

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

Брянский государственный технический университет

**Утверждаю**

**Ректор университета**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.Н.Федонин**

**«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2017 г.**

**Программирование в среде Microsoft .NET**

**Организация работы с потоками данных**

**Методические указания**

**к выполнению лабораторной работы №8**

**для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия»**

**Брянск 2017**

УКД 004.43

Программирование в среде Microsoft .Net. Организация работы с потоками данных [Текст] + [Электронный ресурс]: Методические указания к выполнению лабораторной работы №8 для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия». – Брянск: БГТУ, 2017. – 38 с.

Разработал:

Д.А.Коростелёв

канд. техн. наук, доц.

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол №2 от 16.09.2016)

**Методические издания публикуются в авторской редакции**

**Цель работы**

Целью работы является изучение основ организации работы с потоками данных с помощью библиотеки классов среды Microsoft .Net.

Продолжительность работы – 2 часа.

**Потоки и ввод-вывод**

Ввод-вывод строится на основе потоков. Большинство классов для работы с потоками и вводом-выводом находится в пространстве имен System.IO.

В основе потоковой архитектуры .NET лежат три понятия:

* опорное хранилище (backing store);
* декоратор (decorator);
* адаптер (adapter).

Опорное хранилище – это конечная точка ввода-вывода: файл, сетевое подключение и т.д. Оно может представлять собой либо источник, из которого последовательно считываются байты, либо приемник, куда байты последовательно записываются, либо и то и другое вместе.

Чтобы использовать опорное хранилище его нужно открыть. Этой цели и служат потоки, которые в .NET представлены классом System.IO.Stream, содержащий методы для чтения, записи и позиционирования потоков.

Потоки не загружают опорное хранилище в память целиком, а читают его последовательно по байтам либо блокам управляемого размера. Поэтому поток может потреблять мало памяти независимо от размера его опорного хранилища.

Потоки делятся на две категории:

* потоки опорных хранилищ – потоки, жестко привязанные к конкретным типам опорных хранилищ, такие как FileStream или NetworkStream
* потоки-декораторы – наполняют другие потоки, трансформируя данные тем или иным способом, такие как DeflateStream или CryptoStream

Декораторы освобождают потоки опорных хранилищ от необходимости самостоятельно реализовывать такие вещи, как сжатие и шифрование. Декораторы можно подключать во время выполнения, а также соединять их в цепочки (т.е. использовать несколько декораторов в одном потоке).

Потоки работают с байтами. Это гибко и эффективно, но не всегда удобно, например, когда приложение имеет дело с текстом или XML. Преодолеть этот разрыв позволяют адаптеры, они помещают поток в оболочку класса со специальными методами для конкретного формата. В отличие от декораторов адаптеры сами не являются потоками, они обычно полностью скрывают байт-ориентированные методы.

**Потоки**

Абстрактный класс Stream является базовым для всех потоков. Он определяет методы и свойства для трех фундаментальных операций: чтение, запись и поиск, а также для выполнения служебных задач: закрытие, сброс, конфигурирование тайм-аутов:

// Чтение:

public abstract bool CanRead { get; }

public abstract int Read (byte[] buffer, int offset, int count)

public virtual int ReadByte ();

// Запись:

public abstract bool CanWrite { get; }

public abstract void Write (byte[] buffer, int offset, int count);

public virtual void WriteByte (byte value);

// Поиск:

public abstract bool CanSeek { get; }

public abstract long Position { get; set; }

public abstract void SetLength (long value);

public abstract long Length { get; }

public abstract long Seek (long offset, SeekOrigin origin);

// Закрытие/сброс:

public virtual void Close ();

public void Dispose ();

public abstract void Flush ();

// Тайм-ауты:

public virtual bool CanTimeout { get; }

public virtual int ReadTimeout { get; set; }

public virtual int WriteTimeout { get; set; }

// Другие:

public static readonly Stream Null;

public static Stream Synchronized (Stream stream);

Пример:

using System;

using System.IO;

class Program

{

static void Main()

{

// CСоздать в текущем каталоге файл test.txt:

using (Stream s = new FileStream ("test.txt", FileMode.Create))

{

Console.WriteLine (s.CanRead); // True

Console.WriteLine (s.CanWrite); // True

Console.WriteLine (s.CanSeek); // True

s.WriteByte (101);

s.WriteByte (102);

byte[] block = { 1, 2, 3, 4, 5 };

s.Write (block, 0, block.Length); // Записать блок из 5 байтов

Console.WriteLine (s.Length); // 7

Console.WriteLine (s.Position); // 7

s.Position = 0; // Переместиться обратно в начало

Console.WriteLine (s.ReadByte()); // 101

Console.WriteLine (s.ReadByte()); // 102

// Читать из потока в массив block:

Console.WriteLine (s.Read (block, 0, block.Length)); // 5

Console.WriteLine (s.Read (block, 0, block.Length)); // 0

}

}

}

**Чтение и запись**

Поток может поддерживать чтение, запись или то и другое. Если свойство CanWrite возвращает false — поток предназначен только для чтения, если CanRead возвращает false — поток предназначен только для записи.

Метод Read читает блок данных из потока и записывает их в массив. Он возвращает количество полученных байтов, которое может быть либо равно, либо меньше значения аргумента count. Если оно меньше count — это значит, что достигнут конец потока или поток выдает данные порциями меньшего размера (в связи с этим однозначно судить о том, что достигнут конец потока можно только если метод возвращает 0):

byte[] data = new byte [1000];

// bytesRead в конце всегда будет равен 1000, если только сам поток не короче:

int bytesRead = 0;

int chunkSize = 1;

while (bytesRead < data.Length && chunkSize > 0)

bytesRead += chunkSize = s.Read (data, bytesRead, data.Length - bytesRead);

Метод ReadByte читает один байт из потока, в случае достижения конца потока возвращает -1.

Методы Write и WriteByte отправляют данные в поток, в случае неудачи генерируют исключение.

Аргумент offset методов Read и Write ссылается на индекс в массиве buffer, с которого начинается чтение или запись, а не на позицию в потоке.

**Позиционирование (Seeking)**

Если свойство CanSeek возвращает true, то поток поддерживает возможность позиционирования. Для такого потока можно запрашивать свойство Length и модифицировать его с помощью метода SetLength. Также можно в любой момент изменять свойство Position, отражающее позицию относительно начала потока, в которой производится чтение или запись. Метод Seek позволяет перемещаться относительно текущей позиции или относительно конца потока.

Если поток не поддерживает возможность позиционирования, единственный способ определить длину потока — прочитать его до конца. К тому же, если требуется повторно прочитать предшествующую область, необходимо закрыть поток и начать все заново.

**Закрытие и сброс**

Потоки должны быть освобождены после использования, чтобы освободить лежащие в их основе ресурсы. Самый простой способ обеспечения этого — создание экземпляров потока внутри блока using.

Помимо этого потоки содержат методы Dispose и Close (функционально идентичны, закрывают поток). Многократное освобождение или закрытие потока не вызывает ошибки.

Закрытие потока с декоратором закрывает и декоратор и поток с опорным хранилищем. В случае цепочки декораторов закрытие самого внешнего декоратора закрывает всю цепочку.

