Операционные системы. Практика 1

tags: Операционные системыtutorialsMIREAМИРЭАfileФайловая системаJSONXMLсериализацияZIP

@author: Latypova Olga  
reviewed: 2020-02-11  
группы: БББО-09-19

Ссылка на курс: [**https://hackmd.io/@0x41/OS\_Lab\_1**](https://hackmd.io/@0x41/OS_Lab_1)  
В локальной сети РТУ МИРЭА: S:\OS\SDL Практика 1

Задание

Краткое задание

Разработать программу для взаимодействия с файлами в различных форматах с возможностью их модификации, создания или удаления. Разработанный код должен иметь возможность его повторного использования и модификации.

Детальное задание

1. Вывести информацию в консоль о логических дисках, именах, метке тома, размере типе файловой системы.
2. Работа с файлами ( класс File, FileInfo, FileStream и другие)

* Создать файл
* Записать в файл строку
* Прочитать файл в консоль
* Удалить файл

1. Работа с форматом JSON

* Создать файл формате JSON из редактора в
* Создать новый объект. Выполнить сериализацию объекта в формате JSON и записать в файл.
* Прочитать файл в консоль
* Удалить файл

1. Работа с форматом XML

* Создать файл формате XML из редактора
* Записать в файл новые данные из консоли .
* Прочитать файл в консоль.
* Удалить файл.

1. Создание zip архива, добавление туда файла, определение размера архива

* Создать архив в форматер zip
* Добавить файл в архив
* Разархивировать файл и вывести данные о нем
* Удалить файл и архив

Выполнение задания допускается на любом языке программирования.

Литература:

1. [Полное руководство по языку программирования С# 9.0 и платформе .NET 5](https://metanit.com/sharp/tutorial/)

Теоретическая часть.

Работа с потоками и файловой системой

Большинство задач в программировании так или иначе связаны с работой с файлами и каталогами. Нам может потребоваться прочитать текст из файла или наоборот произвести запись, удалить файл или целый каталог, не говоря уже о более комплексных задачах, как например, создание текстового редактора и других подобных задачах.

Фреймворк .NET предоставляет большие возможности по управлению и манипуляции файлами и каталогами, которые по большей части сосредоточены в пространстве имен [System.IO](http://System.IO/). Классы, расположенные в этом пространстве имен (такие как Stream, StreamWriter, FileStream и др.), позволяют управлять файловым вводом-выводом.

Работа с дисками

Работу с файловой системой начнем с самого верхнего уровня - дисков. Для представления диска в пространстве имен [System.IO](http://System.IO/) имеется класс DriveInfo.

Этот класс имеет статический метод GetDrives, который возвращает имена всех логических дисков компьютера. Также он предоставляет ряд полезных свойств:

* AvailableFreeSpace: указывает на объем доступного свободного места на диске в байтах
* DriveFormat: получает имя файловой системы
* DriveType: представляет тип диска
* IsReady: готов ли диск (например, DVD-диск может быть не вставлен в дисковод)
* Name: получает имя диска
* TotalFreeSpace: получает общий объем свободного места на диске в байтах
* TotalSize: общий размер диска в байтах
* VolumeLabel: получает или устанавливает метку тома

Получим имена и свойства всех дисков на компьютере:

using System;

using System.IO;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

DriveInfo[] drives = DriveInfo.GetDrives();

foreach (DriveInfo drive in drives)

{

Console.WriteLine($"Название: {drive.Name}");

Console.WriteLine($"Тип: {drive.DriveType}");

if (drive.IsReady)

{

Console.WriteLine($"Объем диска: {drive.TotalSize}");

Console.WriteLine($"Свободное пространство: {drive.TotalFreeSpace}");

Console.WriteLine($"Метка: {drive.VolumeLabel}");

}

Console.WriteLine();

}

}

}

}

Работа с каталогами

Для работы с каталогами в пространстве имен [System.IO](http://System.IO/) предназначены сразу два класса: Directory и DirectoryInfo.

Класс Directory

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

* CreateDirectory(path): создает каталог по указанному пути path
* Delete(path): удаляет каталог по указанному пути path
* Exists(path): определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false
* GetDirectories(path): получает список каталогов в каталоге path
* GetFiles(path): получает список файлов в каталоге path
* Move(sourceDirName, destDirName): перемещает каталог
* GetParent(path): получение родительского каталога

Класс DirectoryInfo

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на Directory. Некоторые из его свойств и методов:

* Create(): создает каталог
* CreateSubdirectory(path): создает подкаталог по указанному пути path
* Delete(): удаляет каталог
* Свойство Exists: определяет, существует ли каталог
* GetDirectories(): получает список каталогов
* GetFiles(): получает список файлов
* MoveTo(destDirName): перемещает каталог
* Свойство Parent: получение родительского каталога
* Свойство Root: получение корневого каталога

Посмотрим на примерах применение этих классов

Получение списка файлов и подкаталогов

string dirName = "C:\\";

if (Directory.Exists(dirName))

{

Console.WriteLine("Подкаталоги:");

string[] dirs = Directory.GetDirectories(dirName);

foreach (string s in dirs)

{

Console.WriteLine(s);

}

Console.WriteLine();

Console.WriteLine("Файлы:");

string[] files = Directory.GetFiles(dirName);

foreach (string s in files)

{

Console.WriteLine(s);

}

}

Обратите внимание на использование слешей в именах файлов. Либо мы используем двойной слеш: “C:\”, либо одинарный, но тогда перед всем путем ставим знак @: @“C:\Program Files”

Создание каталога

string path = @"C:\SomeDir";

string subpath = @"program\avalon";

DirectoryInfo dirInfo = new DirectoryInfo(path);

if (!dirInfo.Exists)

{

dirInfo.Create();

}

dirInfo.CreateSubdirectory(subpath);

Вначале проверяем, а нету ли такой директории, так как если она существует, то ее создать будет нельзя, и приложение выбросит ошибку. В итоге у нас получится следующий путь: “C:\SomeDir\program\avalon”

Получение информации о каталоге

string dirName = "C:\\Program Files";

DirectoryInfo dirInfo = new DirectoryInfo(dirName);

Console.WriteLine($"Название каталога: {dirInfo.Name}");

Console.WriteLine($"Полное название каталога: {dirInfo.FullName}");

Console.WriteLine($"Время создания каталога: {dirInfo.CreationTime}");

Console.WriteLine($"Корневой каталог: {dirInfo.Root}");

Удаление каталога

Если мы просто применим метод Delete к непустой папке, в которой есть какие-нибудь файлы или подкаталоги, то приложение нам выбросит ошибку. Поэтому нам надо передать в метод Delete дополнительный параметр булевого типа, который укажет, что папку надо удалять со всем содержимым:

string dirName = @"C:\SomeFolder";

try

{

DirectoryInfo dirInfo = new DirectoryInfo(dirName);

dirInfo.Delete(true);

Console.WriteLine("Каталог удален");

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Или так:

string dirName = @"C:\SomeFolder";

Directory.Delete(dirName, true);

Перемещение каталога

string oldPath = @"C:\SomeFolder";

string newPath = @"C:\SomeDir";

DirectoryInfo dirInfo = new DirectoryInfo(oldPath);

if (dirInfo.Exists && Directory.Exists(newPath) == false)

{

dirInfo.MoveTo(newPath);

}

При перемещении надо учитывать, что новый каталог, в который мы хотим перемесить все содержимое старого каталога, не должен существовать.

