которые наследуются от этого базового, включая `FileStream`, `MemoryStream`, `BufferedStream`, `GZipStream` и `SslStream`, и потому все они работают одинаково.

Все потоки реализуют интерфейс `IDisposable`, поэтому имеют метод `Dispose` для освобождения неуправляемых ресурсов.

В табл. 9.1 приведены некоторые из универсальных членов класса `Stream`.

Таблица 9.1

Члены	Описание
CanRead, CanWrite	Определяют, поддерживает ли текущий поток возможность чтения и записи соответственно
Length, Position	Определяют длину потока в байтах и текущую позицию в потоке соответственно. Эти свойства могут вызвать исключение для некоторых типов потоков
Dispose()	Закрывают поток и освобождает его ресурсы
Flush()	Если поток имеет буфер, то байты в нем записываются в поток и буфер очищается
Read(), ReadAsync()	Считывают определенное количество байтов из потока в байтовый массив и перемещают позицию синхронно и асинхронно соответственно
ReadByte()	Считывает байт из потока и перемещает позицию
Seek()	Задает позицию в текущем потоке (если значение CanSeek истинно)
Write(), WriteAsync()	Записывают последовательность байтов в текущий поток синхронно и асинхронно соответственно
WriteByte()	Записывает байт в поток

В табл. 9.2 приведены некоторые *запоминающие потоки*, предоставляющие место хранения байтов.

В табл. 9.3 приведены некоторые *функциональные потоки*, которые не могут существовать сами по себе. Их можно «подключить» к другим потокам, чтобы расширить их функциональность.

Таблица 9.2

Пространство имен	Класс	Описание
System.IO	FileStream	Создает поток, хранилищем которого является файловая система
System.IO	Memory Stream	Создает поток, хранилищем которого является память
System.Net.Sockets	NetworkStream	Создает поток, хранилищем которого является сеть

Таблица 9.3

Пространство имен	Классы	Описание
System.Security.Cryptography	CryptoStream	Шифрует и дешифрует поток
System.IO.Compression	GZipStream, DeflateStream	Сжимают и распаковывают поток
System.Net.Security	AuthenticatedStream	Передает учетные данные через поток

Временами бывает необходимо работать с потоками на низком уровне. Чаще всего, однако, удается упростить задачу, добавив в цепочку специальные вспомогательные классы.

Все вспомогательные типы для потоков реализуют интерфейс `IDisposable`, поэтому имеют метод `Dispose` для освобождения неуправляемых ресурсов.

В табл. 9.4 представлены некоторые вспомогательные классы для обработки распространенных сценариев.

Таблица 9.4

Пространство имен	Класс	Описание
System.IO	StreamReader	Считывает данные из потока в текстовом формате
	StreamWriter	Записывает данные в поток в текстовом формате
	BinaryReader	Считывает данные из потока в виде типов .NET. Например, метод ReadDecimal считывает следующие 16 байт из базового потока в виде десятичного значения, а метод ReadInt32 считывает следующие четыре байта как значение int
	BinaryWriter	Записывает данные в поток в виде типов .NET. Например, метод Write с параметром типа decimal записывает 16 байт в базовый поток, а метод Write с параметром типа int записывает четыре байта

Продолжение ⇩

Таблица 9.4 *(продолжение)*

Пространство имен	Класс	Описание
System.Xml	XmlReader	Считывает данные из потока в XML-формате
	XmlWriter	Записывает данные в поток в XML-формате

Запись в текстовые потоки

Напишем код для записи текста в поток.

1. Создайте проект консольного приложения `WorkingWithStreams`, добавьте его в рабочую область `Chapter09` и выберите проект как активный для OmniSharp.
2. Импортируйте пространства имен `System.IO` и `System.Xml`, статически импортируйте типы `System.Console`, `System.Environment` и `System.IO.Path`.
3. Определите массив строковых значений позывных пилота вертолета Viper и создайте метод `WorkWithText`, перечисляющий позывные, записывая каждый из них в текстовый файл, как показано ниже:

```
// определение массива позывных пилота Viper
static string[] callsigns = new string[] {
  "Husker", "Starbuck", "Apollo", "Boomer",
  "Bulldog", "Athena", "Helo", "Racetrack" };

staticvoidWorkWithText()
{
  // определение файла для записи
  string textFile = Combine(CurrentDirectory, "streams.txt");

  // создание текстового файла и возвращение
  // вспомогательного объекта для записи
   StreamWriter text = File.CreateText(textFile);

  // перечисление строк с записью каждой из них
  // в поток в отдельной строке
  foreach (string item in callsigns)
  {
    text.WriteLine(item);
  }
  text.Close(); // освобождение ресурсов

  // вывод содержимого файла в консоль
  WriteLine("{0} contains {1:N0} bytes.",
    arg0: textFile,
    arg1: new FileInfo(textFile).Length);

  WriteLine(File.ReadAllText(textFile));
}
```