Для улучшения производительности некоторые потоки буферизуют данные, поступающие в/из опорного хранилища. По этой причине данные записываемые в поток могут не сразу попадать в опорное хранилище, возможна задержка до заполнения буфера. Метод Flush обеспечивает принудительную запись любых буферизированных данных. При закрытии потока метод Flush вызывается автоматически.

**Тайм-ауты**

Если свойство CanTimeout возвращает true, поток поддерживает тайм-аут чтения и записи (тайм-ауты поддерживают сетевые потоки, а файловые и потоки в памяти – нет). Для таких потоков свойство ReadTimeout задает тайм-аут в миллисекундах на чтение, а свойство WriteTimeout – тайм-аут на запись. Ноль означает отсутствие тайм-аута. При наступлении тайм-аута методы Read и Write генерируют исключения.

**Потоки с опорными хранилищами в .NET**

Основными потоками с опорными хранилищами в .NET являются:

* System.IO.FileStream;
* System.IO.MemoryStream;
* System.IO.IsolatedStorageFileStream;
* System.Net.Sockets.NetworkStream;
* System.IO.Pipes.PipeStream.

**Декораторы в .NET**

Декораторы помещают другой поток в оболочку, добавляя ему различный функционал. Основными декораторами в .NET являются:

* System.IO.BufferedStream;
* System.IO.Compression.DeflateStream;
* System.IO.Compression.GZipStream;
* System.Security.Cryptography.CryptoStream;
* System.Net.Security.AuthenticatedStream.

**Адаптеры потоков в .NET**

Текстовые адаптеры (для типов string и char):

* TextReader;
* TextWriter;
* StreamReader;
* StreamWriter;
* StringReader;
* StringWriter.

Двоичные адаптеры (для типов int, bool, string и float):

* BinaryReader;
* BinaryWriter.

Адаптеры XML:

* XmlReader;
* XmlWriter.

**FileStream**

Создать экземпляр FileStream можно с помощью статических методов класса File:

FileStream fs1 = File.OpenRead ("readme.bin"); // Только для чтения

FileStream fs2 = File.OpenWrite (@"c:\temp\writeme.tmp"); // Только для записи

FileStream fs3 = File.Create (@"c:\temp\writeme.tmp"); // Чтение и запись

Методы OpenWrite и Create ведут себя по разному если файл уже существует: метод Create удалит все содержимое существующего файла, а метод OpenWriteоставит содержимое файла не тронутым, установив позицию потока в ноль (если будет записано меньше байтов, чем существовало в файле, метод оставит смесь старого и нового содержимого).

Можно создать экземпляр FileStream с помощью конструктора класса FileStream, которому можно передать имя файла или файловый дескриптор, режимы создания и доступа к файлу, опции для совместного использования, буферизации и безопасности:

var fs = new FileStream ("readwrite.tmp", FileMode.Open); // Чтение и запись

Класс File определяет также более удобные статические методы, позволяющие открыть файл и прочитать/записать его содержимое за одни шаг:

* File.ReadAllText – читает целый файл и возвращает его содержимое в виде одной строки;
* File.ReadAllLines – читает целый файл и возвращает его содержимое в виде массива строк;
* File.ReadAllBytes – читает целый файл и возвращает его содержимое в виде байтового массива;
* File.WriteAllText – записывает строку в файл;
* File.WriteAllLines – записывает массив строк в файл;
* File.WriteAllBytes – записывает байтовый массив в файл;
* File.AppendAllText – добавляет строку в конец файла.

Метод File.ReadLines не загружает весь файл в память, а возвращает лениво-оцениваемое IEnumerable<string>:

int longLines = File.ReadLines ("filePath").Count (l => l.Length > 80);

**Имя файла**

Имя файла может быть абсолютным или относительным к текущему каталогу. Узнать текущий каталог или изменить его можно с помощью статического свойства Environment.CurrentDirectory. Текущий каталог может как совпадать, так и не совпадать с директорией, в которой расположен исполняемый файл программы.

Свойство AppDomain.CurrentDomain.BaseDirectory возвращает базовый каталог приложения, который как правило совпадает с директорией, содержащей исполняемый файл программы.

Чтобы указать имя файла относительно полученного каталога, можно использовать метод Path.Combine:

string baseFolder = AppDomain.CurrentDomain.BaseDirectory;

string logoPath = Path.Combine (baseFolder, "logo.jpg");

Console.WriteLine (File.Exists (logoPath));

Допустимо использовать сетевые имена, такие как \\JoesPC\PicShare\pic.jpg или \\10.1.1.2\PicShare\pic.jpg.

**Режимы файла (FileMode)**

Все конструкторы FileStream помимо имени файла требуют указания режима файла — значения enum FileMode:

* FileMode.CreateNew — будет создан новый файл, доступный для чтения и записи; если файл с таким именем уже существует, будет выброшено исключение
* FileMode.Create — тоже самое, что и FileMode.CreateNew, но если файл с таким именем уже существует, он будет перезаписан новым (содержимое существующего файла будет обнулено)
* FileMode.OpenOrCreate — если файл существует — он будет открыт для чтения и записи, если не существует — будет создан новый (доступный для чтения и записи)
* FileMode.Open — файл открывается для чтения и записи; если файла с таким именем не существует, будет выброшено исключение
* FileMode.Truncate — тоже самое, что и FileMode.Open, но содержимое открытого файла обнуляется
* FileMode.Append — файл открывается только для записи; запись производится в конец файла

Метод File.Create и значение FileMode.Create приведут к генерации исключения, если используются для скрытых файлов. Чтобы перезаписать скрытый файл, его придется сначала удалить и создать заново.

Во всех случаях, кроме FileMode.Append, поток открывается для чтения и записи. Можно ограничить уровень доступа, если в качестве дополнительного параметра передать enum FileAccess:

[Flags]

public enum FileAccess { Read = 1, Write = 2, ReadWrite = 3 }

При создании экземпляра FileStream можно также указывать следующие дополнительные аргументы:

* enum FileShare — уровень доступа для других процессов, пока мы работает с файлом (None, Read (по умолчанию), ReadWrite или Write)
* размер внутреннего буфера в байтах
* флаг, указывающий следует ли отложить асинхронный вывод для ОС
* объект FileSecurity, описывающий права доступа для пользователей для создаваемого файла
* enum FileOptions:
* FileOptions.Encrypted — включает шифрование на уровне ОС
* FileOptions.DeleteOnClose — автоматическое удаление при закрытии временных файлов
* FileOptions.RandomAccess и FileOptions.SequentialScan
* FileOptions.WriteThrough — отключение кэширования при записи

Если файл открыт с ключом FileShare.ReadWrite позволяет другим процессам читать и записывать файл. С помощью методов класса FileStream Lock и Unlock можно блокировать и разблокировать участки файла:

public virtual void Lock (long position, long length);

public virtual void Unlock (long position, long length);

Если часть или вся запрошенная область файла уже заблокированы метод Lock выбросит исключение.

**MemoryStream**

Класс MemoryStream в качестве опорного хранилища использует массив (целиком расположенный в памяти).

С помощью методов ToArray и GetBuffer можно преобразовать MemoryStream в байтовый массив.

Закрытие и сброс MemoryStream являются не обязательными. После закрытия выполнять чтение и запись потока нельзя. Метод Flush потока MemoryStream вообще ничего не делает.