Работа с файлами. Классы File и FileInfo

Подобно паре Directory/DirectoryInfo для работы с файлами предназначена пара классов File и FileInfo. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства класса FileInfo:

* CopyTo(path): копирует файл в новое место по указанному пути path
* Create(): создает файл
* Delete(): удаляет файл
* MoveTo(destFileName): перемещает файл в новое место
* Свойство Directory: получает родительский каталог в виде объекта DirectoryInfo
* Свойство DirectoryName: получает полный путь к родительскому каталогу
* Свойство Exists: указывает, существует ли файл
* Свойство Length: получает размер файла
* Свойство Extension: получает расширение файла
* Свойство Name: получает имя файла
* Свойство FullName: получает полное имя файла

Класс File реализует похожую функциональность с помощью статических методов:

* Copy(): копирует файл в новое место
* Create(): создает файл
* Delete(): удаляет файл
* Move: перемещает файл в новое место
* Exists(file): определяет, существует ли файл

Получение информации о файле

string path = @"C:\apache\hta.txt";

FileInfo fileInf = new FileInfo(path);

if (fileInf.Exists)

{

Console.WriteLine("Имя файла: {0}", fileInf.Name);

Console.WriteLine("Время создания: {0}", fileInf.CreationTime);

Console.WriteLine("Размер: {0}", fileInf.Length);

}

Удаление файла

string path = @"C:\apache\hta.txt";

FileInfo fileInf = new FileInfo(path);

if (fileInf.Exists)

{

fileInf.Delete();

// альтернатива с помощью класса File

// File.Delete(path);

}

Перемещение файла

string path = @"C:\apache\hta.txt";

string newPath = @"C:\SomeDir\hta.txt";

FileInfo fileInf = new FileInfo(path);

if (fileInf.Exists)

{

fileInf.MoveTo(newPath);

// альтернатива с помощью класса File

// File.Move(path, newPath);

}

Копирование файла

string path = @"C:\apache\hta.txt";

string newPath = @"C:\SomeDir\hta.txt";

FileInfo fileInf = new FileInfo(path);

if (fileInf.Exists)

{

fileInf.CopyTo(newPath, true);

// альтернатива с помощью класса File

// File.Copy(path, newPath, true);

}

Метод CopyTo класса FileInfo принимает два параметра: путь, по которому файл будет копироваться, и булевое значение, которое указывает, надо ли при копировании перезаписывать файл (если true, как в случае выше, файл при копировании перезаписывается). Если же в качестве последнего параметра передать значение false, то если такой файл уже существует, приложение выдаст ошибку.

Метод Copy класса File принимает три параметра: путь к исходному файлу, путь, по которому файл будет копироваться, и булевое значение, указывающее, будет ли файл перезаписываться.

FileStream. Чтение и запись файла

Класс FileStream представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Создание FileStream

Для создания объекта FileStream можно использовать как конструкторы этого класса, так и статические методы класса File. Конструктор FileStream имеет множество перегруженных версий, из которых отмечу лишь одну, самую простую и используемую:

FileStream(string filename, FileMode mode)

Здесь в конструктор передается два параметра: путь к файлу и перечисление FileMode. Данное перечисление указывает на режим доступа к файлу и может принимать следующие значения:

* Append: если файл существует, то текст добавляется в конец файл. Если файла нет, то он создается. Файл открывается только для записи.
* Create: создается новый файл. Если такой файл уже существует, то он перезаписывается
* CreateNew: создается новый файл. Если такой файл уже существует, то он приложение выбрасывает ошибку
* Open: открывает файл. Если файл не существует, выбрасывается исключение
* OpenOrCreate: если файл существует, он открывается, если нет - создается новый
* Truncate: если файл существует, то он перезаписывается. Файл открывается только для записи.  
  Другой способ создания объекта FileStream представляют статические методы класса File:

FileStream File.Open(string file, FileMode mode);

FileStream File.OpenRead(string file);

FileStream File.OpenWrite(string file);

Первый метод открывает файл с учетом объекта FileMode и возвращает файловой поток FileStream. У этого метода также есть несколько перегруженных версий. Второй метод открывает поток для чтения, а третий открывает поток для записи.

Свойства и методы FileStream

Рассмотрим наиболее важные его свойства и методы класса FileStream:

* Свойство Length: возвращает длину потока в байтах
* Свойство Position: возвращает текущую позицию в потоке
* void CopyTo(Stream destination): копирует данные из текущего потока в поток destination
* Task CopyToAsync(Stream destination): асинхронная версия метода CopyToAsync
* int Read(byte[] array, int offset, int count): считывает данные из файла в массив байтов и возвращает количество успешно считанных байтов. Принимает три параметра:
  + array - массив байтов, куда будут помещены считываемые из файла данные
  + offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
  + count - максимальное число байтов, предназначеных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
  + Task<int> ReadAsync(byte[] array, int offset, int count): асинхронная версия метода Read
* long Seek(long offset, SeekOrigin origin): устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
* void Write(byte[] array, int offset, int count): записывает в файл данные из массива байтов. Принимает три параметра:
  + array - массив байтов, откуда данные будут записываться в файл
  + offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток
  + count - максимальное число байтов, предназначенных для записи
* ValueTask WriteAsync(byte[] array, int offset, int count): асинхронная версия метода Write

Чтение и запись файлов

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру, FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.  
Посмотрим на примере считывания-записи в текстовый файл:

using System;

using System.IO;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

// создаем каталог для файла

string path = @"C:\SomeDir2";

DirectoryInfo dirInfo = new DirectoryInfo(path);

if (!dirInfo.Exists)

{

dirInfo.Create();

}

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

string text = Console.ReadLine();

// запись в файл

using (FileStream fstream = new FileStream($"{path}\note.txt", FileMode.OpenOrCreate))

{

// преобразуем строку в байты

byte[] array = System.Text.Encoding.Default.GetBytes(text);

// запись массива байтов в файл

fstream.Write(array, 0, array.Length);

Console.WriteLine("Текст записан в файл");

}

// чтение из файла

using (FileStream fstream = File.OpenRead($"{path}\note.txt"))

{

// преобразуем строку в байты

byte[] array = new byte[fstream.Length];

// считываем данные

fstream.Read(array, 0, array.Length);

// декодируем байты в строку

string textFromFile = System.Text.Encoding.Default.GetString(array);

Console.WriteLine($"Текст из файла: {textFromFile}");

}

Console.ReadLine();

}

}

}

Разберем этот пример. Вначале создается папка для файла. Кроме того, на уровне операционной системы могут быть установлены ограничения на запись в опрееделенных каталогах, и при попытке создания и записи файла в подобных каталогах мы получим ошибку.

И при чтении, и при записи используется оператор using. Не надо путать данный оператор с директивой using, которая подключает пространства имен в начале файла кода. Оператор using позволяет создавать объект в блоке кода, по завершению которого вызывается метод Dispose у этого объекта, и, таким образом, объект уничтожается. В данном случае в качестве такого объекта служит переменная fstream.

И при записи, и при чтении применяется объект кодировки Encoding.Default из пространства имен System.Text. В данном случае мы используем два его метода: GetBytes для получения массива байтов из строки и GetString для получения строки из массива байтов.