4. В методе `Main` вызовите метод `WorkWithText`.
5. Запустите приложение и проанализируйте результат:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.txt
contains 60 bytes.
Husker
Starbuck
Apollo
Boomer
Bulldog
Athena
Helo
Racetrack
```

6. Откройте созданный файл и убедитесь, что он содержит список позывных.

Запись в XML-потоки

Существует два способа написания XML элемента:

- `WriteStartElement` и `WriteEndElement` — используется, когда элемент содержит дочерние элементы;
- `WriteElementString` — используется, когда у элемента дочерние элементы отсутствуют.

Теперь сохраним тот же массив строковых значений в XML-файле.

1. Создайте метод `WorkWithXml`, перечисляющий позывные пилота вертолета Viper, как показано ниже:

```
static void WorkWithXml()
{
  // определение файла для записи
  string xmlFile = Combine(CurrentDirectory, "streams.xml");

  // создание файлового потока
  FileStream xmlFileStream = File.Create(xmlFile);

  // оборачивание файлового потока во вспомогательный объект
  // для записи XML и автоматическое добавление отступов
  // для вложенных элементов
  XmlWriter xml = XmlWriter.Create(xmlFileStream,
    new XmlWriterSettings { Indent = true });

  // запись объявления XML
  xml.WriteStartDocument();

  // запись корневого элемента
  xml.WriteStartElement("callsigns");
```

```
    // перечисление строк с записью каждой из них в поток
    foreach (string item in callsigns)
    {
      xml.WriteElementString("callsign", item);
    }

    // запись закрывающего корневого элемента
    xml.WriteEndElement();

    // закрытие вспомогательного объекта и потока
    xml.Close();
    xmlFileStream.Close();

    // вывод содержимого файла
    WriteLine("{0} contains {1:N0} bytes.",
      arg0: xmlFile,
      arg1: new FileInfo(xmlFile).Length);

    WriteLine(File.ReadAllText(xmlFile));
  }
```

2. В методе `Main` закомментируйте предыдущий вызов метода и добавьте вызов в метод `WorkWithXml`.
3. Запустите консольное приложение и проанализируйте результат:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.xml
contains 310 bytes.
<?xml version="1.0" encoding="utf-8"?>
<callsigns>
  <callsign>Husker</callsign>
  <callsign>Starbuck</callsign>
  <callsign>Apollo</callsign>
  <callsign>Boomer</callsign>
  <callsign>Bulldog</callsign>
  <callsign>Athena</callsign>
  <callsign>Helo</callsign>
  <callsign>Racetrack</callsign>
</callsigns>
```

Освобождение файловых ресурсов

При открытии файла для чтения или записи вы используете ресурсы вне .NET. Эти ресурсы называются неуправляемыми и должны быть освобождены по окончании работы с ними. Чтобы гарантировать их освобождение, можно вызывать метод `Dispose` внутри блока `finally`.

Чтобы правильно распоряжаться неуправляемыми ресурсами, отредактируем наш предыдущий код.

1. Измените метод `WorkWithXml`, как показано ниже (выделено полужирным шрифтом).

```
static void WorkWithXml()
{
  FileStream xmlFileStream = null;
  XmlWriterxml = null;

  Try
  {
    // определение файла для записи
    string xmlFile = Combine(CurrentDirectory, "streams.xml");

    // создание файлового потока
    xmlFileStream = File.Create(xmlFile);

    // оборачивание файлового потока во вспомогательный объект для записи
    // XML и автоматическое добавление отступов для вложенных элементов
    xml = XmlWriter.Create(xmlFileStream,
      new XmlWriterSettings { Indent = true });

    // запись объявления XML
    xml.WriteStartDocument();

    // запись корневого элемента
    xml.WriteStartElement("callsigns");

    // перечисление строк с записью каждой из них в поток
    foreach (string item in callsigns)
    {
      xml.WriteElementString("callsign", item);
    }

    // запись закрывающего корневого элемента
    xml.WriteEndElement();

    // закрытие вспомогательного объекта и потока
    xml.Close();
    xmlFileStream.Close();

    // вывод содержимого файла
    WriteLine($"{0} contains {1:N0} bytes.",
      arg0: xmlFile,
      arg1: new FileInfo(xmlFile).Length);

    WriteLine(File.ReadAllText(xmlFile));
  }
  catch(Exception ex)
  {
```

```
      // если путь не существует, то выбрасывается исключение
      WriteLine($"{ex.GetType()} says {ex.Message}");
    }
    finally
    {
      if (xml != null)
      {
        xml.Dispose();
        WriteLine("The XML writer's unmanaged resources have been disposed.");
      }
      if (xmlFileStream != null)
      {
        xmlFileStream.Dispose();
        WriteLine("The file stream's unmanaged resources have been disposed.");
      }
    }
  }
```

Вы также можете вернуться и изменить другие ранее созданные методы, но я оставлю это для вас в качестве дополнительного упражнения.