**BufferedStream**

Класс BufferedStream является декоратором, он помещает другой поток в оболочку и добавляет ему возможность буферизации. Буферизация улучшает производительность, сокращая количество двусторонних обменов с опорным хранилищем.

// Записать 100K в файл:

File.WriteAllBytes ("myFile.bin", new byte [100000]);

using (FileStream fs = File.OpenRead ("myFile.bin"))

using (BufferedStream bs = new BufferedStream (fs, 20000)) // буфер размером 20K

{

bs.ReadByte();

Console.WriteLine (fs.Position); // 20000

}

В примере поток FileStream помещается в декоратор BufferedStream с буфером в 20Kb. При вызова ReadByte фактически из потока FileStream считывается не один байт, а 20 000 байтов (размер буфера), которые помещаются в буфер. При следующих 19 999 вызовах метода ReadByte данные будут читаться из буфера, а не из файлового потока. На практике соединение FileStream с BufferedStream избыточно, так как FileStream сам поддерживает буферизацию.

Закрытие BufferedStream автоматически закрывает лежащий в основе поток с опорным хранилищем.

**Текстовые адаптеры**

**TextReader и TextWriter**

TextReader и TextWriter являются абстрактными базовыми классами для адаптеров, которые имеют дело исключительно с символами и строками. Каждый из них имеет две реализации:

* StreamReader и StreamWriter — для хранения данных используют классStream и транслируют байты потока в символы или строки
* StringReader и StringWriter — для хранения данных используют строки

**Члены класса TextReader:**

// Чтение одного символа:

public virtual int Peek(); // Результат приводится к char

public virtual int Read(); // Результат приводится к char

// Чтение нескольких символов:

public virtual int Read(char[] buffer, int index, int count);

public virtual int ReadBlock(char[] buffer, int index, int count);

public virtual string ReadLine();

public virtual string ReadToEnd();

// Закрытие:

public virtual void Close();

public void Dispose();

// Другие:

public static readonly TextReader Null;

public static TextReader Synchronized (TextReader reader);

Метод Peek возвращает следующий символ из потока, не перемещая текущую позицию вперед. Метод Peek и Read (без аргументов) возвращают -1 при достижении конца потока или целочисленное значение, которое может быть приведено к char.

Перегруженная версия Read, принимающая буфер char[], идентична по функционалу методу ReadBlock.

Метод ReadLine выполняет чтение до тех пор, пока не встретит символы новой строки (CR, LF либо CR+LF). Он возвращает строку, отбрасывая символы новой строки.

Члены класса TextWriter:

// Запись одного символа:

public virtual void Write (char value);

// Запись нескольких символов:

public virtual void Write (string value);

public virtual void Write (char[] buffer, int index, int count);

public virtual void Write (string format, params object[] arg);

public virtual void WriteLine (string value);

// Закрытие и сброс:

public virtual void Close ();

public void Dispose ();

public virtual void Flush ();

// Форматирование и кодировка:

public virtual IFormatProvider FormatProvider { get; }

public virtual string NewLine { get; set; }

public abstract Encoding Encoding { get; }

// Другие:

public static readonly TextWriter Null;

public static TextWriter Synchronized (TextWriter writer);

Методы Write и WriteLine дополнительно перегружены, чтобы принимать каждый из примитивных типов, а также тип object (они просто вызывают ToString на том, что им передано).

Метод WriteLine дополняет переданную ему строку последовательностью CR+LF. Свойство NewLine позволяет заменить эту последовательность на другие символы.

Классы TextReader и TextWriter также определяют асинхронные версии методов чтения и записи.

**StreamReader и StreamWriter**

Сами по себе классы TextReader и TextWriter являются абстрактными и никак не связаны ни с потоком, ни с опорным хранилищем. А вот реализующие их типы StreamReader и StreamWriter имеют в своей основе байтовый поток и выполняют преобразование между символами и байтами.

Для создания экземпляра StreamReader или StreamWriter их конструкторам необходимо передать байтовый поток:

using (FileStream fs = File.Create ("test.txt"))

using (TextWriter writer = new StreamWriter (fs))

{

writer.WriteLine ("Line1");

writer.WriteLine ("Line2");

}

using (FileStream fs = File.OpenRead ("test.txt"))

using (TextReader reader = new StreamReader (fs))

{

Console.WriteLine (reader.ReadLine());

Console.WriteLine (reader.ReadLine());

}

Класс File предоставляет статические методы CreateText, AppendText и OpenText, которые позволяют создать экземпляр StreamReader или StreamWriter более лаконично:

using (TextWriter writer = File.CreateText ("test.txt"))

{

writer.WriteLine ("Line1");

writer.WriteLine ("Line2");

}

using (TextWriter writer = File.AppendText ("test.txt"))

writer.WriteLine ("Line3");

using (TextReader reader = File.OpenText ("test.txt"))

while (reader.Peek() > −1)

Console.WriteLine (reader.ReadLine());

Достижение конца файла можно проверить с помощью метода Peek (вернет -1) или метода ReadLine (вернет null).

Преобразование байтов потока в строку выполняется с помощью класса System.Text.Encoding, который можно передать в конструктор StreamReader или StreamWriter. По умолчанию используется кодировка UTF-8.

using (Stream s = File.Create ("but.txt"))

using (TextWriter w = new StreamWriter (s, Encoding.Unicode))

w.WriteLine ("but-");

**StringReader и StringWriter**

Адаптеры StringReader и StringWriter вообще не имеют дела с потоками. Вместо этого они используют в качестве лежащего в основе источника данных строку или экземпляр StringBuilder. Они не выполняют никаких преобразований байтов в строку и обратно, по сути они не делают ничего такого, что нельзя было бы сделать с помощью строк и StringBuilder. Преимущество же их в том, что они имеют общий базовый класс с типами StreamReader и StreamWriter, облегчая их преобразования.

**Двоичные адаптеры**

Классы BinaryReader и BinaryWriter выполняют чтение и запись в поток предопределенных типов: bool, byte, char, decimal, float, double, short, int,long, sbyte, ushort, uint и ulong, а также строк и массивов предопределенных типов. В отличие от текстовых адаптеров двоичные адаптеры сохраняют предопределенные символы эффективнее, так как они представлены в памяти.

public class Person

{

public string Name;

public int Age;

public double Height;

public void SaveData (Stream s)

{

var w = new BinaryWriter (s);

w.Write (Name);

w.Write (Age);

w.Write (Height);

w.Flush();

}

public void LoadData (Stream s)

{

var r = new BinaryReader (s);

Name = r.ReadString();

Age = r.ReadInt32();

Height = r.ReadDouble();

}

}

Класс BinaryReader может также выполнять чтение в байтовый массив:

byte[] data = new BinaryReader(s).ReadBytes ((int) s.Length);

**Закрытие адаптеров потока**

Закрытие адаптера приводит к автоматическому закрытию лежащего в основе потока:

using (FileStream fs = File.Create ("test.txt"))

using (TextWriter writer = new StreamWriter (fs))

writer.WriteLine ("Line");

При этом если в конструкторе адаптера будет выброшено исключение, поток все равно закроется.