В итоге введенная нами строка записывается в файл note.txt. По сути это бинарный файл (не текстовый), хотя если мы в него запишем только строку, то сможем посмотреть в удобочитаемом виде этот файл, открыв его в текстовом редакторе. Однако если мы в него запишем случайные байты, например:

fstream.WriteByte(13);

fstream.WriteByte(103);

То у нас могут возникнуть проблемы с его пониманием. Поэтому для работы непосредственно с текстовыми файлами предназначены отдельные классы - StreamReader и StreamWriter.

Хотя в данном простеньком консольном приложении, но в реальных приложениях рекомендуется использовать асинхронные версии методов FileStream, поскольку операции с файлами могут занимать продолжительное время и являются узким местом в работе программы. Например, изменим выше приведенную программу, применив асинхронные методы:

using System;

using System.IO;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static async Task Main(string[] args)

{

// создаем каталог для файла

string path = @"C:\SomeDir3";

DirectoryInfo dirInfo = new DirectoryInfo(path);

if (!dirInfo.Exists)

{

dirInfo.Create();

}

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

string text = Console.ReadLine();

// запись в файл

using (FileStream fstream = new FileStream($"{path}\note.txt", FileMode.OpenOrCreate))

{

byte[] array = System.Text.Encoding.Default.GetBytes(text);

// асинхронная запись массива байтов в файл

await fstream.WriteAsync(array, 0, array.Length);

Console.WriteLine("Текст записан в файл");

}

// чтение из файла

using (FileStream fstream = File.OpenRead($"{path}\note.txt"))

{

byte[] array = new byte[fstream.Length];

// асинхронное чтение файла

await fstream.ReadAsync(array, 0, array.Length);

string textFromFile = System.Text.Encoding.Default.GetString(array);

Console.WriteLine($"Текст из файла: {textFromFile}");

}

Console.ReadLine();

}

}

}

Произвольный доступ к файлам

Нередко бинарные файлы представляют определенную структуру. И, зная эту структуру, мы можем взять из файла нужную порцию информации или наоброт записать в определенном месте файла определенный набор байтов. Например, в wav-файлах непосредственно звуковые данные начинаются с 44 байта, а до 44 байта идут различные метаданные - количество каналов аудио, частота дискретизации и т.д.

С помощью метода Seek() мы можем управлять положением курсора потока, начиная с которого производится считывание или запись в файл. Этот метод принимает два параметра: offset (смещение) и позиция в файле. Позиция в файле описывается тремя значениями:

* SeekOrigin.Begin: начало файла
* SeekOrigin.End: конец файла
* SeekOrigin.Current: текущая позиция в файле  
  Курсор потока, с которого начинается чтение или запись, смещается вперед на значение offset относительно позиции, указанной в качестве второго параметра. Смещение может быть отрицательным, тогда курсор сдвигается назад, если положительное - то вперед.

Рассмотрим на примере:

using System.IO;

using System.Text;

class Program

{

static void Main(string[] args)

{

string text = "hello world";

// запись в файл

using (FileStream fstream = new FileStream(@"D:\note.dat", FileMode.OpenOrCreate))

{

// преобразуем строку в байты

byte[] input = Encoding.Default.GetBytes(text);

// запись массива байтов в файл

fstream.Write(input, 0, input.Length);

Console.WriteLine("Текст записан в файл");

// перемещаем указатель в конец файла, до конца файла- пять байт

fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока

// считываем четыре символов с текущей позиции

byte[] output = new byte[4];

fstream.Read(output, 0, output.Length);

// декодируем байты в строку

string textFromFile = Encoding.Default.GetString(output);

Console.WriteLine($"Текст из файла: {textFromFile}"); // worl

// заменим в файле слово world на слово house

string replaceText = "house";

fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока

input = Encoding.Default.GetBytes(replaceText);

fstream.Write(input, 0, input.Length);

// считываем весь файл

// возвращаем указатель в начало файла

fstream.Seek(0, SeekOrigin.Begin);

output = new byte[fstream.Length];

fstream.Read(output, 0, output.Length);

// декодируем байты в строку

textFromFile = Encoding.Default.GetString(output);

Console.WriteLine($"Текст из файла: {textFromFile}"); // hello house

}

Console.Read();

}

}

Консольный вывод:

Текст записан в файл

Текст из файла: worl

Текст из файла: hello house

Вызов fstream.Seek(-5, SeekOrigin.End) перемещает курсор потока в конец файлов назад на пять символов:

То есть после записи в новый файл строки “hello world” курсор будет стоять на позиции символа “w”.

После этого считываем четыре байта начиная с символа “w”. В данной кодировке 1 символ будет представлять 1 байт. Поэтому чтение 4 байтов будет эквивалентно чтению четырех сиволов: “worl”.

Затем опять же перемещаемся в конец файла, не доходя до конца пять символов (то есть опять же с позиции символа “w”), и осуществляем запись строки “house”. Таким образом, строка “house” заменяет строку “world”.

Закрытие потока

В примерах выше для закрытия потока применяется конструкция using. После того как все операторы и выражения в блоке using отработают, объект FileStream уничтожается. Однако мы можем выбрать и другой способ:

FileStream fstream = null;

try

{

fstream = new FileStream(@"D:\note3.dat", FileMode.OpenOrCreate);

// операции с потоком

}

catch(Exception ex)

{

}

finally

{

if (fstream != null)

fstream.Close();

}

Если мы не используем конструкцию using, то нам надо явным образом вызвать метод Close(): fstream.Close()

Чтение и запись текстовых файлов. StreamReader и StreamWriter

Класс FileStream не очень удобно применять для работы с текстовыми файлами. К тому же для этого в пространстве [System.IO](http://System.IO/) определены специальные классы: StreamReader и StreamWriter.

Запись в файл и StreamWriter

Для записи в текстовый файл используется класс StreamWriter. Некоторые из его конструкторов, которые могут применяться для создания объекта StreamWriter:

* StreamWriter(string path): через параметр path передается путь к файлу, который будет связан с потоком
* StreamWriter(string path, bool append): параметр append указывает, надо ли добавлять в конец файла данные или же перезаписывать файл. Если равно true, то новые данные добавляются в конец файла. Если равно false, то файл перезаписываетсяя заново
* StreamWriter(string path, bool append, System.Text.Encoding encoding): параметр encoding указывает на кодировку, которая будет применяться при записи

Свою функциональность StreamWriter реализует через следующие методы:

* int Close(): закрывает записываемый файл и освобождает все ресурсы
* void Flush(): записывает в файл оставшиеся в буфере данные и очищает буфер.
* Task FlushAsync(): асинхронная версия метода Flush
* void Write(string value): записывает в файл данные простейших типов, как int, double, char, string и т.д. Соответственно имеет ряд перегруженных версий для записи данных элементарных типов, например, Write(char value), Write(int value), Write(double value) и т.д.
* Task WriteAsync(string value): асинхронная версия метода Write
* void WriteLine(string value): также записывает данные, только после записи добавляет в файл символ окончания строки
* Task WriteLineAsync(string value): асинхронная версия метода WriteLine

Рассмотрим запись в файл на примере:

using System;

using System.IO;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

string writePath = @"C:\SomeDir\hta.txt";

string text = "Привет мир!\nПока мир...";

try

{

using (StreamWriter sw = new StreamWriter(writePath, false, System.Text.Encoding.Default))

{

sw.WriteLine(text);

}

using (StreamWriter sw = new StreamWriter(writePath, true, System.Text.Encoding.Default))

{

sw.WriteLine("Дозапись");

sw.Write(4.5);

}

Console.WriteLine("Запись выполнена");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

}

В данном случае два раза создаем объект StreamWriter. В первом случае если файл существует, то он будет перезаписан. Если не существует, он будет создан. И в нее будет записан текст из переменной text. Во втором случае файл открывается для дозаписи, и будут записаны атомарные данные - строка и число. В обоих случаях будет использоваться кодировка по умолчанию.