2. Запустите консольное приложение и проанализируйте результат:

```
The XML writer's unmanaged resources have been disposed.
The file stream's unmanaged resources have been disposed.
```

Перед вызовом метода Dispose убедитесь, что объект не равен null.

Вы можете упростить код, проверяющий объект на `null`, а затем вызывающий его метод `Dispose` с помощью оператора `using`.

Может немного сбить с толку то, что существует два варианта использования ключевого слова `using`: импорт пространства имен и генерация оператора `finally`, который вызывает метод `Dispose` для объекта, реализующего интерфейс `IDisposable`.

Компилятор изменяет блок оператора `using` на оператор `try-finally` без оператора `catch`. Кроме того, вы можете использовать вложенные операторы `try`, так что при желании вы все еще можете перехватить какие-либо исключения. Рассмотрим это на примере следующего кода:

```
using (FileStream file2 = File.OpenWrite(
  Path.Combine(path, "file2.txt")))
{
  using (StreamWriter writer2 = new StreamWriter(file2))
  {
    try
```

```
    {
      writer2.WriteLine("Welcome, .NET Core!");
    }
    catch(Exception ex)
    {
      WriteLine($"{ex.GetType()} says {ex.Message}");
    }
  } // автоматический вызов метода Dispose, если объект не равен null
} // автоматический вызов метода Dispose, если объект не равен null
```

Сжатие потоков

Формат XML довольно объемный, поэтому занимает больше памяти в байтах, чем обычный текст. Мы можем сжать XML-данные, воспользовавшись всем знакомым алгоритмом сжатия, известным под названием *GZIP*.

1. Импортируйте следующее пространство имен:

```
using System.IO.Compression;
```

2. Добавьте метод `WorkWithCompression`, использующий экземпляры `GZipSteam` для создания сжатого файла, содержащего те же элементы XML, что и раньше, а затем распаковывающий его при чтении и выводе на консоль, как показано ниже:

```
static void WorkWithCompression()
{
  // сжатие XML-вывода
  string gzipFilePath = Combine(
    CurrentDirectory, "streams.gzip");

  FileStream gzipFile = File.Create(gzipFilePath);

  using (GZipStream compressor = new GZipStream(
    gzipFile, CompressionMode.Compress))
  {
    using (XmlWriter xmlGzip = XmlWriter.Create(compressor))
    {
      xmlGzip.WriteStartDocument();
      xmlGzip.WriteStartElement("callsigns");

      foreach (string item in callsigns)
      {
        xmlGzip.WriteElementString("callsign", item);
      }
      // вызов метода WriteEndElement необязателен,
      // поскольку, освобождаясь, XmlWriter
      // автоматически закрывает любые элементы
    }
  } // закрытие основного потока
```

```
    // выводит все содержимое сжатого файла в консоль
    WriteLine("{0} contains {1:N0} bytes.",
      gzipFilePath, new FileInfo(gzipFilePath).Length);

    WriteLine($"The compressed contents:");
    WriteLine(File.ReadAllText(gzipFilePath));

    // чтение сжатого файла
    WriteLine("Reading the compressed XML file:");
    gzipFile = File.Open(gzipFilePath, FileMode.Open);

    using (GZipStream decompressor = new GZipStream(
      gzipFile, CompressionMode.Decompress))
    {
      using (XmlReader reader = XmlReader.Create(decompressor))
      {
        while (reader.Read()) // чтение сжатого файла
        {
          // проверить, находимся ли мы на элементе с именем callsign
          if ((reader.NodeType == XmlNodeType.Element)
            && (reader.Name == "callsign"))
          {
            reader.Read(); // переход к тексту внутри элемента
            WriteLine($"{reader.Value}"); // чтение его значения
          }
        }
      }
    }
  }
```

3. В методе `Main` оставьте вызов метода `WorkWithXml` и добавьте вызов метода `WorkWithCompression`, как показано ниже:

```
static void Main(string[] args)
{
  // WorkWithText();
  WorkWithXml();
  WorkWithCompression();
}
```

4. Запустите консольное приложение и сравните размеры XML-файла и сжатого XML-файла. Обратите внимание: сжатые XML-данные занимают вполовину меньше объема памяти по сравнению с таким же количеством несжатых.

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.xml
contains 310 bytes.