При закрытии адаптера и потока без использования инструкции using, нужно всегда сначала закрывать или сбрасывать адаптер, и только после этого закрывать поток, иначе любые данные, буферизированные в адаптере, будут утеряны.

Адаптеры относятся к необязательно освобождаемым объектам. Это означает, что их необязательно закрывать перед закрытием потока. В большинстве случаев их достаточно просто сбросить. Это может быть удобно в ситуации, когда после завершения работы с адаптером лежащий в его основе поток должен остаться для дальнейшей работы с ним:

using (FileStream fs = new FileStream ("test.txt", FileMode.Create))

{

StreamWriter writer = new StreamWriter (fs);

writer.WriteLine ("Hello");

writer.Flush();

fs.Position = 0;

Console.WriteLine (fs.ReadByte());

}

В примере если вместо сброса адаптера закрыть его, лежащий в его основе поток тоже будет закрыт, что приведет к сбою последующих операций над потоком.

Если конструкторы адаптера StreamReader/StreamWriter в качестве четвертого параметра передать true, то после его освобождения лежащий в основе поток закрыт не будет:

using (var fs = new FileStream ("test.txt", FileMode.Create))

{

using (var writer = new StreamWriter (fs,

new UTF8Encoding (false, true),

0x400,

true))

writer.WriteLine ("Hello");

fs.Position = 0;

Console.WriteLine (fs.ReadByte());

Console.WriteLine (fs.Length);

}

**Сжатие потоков**

В пространстве имен System.IO.Compression доступно два декоратора для сжатия потоков: DeflateStream и GZipStream. Отличаются они тем, что GZipStreamзаписывает дополнительную информацию в начале и в конце, а также соответствует стандарту. Оба класса являются декораторами: они сжимают или распаковывают данные из другого потока, который указывается при создании их экземпляра:

using (Stream s = File.Create ("compressed.bin"))

using (Stream ds = new DeflateStream (s, CompressionMode.Compress))

for (byte i = 0; i < 100; i++)

ds.WriteByte (i);

using (Stream s = File.OpenRead ("compressed.bin"))

using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))

for (byte i = 0; i < 100; i++)

Console.WriteLine (ds.ReadByte());

В этом же пространстве имен определяются еще два класса: ZipArchive и ZipFile, которые использую популярный алгоритм сжатия, применяемый в zip-файлах, что делает их совместимыми с zip-файлами, созданными в других приложениях, а также позволяет сжимать несколько файлов в один архив.

Класс ZipArchive работает непосредственно с потоками, а ZipFile является статическим вспомогательным классом для него. ZipFile более удобен в использовании при работе с файлами.

Метод CreateFromDirectory класса ZipFile добавляет все файлы из указанного каталога в zip-архив:

ZipFile.CreateFromDirectory (@"d:\MyFolder", @"d:\compressed.zip");

Метод ExtractToDirectory того же класса извлекает содержимое zip-архива в указанный каталог:

ZipFile.ExtractToDirectory (@"d:\compressed.zip", @"d:\MyFolder");

При сжатии можно задать оптимизацию по размеру файла или по скорости сжатия, а также необходимость включения в архив исходных директории.

Экземпляр ZipArchive можно создать либо с помощью конструктора, передав ему поток (объект Stream), либо с помощью статического метода Open класс ZipFile, передав ему имя файла и действие, которое должно быть произведено над архивом — Read (чтение), Create (создание) или Update (обновление). Свойство Entries объекта ZipArchive возвращает коллекцию входящих в архив файлов, а метод GetEntry позволяет найти конкретный файл:

using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Read))

foreach (ZipArchiveEntry entry in zip.Entries)

Console.WriteLine (entry.FullName + " " + entry.Length);

Класс ZipArchiveEntry инкапсулирует отдельный файл в архиве. Он имеет методыDelete (позволяет удалить файл из архива), ExtractToFile (позволяет извлечь файл из архива) и Open (возвращает экземпляр Stream, с возможностью чтения/записи). Создавать новые файлы в архиве можно с помощью методов CreateEntry иCreateEntryFromFile класса ZipArchive.

byte[] data = File.ReadAllBytes (@"d:\foo.dll");

using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Update))

zip.CreateEntry (@"bin\X64\foo.dll").Open().Write (data, 0, data.Length);

**Манипулирование файлами и каталогами**

Пространство имен System.IO содержит ряд типов, предназначенных для манипулирования файлами и каталогами, позволяющих копировать, перемещать, создавать файлы и каталоги, устанавливать их атрибуты и права доступа. Сюда входят статические классы File и Directory, экземплярные классы FileInfo иDirectoryInfo, а также статический класс Path, позволяющий манипулировать строками путей к файлам и каталогам.

**Класс File**

File — статический класс, все методы которого принимают имя файла. Имя файла может быть абсолютным или относительным (относительно текущего каталога). Класс включает следующие статические методы:

bool Exists (string path); // Возвращает true если файл существует

void Delete (string path);

void Copy (string sourceFileName, string destFileName);

void Move (string sourceFileName, string destFileName);

void Replace (string sourceFileName, string destinationFileName,

string destinationBackupFileName);

FileAttributes GetAttributes (string path);

void SetAttributes (string path, FileAttributes fileAttributes);

void Decrypt (string path);

void Encrypt (string path);

DateTime GetCreationTime (string path);

DateTime GetLastAccessTime (string path);

DateTime GetLastWriteTime (string path);

void SetCreationTime (string path, DateTime creationTime);

void SetLastAccessTime (string path, DateTime lastAccessTime);

void SetLastWriteTime (string path, DateTime lastWriteTime);

FileSecurity GetAccessControl (string path);

FileSecurity GetAccessControl (string path,

AccessControlSections includeSections);

void SetAccessControl (string path, FileSecurity fileSecurity);

Методы Move и Replace позволяют переименовать файл или переместить его в другой каталог, при этом Move генерирует исключение, если файл назначения существует, а Replace этого не делает.

Метод Delete удаляет файл, а если файл помечен как предназначенный только для чтения метод сгенерирует исключение UnauthorizedAccessException.

Метод GetAttributes возвращает enum FileAttribute со следующими значениями, которые можно комбинировать:

* Archive
* Compressed
* Device
* Directory
* Encrypted
* Hidden
* Normal
* NotContentIndexed
* Offline
* ReadOnly
* ReparsePoint
* SparseFile
* System
* Temporary

С помощью метода SetAttributes атрибуты файла можно менять:

string filePath = @"c:\temp\test.txt";

FileAttributes fa = File.GetAttributes (filePath);

if ((fa & FileAttributes.ReadOnly) > 0)

{

fa ^= FileAttributes.ReadOnly;

File.SetAttributes (filePath, fa);

}

// теперь файл можно, например, удалить

File.Delete (filePath);

С помощью класса FileInfo это можно сделать лаконичней:

new FileInfo (@"c:\temp\test.txt").IsReadOnly = false;

Атрибуты Compressed и Encrypted (сжатие и шифрование) с помощью методаSetAttribute изменить нельзя. Для шифрования и дешифрования предназначены методы Encrypt и Decrypt класса File. Для сжатия класс File методов не содержит.

Методы GetAccessControl и SetAccessControl позволяют получать и задавать права доступа ОС через объект FileSecurity (пространство имен System.Security.AccessControl). Этот объект также можно передать в конструктор FileStream для указания прав доступа при создании файла.