По завершении программы в папке C://SomeDir мы сможем найти файл hta.txt, который будет иметь следующие строки:

Привет мир!

Пока мир...

Дозапись

4,5

Поскольку операции с файлами могут занимать продолжительное время, то в общем случае рекомендуется использовать асинхронную запись. Используем асинхронные версии методов:

using System;

using System.IO;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static async Task Main(string[] args)

{

string writePath = @"C:\SomeDir\hta2.txt";

string text = "Привет мир!\nПока мир...";

try

{

using (StreamWriter sw = new StreamWriter(writePath, false, System.Text.Encoding.Default))

{

await sw.WriteLineAsync(text);

}

using (StreamWriter sw = new StreamWriter(writePath, true, System.Text.Encoding.Default))

{

await sw.WriteLineAsync("Дозапись");

await sw.WriteAsync("4,5");

}

Console.WriteLine("Запись выполнена");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

}

Обратите внимание, что асинхронные версии есть не для всех перегрузок метода Write.

Чтение из файла и StreamReader

Класс StreamReader позволяет нам легко считывать весь текст или отдельные строки из текстового файла.

Некоторые из конструкторов класса StreamReader:

* StreamReader(string path): через параметр path передается путь к считываемому файлу
* StreamReader(string path, System.Text.Encoding encoding): параметр encoding задает кодировку для чтения файла

Среди методов StreamReader можно выделить следующие:

* void Close(): закрывает считываемый файл и освобождает все ресурсы
* int Peek(): возвращает следующий доступный символ, если символов больше нет, то возвращает -1
* int Read(): считывает и возвращает следующий символ в численном представлении. Имеет перегруженную версию: Read(char[] array, int index, int count), где array - массив, куда считываются символы, index - индекс в массиве array, начиная с которого записываются считываемые символы, и count - максимальное количество считываемых символов
* Task<int> ReadAsync(): асинхронная версия метода Read
* string ReadLine(): считывает одну строку в файле
* string ReadLineAsync(): асинхронная версия метода ReadLine
* string ReadToEnd(): считывает весь текст из файла
* string ReadToEndAsync(): асинхронная версия метода ReadToEnd

Сначала считаем текст полностью из ранее записанного файла:

using System;

using System.IO;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static async Task Main(string[] args)

{

string path = @"C:\SomeDir\hta.txt";

try

{

using (StreamReader sr = new StreamReader(path))

{

Console.WriteLine(sr.ReadToEnd());

}

// асинхронное чтение

using (StreamReader sr = new StreamReader(path))

{

Console.WriteLine(await sr.ReadToEndAsync());

}

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

}

Считаем текст из файла построчно:

string path= @"C:\SomeDir\hta.txt";

using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))

{

string line;

while ((line = sr.ReadLine()) != null)

{

Console.WriteLine(line);

}

}

// асинхронное чтение

using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))

{

string line;

while ((line = await sr.ReadLineAsync()) != null)

{

Console.WriteLine(line);

}

}

В данном случае считываем построчно через цикл while: while ((line = sr.ReadLine()) != null) - сначала присваиваем переменной line результат функции sr.ReadLine(), а затем проверяем, не равна ли она null. Когда объект sr дойдет до конца файла и больше строк не останется, то метод sr.ReadLine() будет возвращать null.

Бинарные файлы. BinaryWriter и BinaryReader

ля работы с бинарными файлами предназначена пара классов BinaryWriter и BinaryReader. Эти классы позволяют читать и записывать данные в двоичном формате.

Основные метода класса BinaryWriter

* Close(): закрывает поток и освобождает ресурсы
* Flush(): очищает буфер, дописывая из него оставшиеся данные в файл
* Seek(): устанавливает позицию в потоке
* Write(): записывает данные в поток

Основные метода класса BinaryReader

* Close(): закрывает поток и освобождает ресурсы
* ReadBoolean(): считывает значение bool и перемещает указатель на один байт
* ReadByte(): считывает один байт и перемещает указатель на один байт
* ReadChar(): считывает значение char, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке
* ReadDecimal(): считывает значение decimal и перемещает указатель на 16 байт
* ReadDouble(): считывает значение double и перемещает указатель на 8 байт
* ReadInt16(): считывает значение short и перемещает указатель на 2 байта
* ReadInt32(): считывает значение int и перемещает указатель на 4 байта
* ReadInt64(): считывает значение long и перемещает указатель на 8 байт
* ReadSingle(): считывает значение float и перемещает указатель на 4 байта
* ReadString(): считывает значение string. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

С чтением бинарных данных все просто: соответствующий метод считывает данные определенного типа и перемещает указатель на размер этого типа в байтах, например, значение типа int занимает 4 байта, поэтому BinaryReader считает 4 байта и переместит указать на эти 4 байта.

Посмотрим на реальной задаче применение этих классов. Попробуем с их помощью записывать и считывать из файла массив структур:

struct State

{

public string name;

public string capital;

public int area;

public double people;

public State(string n, string c, int a, double p)

{

name = n;

capital = c;

people = p;

area = a;

}

}

class Program

{

static void Main(string[] args)

{

State[] states = new State[2];

states[0] = new State("Германия", "Берлин", 357168, 80.8);

states[1] = new State("Франция", "Париж", 640679, 64.7);

string path= @"C:\SomeDir\states.dat";

try

{

// создаем объект BinaryWriter

using (BinaryWriter writer = new BinaryWriter(File.Open(path, FileMode.OpenOrCreate)))

{

// записываем в файл значение каждого поля структуры

foreach (State s in states)

{

writer.Write(s.name);

writer.Write(s.capital);

writer.Write(s.area);

writer.Write(s.people);

}

}

// создаем объект BinaryReader

using (BinaryReader reader = new BinaryReader(File.Open(path, FileMode.Open)))

{

// пока не достигнут конец файла

// считываем каждое значение из файла

while (reader.PeekChar() > -1)

{

string name = reader.ReadString();

string capital = reader.ReadString();

int area = reader.ReadInt32();

double population = reader.ReadDouble();

Console.WriteLine("Страна: {0} столица: {1} площадь {2} кв. км численность населения: {3} млн. чел.",

name, capital, area, population);

}

}

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

Console.ReadLine();

}

}

Итак, у нас есть структура State с некоторым набором полей. В основной программе создаем массив структур и записываем с помощью BinaryWriter. Этот класс в качестве параметра в конструкторе принимает объект Stream, который создается вызовом File.Open(path, FileMode.OpenOrCreate).

Затем в цикле пробегаемся по массиву структур и записываем каждое поле структуры в поток. В том порядке, в каком эти значения полей записываются, в том порядке они и будут размещаться в файле.