/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.gzip
contains 150 bytes.
```

Сжатие с помощью алгоритма Brotli

Выпуская платформу .NET Core 2.1, корпорация Microsoft представила реализацию алгоритма сжатия Brotli. По производительности он похож на алгоритм, используемый в DEFLATE и GZIP, но результат, как правило, сжат на 20 % больше.

1. Измените метод `WorkWithCompression`, в котором необязательный параметр указывает, следует ли использовать алгоритм Brotli, и по умолчанию указывает, что алгоритм Brotli следует использовать, как показано ниже (выделено полужирным шрифтом):

```
static void WorkWithCompression(bool useBrotli = true)
{
  string fileExt = useBrotli ? "brotli" : "gzip";

  // сжатие XML-вывода
  string filePath = Combine(
    CurrentDirectory, $"streams.{fileExt}");

  FileStream file = File.Create(filePath);

  Stream compressor;

  if (useBrotli)
  {
    compressor = new BrotliStream(file, CompressionMode.Compress);
  }
  else
  {
    compressor = new GZipStream(file, CompressionMode.Compress);
  }

  using (compressor)
  {
    using (XmlWriter xml = XmlWriter.Create(compressor))
    {
      xml.WriteStartDocument();
      xml.WriteStartElement("callsigns");
      foreach (string item in callsigns)
      {
        xml.WriteElementString("callsign", item);
      }
    }
  } // закрытие основного потока

  // выводит все содержимое сжатого файла в консоль
  WriteLine("{0} contains {1:N0} bytes.",
    filePath, new FileInfo(filePath).Length);

  WriteLine(File.ReadAllText(filePath));
```

```
    // чтение сжатого файла
    WriteLine("Reading the compressed XML file:");
    file = File.Open(filePath, FileMode.Open);

    Stream decompressor;

    if (useBrotli)
    {
      decompressor = new BrotliStream(
        file, CompressionMode.Decompress);
    }
    else
    {
      decompressor = new GZipStream(
        file, CompressionMode.Decompress);
    }

    using (decompressor)
    {
      using (XmlReader reader = XmlReader.Create(decompressor))
      {
        while (reader.Read())
        {
          // проверить, находимся ли мы на элементе с именем callsign
          if ((reader.NodeType == XmlNodeType.Element)
            && (reader.Name == "callsign"))
          {
            reader.Read(); // переход к тексту внутри элемента
            WriteLine($"{reader.Value}"); // чтение его значения
          }
        }
      }
    }
  }
```

2. Измените метод `Main`, чтобы он дважды вызывал метод `WorkWithCompression`, один раз по умолчанию с помощью алгоритма Brotli и один раз с использованием утилиты GZIP, как показано ниже:

```
WorkWithCompression();
WorkWithCompression(useBrotli: false);
```

3. Запустите консольное приложение и сравните размеры двух сжатых файлов XML. Благодаря использованию алгоритма Бротли сжатые XML-данные занимают почти на 21 % меньше места, как показано в следующем отредактированном выводе:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.brotli
contains 118 bytes.
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.gzip
contains 150 bytes.
```

Высокопроизводительные потоки с использованием конвейеров

Выпуская платформу .NET Core 2.1, корпорация Microsoft представила *конвейеры*. Для правильной обработки данных из потока требуется написать сложный стандартный код, который трудно сопровождать. Тестирование на вашем локальном ноутбуке часто выполняется успешно с небольшими тестовыми файлами, но в действительности код может не работать из-за неправильных предположений. Разобраться с этой проблемой помогут конвейеры.

Несмотря на то что конвейеры эффективны в использовании и вы захотите узнать о них больше, привести хороший пример будет сложно, поэтому я не планирую освещать их в данной книге. Более подробно о конвейерах можно прочитать на сайте blogs.msdn.microsoft.com/dotnet/2018/07/09/system-io-pipelines-high-performance-io-in-net/.

Асинхронные потоки

Начиная с C# 8.0 и .NET Core 3.0 корпорация Microsoft ввела асинхронную обработку потоков, о которой вы узнаете в главе 13.

Учебное пособие по асинхронным потокам можно получить по следующей ссылке: docs.microsoft.com/ru-ru/dotnet/csharp/tutorials/generate-consume-asynchronous-stream.

Кодирование и декодирование текста

Текстовые символы могут быть представлены разными способами. Например, алфавит можно закодировать с помощью азбуки Морзе в серии точек и тире для передачи по телеграфной линии.

Аналогичным образом текст в памяти компьютера сохраняется в виде битов (единиц и нулей), представляющих кодовую точку в кодовом пространстве. Большинство кодовых точек представляют один текстовый символ, но некоторые могут иметь и другое значение, например, форматирование.

Например, таблица ASCII имеет кодовое пространство со 128 кодовыми точками. Платформа .NET использует стандарт *Unicode* для внутреннего кодирования текста. Данный стандарт содержит более миллиона кодовых точек.

Иногда возникает необходимость переместить текст за пределы .NET для использования системами, не применяющими Unicode или другую вариацию стандарта Unicode, поэтому важно научиться преобразовывать кодировки.