**Класс Directory**

Статический класс Directory содержит методы аналогичные методам класса File: для проверки существования каталога (Exists), перемещения каталога (Move), удаления каталога (Delete), получения/установки времени создания и последнего доступа, получения/установки прав доступа. Кроме того Directory включает следующие статические методы:

string GetCurrentDirectory();

void SetCurrentDirectory(string path);

DirectoryInfo CreateDirectory(string path);

DirectoryInfo GetParent(string path);

string GetDirectoryRoot(string path);

string[] GetLogicalDrives();

// Все следующие методы возвращают полный путь:

string[] GetFiles(string path);

string[] GetDirectories(string path);

string[] GetFileSystemEntries(string path);

IEnumerable<string> EnumerateFiles(string path);

IEnumerable<string> EnumerateDirectories(string path);

IEnumerable<string> EnumerateFileSystemEntries(string path);

Методы Enumerate\* эффективней методов Get\*, поскольку извлекают данные только при перечислении. И те и другие перегружены и могут принимать searchPattern(string) и searchOption (enum). При указании SearchOption.SearchAllSubDirectories также будет выполняться рекурсивный поиск в подкаталогах. Методы \*FileSystemEntries сочетают в себе функционал методов \*Files и \*Directories.

**FileInfo и DirectoryInfo**

Если над файлом или каталогом требуется выполнить последовательность операций, то более удобными будут экземплярные методы FileInfo и DirectoryInfo.

Класс FileInfo предлагает большинство статических методов класс File в форме экземплярных методов, а также содержит ряд дополнительных свойств — Extension,Length, IsReadOnly и Directory. Последний возвращает объект DirectoryInfo.

FileInfo fi = new FileInfo (@"c:\temp\FileInfo.txt");

Console.WriteLine (fi.Exists); // false

using (TextWriter w = fi.CreateText())

w.Write ("Some text");

Console.WriteLine (fi.Exists); // по прежнему false

fi.Refresh();

Console.WriteLine (fi.Exists); // true

Console.WriteLine (fi.Name); // FileInfo.txt

Console.WriteLine (fi.FullName); // c:\temp\FileInfo.txt

Console.WriteLine (fi.DirectoryName); // c:\temp

Console.WriteLine (fi.Directory.Name); // temp

Console.WriteLine (fi.Extension); // .txt

Console.WriteLine (fi.Length); // 9

fi.Encrypt();

fi.Attributes ^= FileAttributes.Hidden; // Переключает флаг "скрытый"

fi.IsReadOnly = true;

Console.WriteLine (fi.Attributes); // ReadOnly, Archive, Hidden, Encrypted

Console.WriteLine (fi.CreationTime);

fi.MoveTo (@"c:\temp\FileInfoX.txt");

DirectoryInfo di = fi.Directory;

Console.WriteLine (di.Name); // temp

Console.WriteLine (di.FullName); // c:\temp

Console.WriteLine (di.Parent.FullName); // c:\

di.CreateSubdirectory ("SubFolder");

Пример использования класса DirectoryInfo:

DirectoryInfo di = new DirectoryInfo (@"e:\photos");

foreach (FileInfo fi in di.GetFiles ("\*.jpg"))

Console.WriteLine (fi.Name);

foreach (DirectoryInfo subDir in di.GetDirectories())

Console.WriteLine (subDir.FullName);

**Path**

Статический класс Path содержит методы и поля для работы с путями и именами файлов:

string dir = @"c:\mydir";

string file = "myfile.txt";

string path = @"c:\mydir\myfile.txt";

Directory.SetCurrentDirectory (@"k:\demo");

// Методы Path:

Path.IsPathRooted (file) // False

Path.IsPathRooted (path) // True

Path.GetPathRoot (path) // c:\

Path.GetDirectoryName (path) // c:\mydir

Path.GetFileName (path) // myfile.txt

Path.GetFullPath (file) // k:\demo\myfile.txt

Path.Combine (dir, file) // c:\mydir\myfile.txt

// Расширения файлов:

Path.HasExtension (file) // True

Path.GetExtension (file) // .txt

Path.GetFileNameWithoutExtension (file) // myfile

Path.ChangeExtension (file, ".log") // myfile.log

// Разделители и символы:

Path.AltDirectorySeparatorChar // /

Path.PathSeparator // ;

Path.VolumeSeparatorChar // :

Path.GetInvalidPathChars() // символы от 0 до 31 и "<>|

Path.GetInvalidFileNameChars() // символы от 0 до 31 и "<>|:\*?\/

// Временные файлы:

Path.GetTempPath() // <local user folder>\Temp

Path.GetRandomFileName() // d2dwuzjf.dnp

Path.GetTempFileName() // <local user folder>\Temp\tmp14B.tmp

Метод Combine объединяет каталог и имя файла, при необходимости дополняя имя каталога косой чертой. Метод GetFullPath преобразует относительный путь в абсолютный. Метод GetRandomFileName возвращает уникальное случайное имя файла в формате 8.3, не создавая при этом файл. Метод GetTempFileName генерирует временное имя файла с применением автоинкрементного счетчика и затем создает пустой файл с этим именем в локальном временном каталоге.

**Специальные директории**

В перечисленных выше классах отсутствуют средства нахождения специальных директорий Windows, таких как My Documents, Program Files и др. Сделать это можно с помощью метода GetFolderPath класса System.Environment:

string myDocPath = Environment.GetFolderPath

(Environment.SpecialFolder.MyDocuments);

Значения enum Environment.SpecialFolder охватывают все специальные каталоги в Windows:

* AdminTools
* CommonVideos
* Personal
* ApplicationData
* Cookies
* PrinterShortcuts
* CDBurning
* Desktop
* ProgramFiles
* CommonAdminTools
* DesktopDirectory
* ProgramFilesX86
* CommonApplicationData
* Favorites
* Programs
* CommonDesktopDirectory
* Fonts
* Recent
* CommonDocuments
* History
* Resources
* CommonMusic
* InternetCache
* SendTo
* CommonOemLinks
* LocalApplicationData
* StartMenu
* CommonPictures
* LocalizedResources
* Startup
* CommonProgramFiles
* MyComputer
* System
* CommonProgramFilesX86
* MyDocuments
* SystemX86
* CommonPrograms
* MyMusic
* Templates
* CommonStartMenu
* MyPictures
* UserProfile
* CommonStartup
* MyVideos
* Windows
* CommonTemplates
* NetworkShortcuts

Использование папок ApplicationData (настройки, перемещаемые с пользователем по сети), LocalApplicationData (неперемещаемые настройки пользователя), CommonApplicationData (настройки для всех пользователей компьютера) для сохранения данных и настроек приложения более предпочтительно, чем использование системного реестра. Как правило, в этих директориях создаются подкаталоги с именем, совпадающим с названием приложения.