Затем считываем из записанного файла. Конструктор класса BinaryReader также в качестве параметра принимает объект потока, только в данном случае устанавливаем в качестве режима FileMode.Open: new BinaryReader(File.Open(path, FileMode.Open))

В цикле while считываем данные. Чтобы узнать окончание потока, вызываем метод PeekChar(). Этот метод считывает следующий символ и возвращает его числовое представление. Если символ отсутствует, то метод возвращает -1, что будет означать, что мы достигли конца файла.

В цикле последовательно считываем значения поле структур в том же порядке, в каком они записывались.

Таким образом, классы BinaryWriter и BinaryReader очень удобны для работы с бинарными файлами, особенно когда нам известна структура этих файлов. В то же время для хранения и считывания более комплексных объектов, например, объектов классов, лучше подходит другое решение - сериализация.

Бинарная сериализация. BinaryFormatter

В прошлых темах было рассмотрено как сохранять и считывать информацию с текстовых и бинарных файлов с помощью классов из пространства [System.IO](http://System.IO/). Но .NET также предоставляет еще один механизм для удобной работы с бинарными файлами и их данными - бинарную сериализацию. Сериализация представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - десериализацию, то есть получить из потока байтов ранее сохраненный объект.  
Атрибут Serializable

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом Serializable:

[Serializable]

class Person

{

public string Name { get; set; }

public int Year { get; set; }

public Person(string name, int year)

{

Name = name;

Year = year;

}

}

При отстутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение SerializationException.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом NonSerialized:

[Serializable]

class Person

{

public string Name { get; set; }

public int Year { get; set; }

[NonSerialized]

public string accNumber;

public Person(string name, int year, string acc)

{

Name = name;

Year = year;

accNumber = acc;

}

}

При наследовании подобного класса, следует учитывать, что атрибут Serializable автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут:

[Serializable]

class Worker : Person

Для бинарной сериализации применяется класс BinaryFormatter:

using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

namespace Serialization

{

[Serializable]

class Person

{

public string Name { get; set; }

public int Age { get; set; }

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");

// создаем объект BinaryFormatter

BinaryFormatter formatter = new BinaryFormatter();

// получаем поток, куда будем записывать сериализованный объект

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

formatter.Serialize(fs, person);

Console.WriteLine("Объект сериализован");

}

// десериализация из файла people.dat

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

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

Console.WriteLine("Объект десериализован");

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

}

Console.ReadLine();

}

}

}

Так как класс BinaryFormatter определен в пространстве имен System.Runtime.Serialization.Formatters.Binary, то в самом начале подключаем его.

У нас есть простенький класс Person, который объявлен с атрибутом Serializable. Благодаря этому его объекты будут доступны для сериализации.

Далее создаем объект BinaryFormatter: BinaryFormatter formatter = new BinaryFormatter();

Затем последовательно выполняем сериализацию и десериализацию. Для обоих операций нам нужен поток, в который либо сохранять, либо из которого считывать данные. Данный поток представляет объект FileStream, который записывает нужный нам объект Person в файл people.dat.

Сериализация одним методом formatter.Serialize(fs, person) добавляет все данные об объекте Person в файл people.dat.

При десериализации нам нужно еще преобразовать объект, возвращаемый функцией Deserialize, к типу Person: (Person)formatter.Deserialize(fs).

Как вы видите, сериализация значительно упрощает процесс сохранения объектов в бинарную форму по сравнению, например, с использованием связки классов BinaryWriter/BinaryReader.

Хотя мы взяли лишь один объект Person, но равным образом мы можем использовать и массив подобных объектов, список или иную коллекцию, к которой применяется атрибут Serializable. Посмотрим на примере массива:

Person person1 = new Person("Tom", 29);

Person person2 = new Person("Bill", 25);

// массив для сериализации

Person[] people = new Person[] { person1, person2 };

BinaryFormatter formatter = new BinaryFormatter();

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

// сериализуем весь массив people

formatter.Serialize(fs, people);

Console.WriteLine("Объект сериализован");

}

// десериализация

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

Person[] deserilizePeople = (Person[])formatter.Deserialize(fs);

foreach (Person p in deserilizePeople)

{

Console.WriteLine($"Имя: {p.Name} --- Возраст: {p.Age}");

}

}

Архивация и сжатие файлов

Кроме классов чтения-записи .NET предоставляет классы, которые позволяют сжимать файлы, а также затем восстанавливать их в исходное состояние.

Это классы ZipFile, DeflateStream и GZipStream, которые находятся в пространстве имен System.IO.Compression и представляют реализацию одного из алгоритмов сжатия Deflate или GZip.

GZipStream и DeflateStream

Для создания объекта GZipStream можно использовать один из его конструкторов:

* GZipStream(Stream stream, CompressionLevel level): stream представляетданные, а level задаетуровеньсжатия
* GZipStream(Stream stream, CompressionMode mode): mode указывает, будут ли данные сжиматься или, наоборот, восстанавливаться и может принимать два значения:
  + CompressionMode.Compress: данные сжимаются
  + CompressionMode.Decompress: данные восстанавливатьсяся

Если данные сжимаются, то stream указывает на поток архивируемых данных. Если данные восстанавливаются, то stream указывает на поток, куда будут передаваться восстановленные данные.

* GZipStream(Stream stream, CompressionLevel level, bool leaveMode): параметр leaveMode указывает, надо ли оставить открытым поток stream после удаления объекта GZipStream. Если значение true, то поток остается открытым
* GZipStream(Stream stream, CompressionMode mode, bool leaveMode)

Для управления сжатием/восстанавлением данных GZipStream предоставляет ряд методов. Основые из них:

* void CopyTo(Stream destination): копируетвседанныевпоток destination
* Task CopyToAsync(Stream destination): асинхроннаяверсияметода CopyTo
* void Flush(): очищает буфер, записывая все его данные в файл
* Task FlushAsync(): асинхронная версия метода Flush
* int Read(byte[] array, int offset, int count): считывает данные из файла в массив байтов и возвращает количество успешно считанных байтов. Принимает три параметра:
  + array - массив байтов, куда будут помещены считываемые из файла данные
  + offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
  + count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
  + int Read(byte[] array, int offset, int count): считывает данные из файла в массив байтов и возвращает количество успешно считанных байтов. Принимает три параметра:
    - array - массив байтов, куда будут помещены считываемые из файла данные
    - offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
    - count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
    - Task<int> ReadAsync(byte[] array, int offset, int count): асинхроннаяверсияметода Read
  + long Seek(long offset, SeekOrigin origin): устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
  + void Write(byte[] array, int offset, int count): записывает в файл данные из массива байтов. Принимает три параметра:
    - array - массив байтов, откуда данные будут записываться в файл
    - offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток
    - count - максимальное число байтов, предназначенных для записи
  + Task WriteAsync(byte[] array, int offset, int count): асинхроннаяверсияметода Write

Рассмотрим применение класса GZipStream на примере:

using System.IO;

using System.IO.Compression;

class Program

{

static void Main(string[] args)

{

string sourceFile = "D://test/book.pdf"; // исходный файл

string compressedFile = "D://test/book.gz"; // сжатый файл

string targetFile = "D://test/book\_new.pdf"; // восстановленный файл

// создание сжатого файла

Compress(sourceFile, compressedFile);

// чтение из сжатого файла

Decompress(compressedFile, targetFile);

Console.ReadLine();

}

public static void Compress(string sourceFile, string compressedFile)

{

// поток для чтения исходного файла

using (FileStream sourceStream = new FileStream(sourceFile, FileMode.OpenOrCreate))

{

// поток для записи сжатого файла

using (FileStream targetStream = File.Create(compressedFile))

{

// поток архивации

using (GZipStream compressionStream = new GZipStream(targetStream, CompressionMode.Compress))

{

sourceStream.CopyTo(compressionStream); // копируем байты из одного потока в другой

Console.WriteLine("Сжатие файла {0} завершено. Исходный размер: {1} сжатый размер: {2}.",

sourceFile, sourceStream.Length.ToString(), targetStream.Length.ToString());

}

}

}

}

public static void Decompress(string compressedFile, string targetFile)

{

// поток для чтения из сжатого файла

using (FileStream sourceStream = new FileStream(compressedFile, FileMode.OpenOrCreate))

{

// поток для записи восстановленного файла

using (FileStream targetStream = File.Create(targetFile))

{

// поток разархивации

using (GZipStream decompressionStream = new GZipStream(sourceStream, CompressionMode.Decompress))

{

decompressionStream.CopyTo(targetStream);

Console.WriteLine("Восстановлен файл: {0}", targetFile);

}

}

}

}

}

Метод Compress получает название исходного файла, который надо архивировать, и название будущего сжатого файла.

Сначала создается поток для чтения из исходного файла - FileStream sourceStream. Затем создается поток для записи в сжатый файл - FileStream targetStream. Поток архивации GZipStream compressionStream инициализируется потоком targetStream и с помощью метода CopyTo() получает данные от потока sourceStream.

Метод Decompress производит обратную операцию по восстановлению сжатого файла в исходное состояние. Он принимает в качестве параметров пути к сжатому файлу и будущему восстановленному файлу.

Здесь в начале создается поток для чтения из сжатого файла FileStream sourceStream, затем поток для записи в восстанавливаемый файл FileStream targetStream. В конце создается поток GZipStream decompressionStream, который с помощью метода CopyTo() копирует восстановленные данные в поток targetStream.

Чтобы указать потоку GZipStream, для чего именно он предназначен - сжатия или восстановления - ему в конструктор передается параметр CompressionMode, принимающий два значения: Compress и Decompress.

Если бы захотели бы использовать другой класс сжатия - DeflateStream, то мы могли бы просто заменить в коде упоминания GZipStream на DeflateStream, без изменения остального кода. Их использование идентично.  
В то же время при использовании этих классов есть некоторые ограничения, в частности, мы можем сжимать только один файл. Для архивации группы файлы лучше выбрать другие инструменты, например, ZipFile.

ZipFile

Статический класс ZipFile из простанства имен System.IO.Compression предоставляет дополнительные возможности для создания архивов. Он позволяет создавать архив из каталогов. Его основные методы:

* void CreateFromDirectory(string sourceDirectoryName, string destinationFileName): архивируетпапкупопути sourceDirectoryName вфайлсназванием destinationFileName
* void CreateFromDirectory(string sourceFileName, string destinationDirectoryName): извлекаетвсефайлыиз zip-файла sourceFileName вкаталог destinationDirectoryName

Оба метода имеют ряд дополнительных перегруженных версий. Рассмотрим их применение.

using System;

using System.IO.Compression;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

string sourceFolder = "D://test/"; // исходная папка

string zipFile = "D://test.zip"; // сжатый файл

string targetFolder = "D://newtest"; // папка, куда распаковывается файл

ZipFile.CreateFromDirectory(sourceFolder, zipFile);

Console.WriteLine($"Папка {sourceFolder} архивирована в файл {zipFile}");

ZipFile.ExtractToDirectory(zipFile, targetFolder);

Console.WriteLine($"Файл {zipFile} распакован в папку {targetFolder}");

Console.ReadLine();

}

}

}

В данном случае папка “D://test/” методом ZipFile.CreateFromDirectory архивируется в файл test.zip. Затем метод ZipFile.ExtractToDirectory() распаковывает данный файл в папку “D://newtest” (если такой папки нет, она создается).

Работа с JSON

Сериализация в JSON. JsonSerializer

Основная функциональность по работе с JSON сосредоточена в пространстве имен System.Text.Json.

Ключевым типом является класс JsonSerializer, который и позволяет сериализовать объект в json и, наоборот, десериализовать код json в объект C#.

Для сохранения объекта в json в классе JsonSerializer определен статический метод Serialize(), который имеет ряд перегруженных версий. Некоторые из них:

* string Serialize(Object obj, Type type, JsonSerializerOptions options): сериализует объект obj типа type и возвращает код json в виде строки. Последний необязательный параметр options позволяет задать дополнительные опции сериализации
* string Serialize<T>(T obj, JsonSerializerOptions options): типизированная версия сериализует объект obj типа T и возвращает код json в виде строки.
* Task SerializeAsync(Object obj, Type type, JsonSerializerOptions options): сериализует объект obj типа type и возвращает код json в виде строки. Последний необязательный параметр options позволяет задать дополнительные опции сериализации
* Task SerializeAsync<T>(T obj, JsonSerializerOptions options): типизированная версия сериализует объект obj типа T и возвращает код json в виде строки.
* object Deserialize(string json, Type type, JsonSerializerOptions options): десериализует строку json в объект типа type и возвращает десериализованный объект. Последний необязательный параметр options позволяет задать дополнительные опции десериализации
* T Deserialize<T>(string json, JsonSerializerOptions options): десериализует строку json в объект типа T и возвращает его.
* ValueTask<object> DeserializeAsync(Stream utf8Json, Type type, JsonSerializerOptions options, CancellationToken token): десериализует текст UTF-8, который представляет объект JSON, в объект типа type. Последние два параметра необязательны: options позволяет задать дополнительные опции десериализации, а token устанавливает CancellationToken для отмены задачи. Возвращается десериализованный объект, обернутый в ValueTask
* ValueTask<T> DeserializeAsync<T>(Stream utf8Json, JsonSerializerOptions options, CancellationToken token): десериализует текст UTF-8, который представляет объект JSON, в объект типа T. Возвращается десериализованный объект, обернутый в ValueTask

Рассмотрим применение класса на простом примере. Сериализуем и десериализуем простейший объект:

using System;

using System.Text.Json;

namespace HelloApp

{

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

class Program

{

static void Main(string[] args)

{

Person tom = new Person { Name = "Tom", Age = 35 };

string json = JsonSerializer.Serialize<Person>(tom);

Console.WriteLine(json);

Person restoredPerson = JsonSerializer.Deserialize<Person>(json);

Console.WriteLine(restoredPerson.Name);

}

}

}

Здесь вначале сериализуем с помощью метода JsonSerializer.Serialize() объект типа Person в стоку с кодом json. Затем обратно получаем из этой строки объект Person посредством метода JsonSerializer.Deserialize().

Консольный вывод:

{"Name":"Tom","Age": 35}

Tom

Хотя в примере выше сериализовался/десериализовался объект класса, но подобным способом мы также можем сериализовать/десериализовать структуры.

Некоторые замечания по сериализации/десериализации

Объект, который подвергается десериализации, должен иметь конструктор без параметров. Например, в примере выше этот конструктор по умолчанию. Но можно также явным образом определить подобный конструктор в классе.