**Информация о диске**

Запрашивать информацию об устройствах на компьютере можно с помощью класса DriveInfo:

DriveInfo c = new DriveInfo ("C"); // Запросить диск C:

long totalSize = c.TotalSize; // Размер в байтах

long freeBytes = c.TotalFreeSpace; // Игнорирует дисковую квоту

long freeToMe = c.AvailableFreeSpace; // Учитывает дисковую квоту

foreach (DriveInfo d in DriveInfo.GetDrives()) // Все устройства хранения данных

{

Console.WriteLine (d.Name); // C:\

Console.WriteLine (d.DriveType); // Жесткий диск

Console.WriteLine (d.RootDirectory); // C:\

if (d.IsReady) // Если устройство не готово следующие свойства выбросят исключение

{

Console.WriteLine (d.VolumeLabel); // System

Console.WriteLine (d.DriveFormat); // NTFS

}

}

enum DriveType содержит следующие значения:

* Unknown
* NoRootDirectory
* Removable
* Fixed
* Network
* CDRom
* Ram

**Перехват событий файловой системы**

Класс FileSystemWatcher позволяет отслеживать действия над каталогами и их подкаталогами: создание, модификация, переименование, удаление файлов и поддиректорий, а также изменение их атрибутов. Событие генерируется независимо от того, кем оно совершено — пользователем или процессом:

static void Main() { Watch (@"c:\temp", "\*.txt", true); }

static void Watch (string path, string filter, bool includeSubDirs)

{

using (var watcher = new FileSystemWatcher (path, filter))

{

watcher.Created += FileCreatedChangedDeleted;

watcher.Changed += FileCreatedChangedDeleted;

watcher.Deleted += FileCreatedChangedDeleted;

watcher.Renamed += FileRenamed;

watcher.Error += FileError;

watcher.IncludeSubdirectories = includeSubDirs;

watcher.EnableRaisingEvents = true;

Console.WriteLine ("Listening for events - press <enter> to end");

Console.ReadLine();

}

}

static void FileCreatedChangedDeleted (object o, FileSystemEventArgs e)

{

Console.WriteLine ("File {0} has been {1}", e.FullPath, e.ChangeType);

}

static void FileRenamed (object o, RenamedEventArgs e)

{

Console.WriteLine ("Renamed: {0}->{1}", e.OldFullPath, e.FullPath);

}

static void FileError (object o, ErrorEventArgs e)

{

Console.WriteLine ("Error: " + e.GetException().Message);

}

**Сериализация и десериализация объектов**

**Сериализация и десериализация в формате XML**

XML – расширяемый язык разметки. Рекомендован Консорциумом Всемирной паутины (W3C). Спецификация XML описывает XML-документы и частично описывает поведение XML-процессоров (программ, читающих XML-документы и обеспечивающих доступ к их содержимому). XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком, с подчёркиванием нацеленности на использование в Интернете. Язык называется расширяемым, поскольку он не фиксирует разметку, используемую в документах: разработчик волен создать разметку в соответствии с потребностями к конкретной области, будучи ограниченным лишь синтаксическими правилами языка. Расширение XML — это конкретная грамматика, созданная на базе XML и представленная словарём тегов и их атрибутов, а также набором правил, определяющих какие атрибуты и элементы могут входить в состав других элементов. Если бы HTML появился позднее, чем XML, то он, возможно, также бы являлся его расширением. Сочетание простого формального синтаксиса, удобства для человека, расширяемости, а также базирование на кодировках Юникод для представления содержания документов привело к широкому использованию как собственно XML, так и множества производных специализированных языков на базе XML в самых разнообразных программных средствах.

Пример файла XML:

<?xml version="1.0" encoding="koi-8"?>

<notepad>

<note id="1" date="12/04/99" time="13:40">

<subject>Важная деловая встреча</subject>

<importance/>

<text>

Надо встретиться с <person id="1625">Иваном Ивановичем</person>,

предварительно позвонив ему по телефону <tel>123-12-12</tel>

</text>

</note>

...

<note id="2" date="12/04/99" time="13:58">

<subject>Позвонить домой</subject>

<text>

<tel>124-13-13</tel>

</text>

</note>

</notepad>

Для сериализации объектов в файлы XML используется класс XmlSerializer. XmlSerializer предполагает некоторые ограничения. Например, класс, подлежащий сериализации, должен иметь стандартный конструктор без параметров. Также сериализации подлежат только открытые члены. Если в классе есть поля или свойства с модификатором private, то при сериализации они будут игнорироваться.

XmlSerializer требует указания типа:

using System;

using System.IO;

using System.Xml.Serialization;

namespace Serialization

{

    // класс и его члены объявлены как public

    [Serializable]

    public class Person

    {

        public string Name { get; set; }

        public int Age { get; set; }

        // стандартный конструктор без параметров

        public Person()

        { }

        public Person(string name, int age)

        {

            Name = name;

            Age = age;

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            // объект для сериализации

            Person person = new Person("Tom", 29);

            Console.WriteLine("Объект создан");

            // передаем в конструктор тип класса

            XmlSerializer formatter = new XmlSerializer(typeof(Person));

            // получаем поток, куда будем записывать сериализованный объект

            using (FileStream fs = new FileStream("persons.xml", FileMode.OpenOrCreate))

            {

               formatter.Serialize(fs, person);

                Console.WriteLine("Объект сериализован");

            }

            // десериализация

            using (FileStream fs = new FileStream("persons.xml", FileMode.OpenOrCreate))

            {

                Person newPerson = (Person)formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");

                Console.WriteLine("Имя: {0} --- Возраст: {1}", newPerson.Name, newPerson.Age);

            }

            Console.ReadLine();

        }

    }

}

Класс Person общедоступный и имеет общедоступные свойства, поэтому он может сериализоваться. При создании объекта XmlSerializer передаем в конструктор тип класса.

И, как и с другими классами-сериализаторами, метод Serialize добавляет данные в файл persons.xml. А метод Deserialize извлекает их оттуда.

Если мы откроем файл persons.xml, то увидим содержание нашего объекта:

<?xml version="1.0"?>

<Person xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns:xsd="<http://www.w3.org/2001/XMLSchema>">

  <Name>Tom</Name>

  <Age>29</Age>

</Person>

**Сериализация и десериализация в формате JSON**

JSON — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми.

Несмотря на происхождение от JavaScript (точнее, от подмножества языка стандарта ECMA-262 1999 года), формат считается независимым от языка и может использоваться практически с любым языком программирования. Для многих языков существует готовый код для создания и обработки данных в формате JSON.

За счёт своей лаконичности по сравнению с XML, формат JSON может быть более подходящим для сериализации сложных структур. Если говорить о веб-приложениях, в таком ключе он уместен в задачах обмена данными как между браузером и сервером (AJAX), так и между самими серверами (программные HTTP-сопряжения).

Поскольку формат JSON является подмножеством синтаксиса языка JavaScript, то он может быть быстро десериализован встроенной функцией eval(). Кроме того, возможна вставка вполне работоспособных JavaScript-функций. В языке PHP, начиная с версии 5.2.0, поддержка JSON включена в ядро в виде функций json\_decode() и json\_encode(), которые сами преобразуют типы данных JSON в соответствующие типы PHP и наоборот.

JSON-текст представляет собой (в закодированном виде) одну из двух структур:

* Набор пар ключ: значение. В различных языках это реализовано как объект, запись, структура, словарь, хэш-таблица, список с ключом или ассоциативный массив. Ключом может быть только строка (регистрозависимая: имена с буквами в разных регистрах считаются разными), значением — любая форма.
* Упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

Это универсальные структуры данных: как правило, любой современный язык программирования поддерживает их в той или иной форме. Они легли в основу JSON, так как он используется для обмена данными между различными языками программирования.