Сериализации подлежат только публичные свойства объекта (с модификатором public).

Запись и чтение файла json

Поскольку методы SerializeAsyc/DeserializeAsync могут принимать поток типа Stream, то соответственно мы можем использовать файловый поток для сохранения и последующего извлечения данных:

using System;

using System.IO;

using System.Text.Json;

using System.Threading.Tasks;

namespace HelloApp

{

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

class Program

{

static async Task Main(string[] args)

{

// сохранение данных

using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))

{

Person tom = new Person() { Name = "Tom", Age = 35 };

await JsonSerializer.SerializeAsync<Person>(fs, tom);

Console.WriteLine("Data has been saved to file");

}

// чтение данных

using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))

{

Person restoredPerson = await JsonSerializer.DeserializeAsync<Person>(fs);

Console.WriteLine($"Name: {restoredPerson.Name} Age: {restoredPerson.Age}");

}

}

}

}

В данном случае вначале данные сохраняются в файл user.json и затем считываются из него.

Настройка сериализации с помощью JsonSerializerOptions

По умолчанию JsonSerializer сериализует объекты в минимифицированный код. С помощью дополнительного параметра типа JsonSerializerOptions можно настроить механизм сериализации/десериализации, используя свойства JsonSerializerOptions. Некоторые из его свойств:

* AllowTrailingCommas: устанавливает, надо ли добавлять после последнего элемента в json запятую. Если равно true, запятая добавляется
* IgnoreNullValues: устанавливает, будут ли сериализоваться/десериализоваться в json объекты и их свойства со значением null
* IgnoreReadOnlyProperties: аналогично устанавливает, будут ли сериализоваться свойства, предназначенные только для чтения
* WriteIndented: устанавливает, будут ли добавляться в json пробелы (условно говоря, для красоты). Если равно true устанавливаются дополнительные пробелы

Применение:

var options = new JsonSerializerOptions

{

WriteIndented = true

};

Person tom = new Person { Name = "Tom", Age = 35 };

string json = JsonSerializer.Serialize<Person>(tom, options);

Console.WriteLine(json);

Person restoredPerson = JsonSerializer.Deserialize<Person>(json);

Console.WriteLine(restoredPerson.Name);

Консольный вывод:

{

"Name": "Tom",

"Age": 35

}

Tom

Настройка сериализации с помощью атрибутов

По умолчанию сериализации подлежат все публичные свойства. Кроме того, в выходном объекте json все названия свойств соответствуют названиям свойств объекта C#. Однако с помощью атрибутов JsonIgnore и JsonPropertyName.

Атрибут JsonIgnore позволяет исключить из сериализации определенное свойство. А JsonPropertyName позволяет замещать оригинальное название свойства. Пример использования:

using System;

using System.Text.Json;

using System.Text.Json.Serialization;

namespace HelloApp

{

class Person

{

[JsonPropertyName("firstname")]

public string Name { get; set; }

[JsonIgnore]

public int Age { get; set; }

}

class Program

{

static void Main(string[] args)

{

Person tom = new Person() { Name = "Tom", Age = 35 };

string json = JsonSerializer.Serialize<Person>(tom);

Console.WriteLine(json);

Person restoredPerson = JsonSerializer.Deserialize<Person>(json);

Console.WriteLine($"Name: {restoredPerson.Name} Age: {restoredPerson.Age}");

}

}

}

В данном случае свойство Age будет игнорироваться, а для свойства Name будет использоваться псевдоним “firstname”. Консольный вывод:

{"firstname":"Tom"}

Name: Tom Age: 0

Обратите внимание, что, поскольку свойство Age не было сериализовано, то при десериализации для него используется значение по умолчанию.

Работа с XML в C#

XML-документы

На сегодняшний день XML является одним из распространенных стандартов документов, который позволяет в удобной форме сохранять сложные по структуре данные. Поэтому разработчики платформы .NET включили в фреймворк широкие возможности для работы с XML.

Прежде чем перейти непосредственно к работе с XML-файлами, сначала рассмотрим, что представляет собой xml-документ и как он может хранить объекты, используемые в программе на c#.

Например, у нас есть следующий класс:

class User

{

public string Name { get; set; }

public int Age { get; set; }

public string Company { get; set; }

}

В программе на C# мы можем создать список объектов класса User:

User user1 = new User { Name = "Bill Gates", Age = 48, Company = "Microsoft" };

User user2 = new User { Name = "Larry Page", Age = 42, Company = "Google" };

List<User> users = new List<User> { user1, user2 };

Чтобы сохранить список в формате xml мы могли бы использовать следующий xml-файл:

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

<users>

<username="Bill Gates">

<company>Microsoft</company>

<age>48</age>

</user>

<username="Larry Page">

<company>Google</company>

<age>48</age>

</user>

</users>

XML-документ объявляет строка <?xml version="1.0" encoding="utf-8" ?>. Она задает версию (1.0) и кодировку (utf-8) xml. Далее идет собственно содержимое документа.

XML-документ должен иметь один единственный корневой элемент, внутрь которого помещаются все остальные элементы. В данном случае таким элементом является элемент <users>. Внутри корневого элемента <users> задан набор элементов <user>. Вне корневого элемента мы не можем разместить элементы user.

Каждый элемент определяется с помощью открывающего и закрывающего тегов, например, <user> и </user>, внутри которых помещается значение или содержимое элементов. Также элемент может иметь сокращенное объявление: <user /> - в конце элемента помещается слеш.

Элемент может иметь вложенные элементы и атрибуты. В данном случае каждый элемент user имеет два вложенных элемента company и age и атрибут name.

Атрибуты определяются в теле элемента и имеют следующую форму: название=“значение”. Например, <user name="Bill Gates">, в данном случае атрибут называется name и имеет значение Bill Gates

Внутри простых элементов помещается их значение. Например, <company>Google</company> - элемент company имеет значение Google.

Названия элементов являются регистрозависимыми, поэтому <company> и <COMPANY> будут представлять разные элементы.

Таким образом, весь список Users из кода C# сопоставляется с корневым элементом <users>, каждый объект User - с элементом <user>, а каждое свойство объекта User - с атрибутом или вложенным элементом элемента <user>

Что использовать для свойств - вложенные элементы или атрибуты? Это вопрос предпочтений - мы можем использовать как атрибуты, так и вложенные элементы. Так, в предыдущем примере вполне можно использовать вместо атрибута вложенный элемент:

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

<users>

<user>

<name>Bill Gates</name>

<company>Microsoft</company>

<age>48</age>

</user>

<user>

<name>Larry Page</name>

<company>Google</company>

<age>48</age>

</user>

</users>

Теперь рассмотрим основные подходы для работы с XML, которые имеются в C#.