В качестве значений в JSON могут быть использованы:

* Объект — это неупорядоченное множество пар ключ:значение, заключённое в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары ключ-значение отделяются друг от друга запятыми.
* Массив (одномерный) — это упорядоченное множество значений. Массив заключается в квадратные скобки «[ ]». Значения разделяются запятыми.
* Число.
* Литералы true, false и null.
* Строка — это упорядоченное множество из нуля или более символов юникода, заключённое в двойные кавычки. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты «\» (поддерживаются варианты \", \\, \/, \t, \n, \r, \f и \b), или записаны шестнадцатеричным кодом в кодировке Unicode в виде \uFFFF.

Строка очень похожа на одноимённый тип данных в языках С и Java. Число тоже очень похоже на С- или Java-число, за исключением того, что используется только десятичный формат. Пробелы могут быть вставлены между любыми двумя синтаксическими элементами.

Следующий пример показывает JSON-представление объекта, описывающего человека. В объекте есть строковые поля имени и фамилии, объект, описывающий адрес, и массив, содержащий список телефонов. Как видно из примера, значение может представлять собой вложенную структуру.

{

"firstName": "Иван",

"lastName": "Иванов",

"address": {

"streetAddress": "Московское ш., 101, кв.101",

"city": "Ленинград",

"postalCode": 101101

},

"phoneNumbers": [

"812 123-1234",

"916 123-4567"

]

}

На языке XML подобная структура выглядела бы примерно так:

<person>

<firstName>Иван</firstName>

<lastName>Иванов</lastName>

<address>

<streetAddress>Московское ш., 101, кв.101</streetAddress>

<city>Ленинград</city>

<postalCode>101101</postalCode>

</address>

<phoneNumbers>

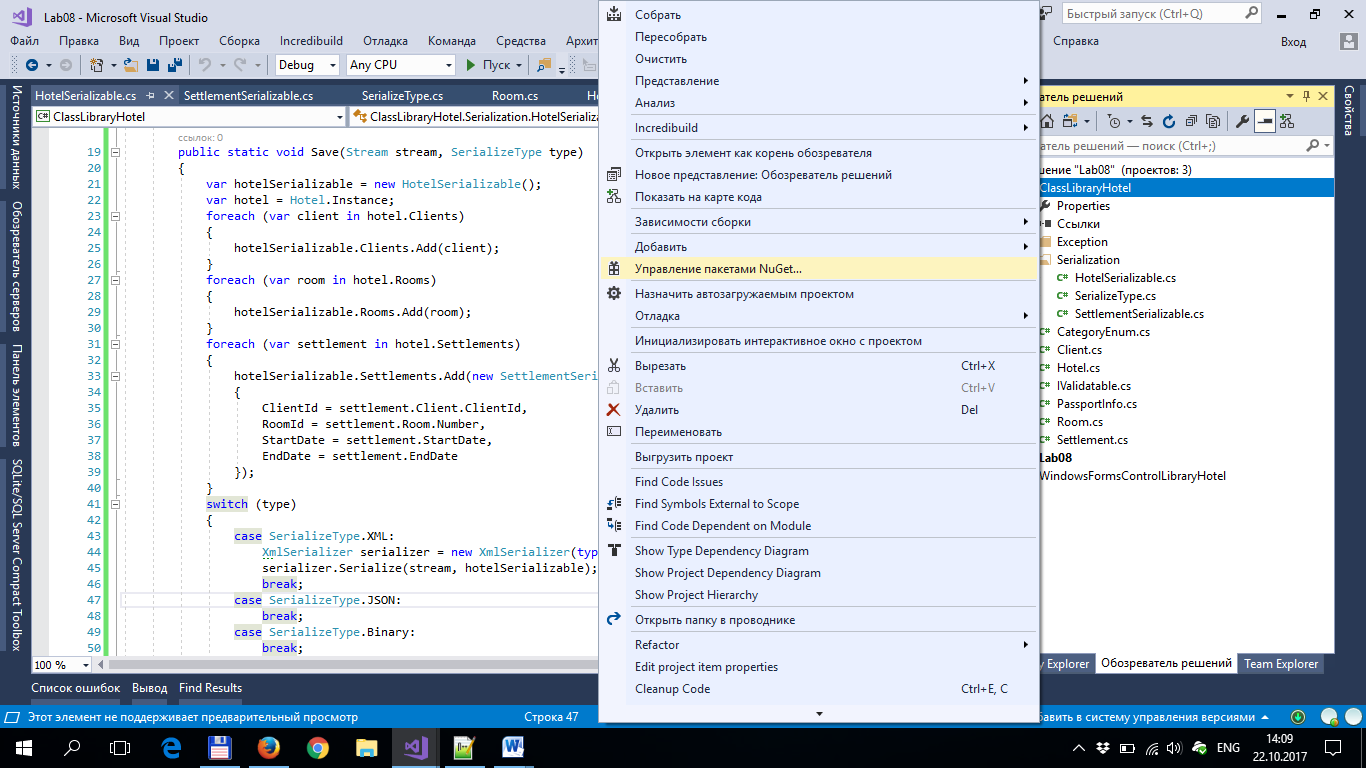
<phoneNumber>812 123-1234</phoneNumber>

<phoneNumber>916 123-4567</phoneNumber>

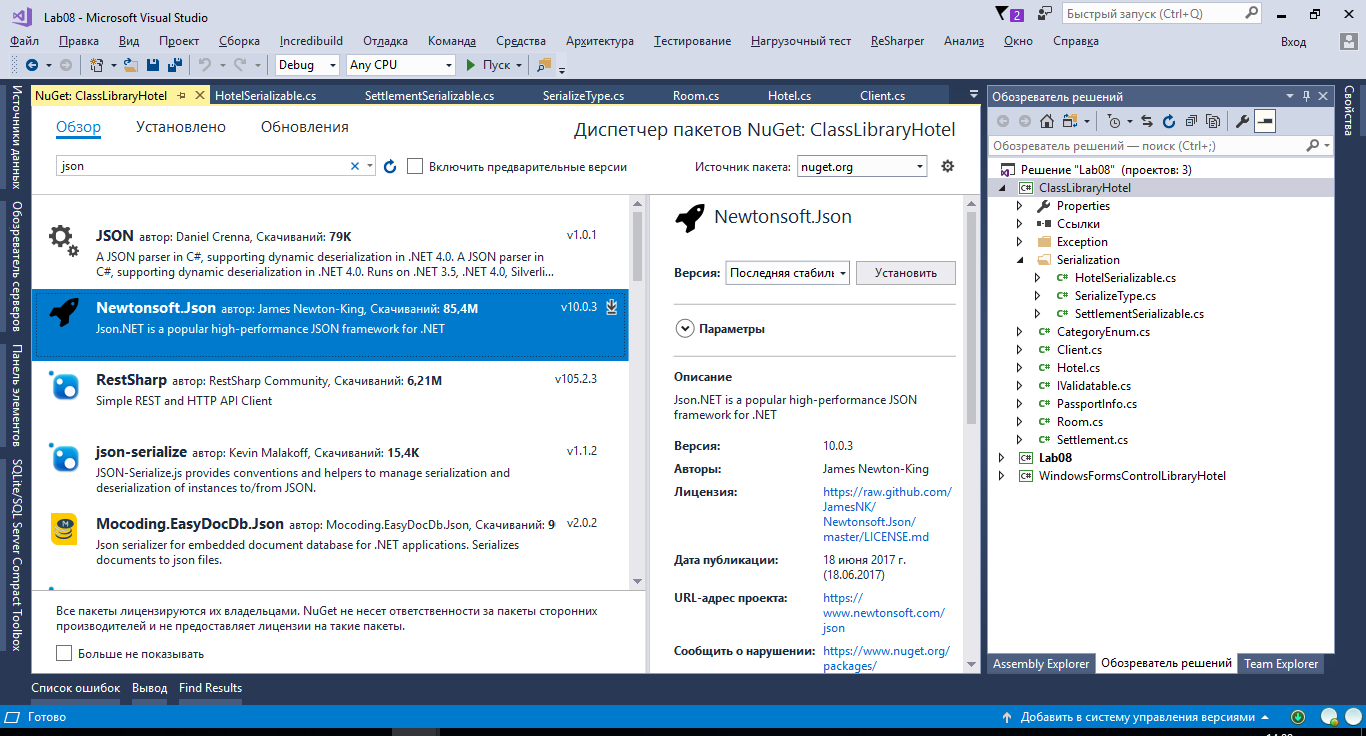
</phoneNumbers>

</person>

Для поддержки JSON-сериализации в проектах Microsoft .Net воспользуемся библиотекой Json.NET. Добавить эту библиотеку в проект можно через с помощью пакетов Nuget (рис. 1-2).



***Рис. 1. Вызов окна управления пакетами Nuget***



***Рис. 2. Добавление пакета Newtonsoft.Json в проект***

В простейшем случае сериализация осуществляется следующим образом:

using (StreamWriter jsonStreamWriter = File.CreateText(fileName))

{

JsonSerializer jsonSerializer = new JsonSerializer { Formatting = Formatting.Indented };

jsonSerializer.Serialize(jsonStreamWriter, hotelSerializable);

}

Десериализация объекта из файла осуществляется следующим образом:

StreamReader jsonStreamReader = File.OpenText(fileName);

JsonSerializer jsonSerializer = new JsonSerializer();

hotelSerializable = (HotelSerializable)jsonSerializer.Deserialize(jsonStreamReader, typeof(HotelSerializable));

**Двоичная сериализация и десериализация**

Чтобы сделать объект доступным для служб сериализации .NET, все, что понадобится — это декорировать каждый связанный класс (или структуру) атрибутом [Serializable].

Если вы определили, что некоторый тип имеет члены-данные, которые не должны (или не могут) участвовать в схеме сериализации, можете пометить такие поля атрибутом [NonSerialized]. Это пригодится, если в сериализуемом классе есть переменные-члены, которые не следует «запоминать» (например, фиксированные значения, случайные значения, кратковременные данные и т.п.), и планируется сократить размер хранимых данных.

Атрибут [Serializable] не может наследоваться от родительского класса. Поэтому, если вы наследуете тип, помеченный [Serializable], то дочерний класс также должен быть помечен [Serializable] или же его нельзя будет сохранить в потоке. Фактически все объекты в графе объектов должны быть помечены атрибутом [Serializable].

Тип BinaryFormatter сериализует состояние вашего объекта в поток, используя компактный двоичный формат. Этот тип определен в пространстве имен System. Runtime.Serialization.Formatters.Binary, которое входит в сборку mscorlib.dll. Таким образом, если вы хотите получить доступ к этому типу, просто специфицируйте следующую директиву С# using:

using System.Runtime.Serialization.Formatters.Binary;

Двумя ключевыми методами типа BinaryFormatter, о которых следует знать, являются:

* Serialize () — сохраняет граф объектов в указанный поток в виде последовательности байтов;
* Deserialize() — преобразует сохраненную последовательность байт в граф объектов

Пример сериализации объекта:

BinaryFormatter formatter = new BinaryFormatter();

using (FileStream binaryFileStream = new FileStream(fileName, FileMode.OpenOrCreate))

{

formatter.Serialize(binaryFileStream, hotelSerializable);

}

Пример десериализации объекта:

BinaryFormatter formatter = new BinaryFormatter();

FileStream binaryFileStream = new FileStream(fileName, FileMode.Open);

hotelSerializable = (HotelSerializable)formatter.Deserialize(binaryFileStream);

**Задание на лабораторную работу**

1. Разработать Windows Forms приложение для просмотра, редактирования и сохранения текстового файла. Программа должна автоматически перезагружать содержимое загруженного файла, если оно изменилось другим процессом.
2. Добавьте в предыдущую лабораторную работу возможность загружать и сохранять коллекции в файлы, использую сериализацию и десериализацию в 3-х форматах: XML, JSON и двоичный.

**Контрольные вопросы**

1. Какие классы и как используются для работы с потоками данных?
2. Какие классы и как используются для работы с файловой системой?
3. Какие классы и атрибуты используются для организации бинарной сериализации и десериализации в .NET Framework?
4. Какие классы и атрибуты используются для организации XML сериализации и десериализации в .NET Framework?
5. Какие классы и атрибуты используются для организации JSON сериализации и десериализации в .NET Framework?

# Список рекомендуемой литературы

1. Нейгел, К. C# 5.0 и платформа .NET 4.5 для профессионалов / К.Нейгел, Б.Ивьен, Дж.Глинн, М.Скиннер, К.Уотсон. – М.: Диалектика, 2013. – 1440 с.
2. Неш, Т. C# 2010: ускоренный курс для профессионалов / Т.Неш – М.: Вильямс, 2010. – 592 с.
3. Снелл, М. Microsoft Visual Studio 2008 / М.Снелл, Л.Пауэрс. – СПб: БХВ-Петербург, 2009. – 1200 с.
4. Троелсен, Э. Язык программирования С# 5.0 и платформа .NET 4.5 / Э.Троелсен. – 6-е изд. – М.: Вильямс, 2015. – 1312 с.
5. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж.Рихтер. – СПб.: Питер, 2016. – 896 с.
6. Потоки (Streams) и ввод-вывод (IO) в C#. – Режим доступа: http://blog.rc21net.ru/%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B8-streams-%D0%B8-%D0%B2%D0%B2%D0%BE%D0%B4-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4-io-%D0%B2-c-sharp/

Программирование в среде Microsoft .Net. Организация работы с потоками данных [Текст] + [Электронный ресурс]: Методические указания к выполнению лабораторной работы №8 для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия». – Брянск: БГТУ, 2017. – 38 с.

КОРОСТЕЛЁВ ДМИТРИЙ АЛЕКСАНДРОВИЧ

Научный редактор Д.И. Булатицкий

Компьютерный набор Д.А. Коростелёв

Иллюстрации Д.А. Коростелёв

Подписано в печать 07.07.2017 г. Формат 60х84 1/16 Бумага офсетная. Офсетная печать. Усл.печ.л. 0,63 Уч.-изд.л. 0,63 Тираж 1 экз.

Брянский государственный технический университет

Кафедра «Информатика и программное обеспечение», тел. 56-09-84

241035, Брянск, бульвар 50 лет Октября, 7 БГТУ