Работа с XML с помощью классов System.Xml

Для работы с XML в C# можно использовать несколько подходов. В первых версиях фреймворка основной функционал работы с XML предоставляло пространство имен System.Xml. В нем определен ряд классов, которые позволяют манипулировать xml-документом:

* XmlNode: представляет узел xml. В качестве узла может использоваться весь документ, так и отдельный элемент
* XmlDocument: представляет весь xml-документ
* XmlElement: представляет отдельный элемент. Наследуется от класса XmlNode
* XmlAttribute: представляет атрибут элемента
* XmlText: представляет значение элемента в виде текста, то есть тот текст, который находится в элементе между его открывающим и закрывающим тегами
* XmlComment: представляет комментарий в xml
* XmlNodeList: используется для работы со списком узлов

Ключевым классом, который позволяет манипулировать содержимым xml, является XmlNode, поэтому рассмотрим некоторые его основные методы и свойства:

* Свойство Attributes возвращает объект XmlAttributeCollection, который представляет коллекцию атрибутов
* Свойство ChildNodes возвращает коллекцию дочерних узлов для данного узла
* Свойство HasChildNodes возвращает true, если текущий узел имеет дочерние узлы
* Свойство FirstChild возвращает первый дочерний узел
* Свойство LastChild возвращает последний дочерний узел
* Свойство InnerText возвращает текстовое значение узла
* Свойство InnerXml возвращает всю внутреннюю разметку xml узла
* Свойство Name возвращает название узла. Например, <user> - значение свойства Name равно “user”
* Свойство ParentNode возвращает родительский узел у текущего узла

Применим эти классы и их функционал. И вначале для работы с xml создадим новый файл. Назовем его users.xml и определим в нем следующее содержание:

<?xmlversion="1.0" encoding="utf-8"?>

<users>

<username="Bill Gates">

<company>Microsoft</company>

<age>48</age>

</user>

<username="Larry Page">

<company>Google</company>

<age>42</age>

</user>

</users>

Теперь пройдемся по этому документу и выведем его данные на консоль:

using System.Xml;

class Program

{

static void Main(string[] args)

{

XmlDocument xDoc = new XmlDocument();

xDoc.Load("D://users.xml");

// получим корневой элемент

XmlElement xRoot = xDoc.DocumentElement;

// обход всех узлов в корневом элементе

foreach(XmlNode xnode in xRoot)

{

// получаем атрибут name

if(xnode.Attributes.Count>0)

{

XmlNode attr = xnode.Attributes.GetNamedItem("name");

if (attr!=null)

Console.WriteLine(attr.Value);

}

// обходим все дочерние узлы элемента user

foreach(XmlNode childnode in xnode.ChildNodes)

{

// если узел - company

if(childnode.Name=="company")

{

Console.WriteLine($"Компания: {childnode.InnerText}");

}

// если узел age

if (childnode.Name == "age")

{

Console.WriteLine($"Возраст: {childnode.InnerText}");

}

}

Console.WriteLine();

}

Console.Read();

}

}

В итоге я получу следующий вывод на консоли:

Чтобы начать работу с документом xml, нам надо создать объект XmlDocument и затем загрузить в него xml-файл: xDoc.Load(“users.xml”);

При разборе xml для начала мы получаем корневой элемент документа с помощью свойства xDoc.DocumentElement. Далее уже происходит собственно разбор узлов документа.

В цикле foreach(XmlNode xnode in xRoot) пробегаемся по всем дочерним узлам корневого элемента. Так как дочерние узлы представляют элементы <user>, то мы можем получить их атрибуты: XmlNode attr = xnode.Attributes.GetNamedItem(“name”); и вложенные элементы: foreach(XmlNode childnode in xnode.ChildNodes)

Чтобы определить, что за узел перед нами, мы можем сравнить его название: if(childnode.Name==“company”)

Подобным образом мы можем создать объекты User по данным из xml:

using System;

using System.Collections.Generic;

using System.Xml;

namespace HelloApp

{

class User

{

public string Name { get; set; }

public int Age { get; set; }

public string Company { get; set; }

}

class Program

{

static void Main(string[] args)

{

List<User> users = new List<User>();

XmlDocument xDoc = new XmlDocument();

xDoc.Load("D://users.xml");

XmlElement xRoot = xDoc.DocumentElement;

foreach (XmlElement xnode in xRoot)

{

User user = new User();

XmlNode attr = xnode.Attributes.GetNamedItem("name");

if (attr != null)

user.Name = attr.Value;

foreach (XmlNode childnode in xnode.ChildNodes)

{

if (childnode.Name == "company")

user.Company = childnode.InnerText;

if (childnode.Name == "age")

user.Age = Int32.Parse(childnode.InnerText);

}

users.Add(user);

}

foreach (User u in users)

Console.WriteLine($"{u.Name} ({u.Company}) - {u.Age}");

Console.Read();

}

}

}

Изменение XML-документа

Для редактирования xml-документа (изменения, добавления, удаления элементов) мы можем воспользоваться методами класса XmlNode:

* AppendChild: добавляет в конец текущего узла новый дочерний узел
* InsertAfter: добавляет новый узел после определенного узла
* InsertBefore: добавляет новый узел до определенного узла
* RemoveAll: удаляет все дочерние узлы текущего узла
* RemoveChild: удаляет у текущего узла один дочерний узел и возвращает его

Класс XmlElement, унаследованный от XmlNode, добавляет еще ряд методов, которые позволяют создавать новые узлы:

* CreateNode: создает узел любого типа
* CreateElement: создает узел типа XmlDocument
* CreateAttribute: создает узел типа XmlAttribute
* CreateTextNode: создает узел типа XmlTextNode
* CreateComment: создает комментарий

Возьмем xml-документ из прошлой темы и добавим в него новый элемент:

XmlDocument xDoc = new XmlDocument();

xDoc.Load("D://users.xml");

XmlElement xRoot = xDoc.DocumentElement;

// создаем новый элемент user

XmlElement userElem = xDoc.CreateElement("user");

// создаем атрибут name

XmlAttribute nameAttr = xDoc.CreateAttribute("name");

// создаем элементы company и age

XmlElement companyElem = xDoc.CreateElement("company");

XmlElement ageElem = xDoc.CreateElement("age");

// создаем текстовые значения для элементов и атрибута

XmlText nameText = xDoc.CreateTextNode("Mark Zuckerberg");

XmlText companyText = xDoc.CreateTextNode("Facebook");

XmlText ageText = xDoc.CreateTextNode("30");

//добавляем узлы

nameAttr.AppendChild(nameText);

companyElem.AppendChild(companyText);

ageElem.AppendChild(ageText);

userElem.Attributes.Append(nameAttr);

userElem.AppendChild(companyElem);

userElem.AppendChild(ageElem);

xRoot.AppendChild(userElem);

xDoc.Save("D://users.xml");

Добавление элементов происходит по одной схеме. Сначала создаем элемент (xDoc.CreateElement(“user”)). Если элемент сложный, то есть содержит в себе другие элементы, то создаем эти элементы. Если элемент простой, содержащий внутри себя некоторое текстовое значение, то создаем этот текст (XmlText companyText = xDoc.CreateTextNode(“Facebook”);).

Затем все элементы добавляются в основной элемент user, а тот добавляется в корневой элемент (xRoot.AppendChild(userElem);).

Чтобы сохранить измененный документ на диск, используем метод Save: xDoc.Save(“users.xml”)

После этого в xml-файле появится следующий элемент:

<username="Mark Zuckerberg">

<company>Facebook</company>

<age>30</age>

</user>

Удаление узлов

Удалим первый узел xml-документа:

XmlDocument xDoc = new XmlDocument();

xDoc.Load("D://users.xml");

XmlElement xRoot = xDoc.DocumentElement;

XmlNode firstNode = xRoot.FirstChild;

xRoot.RemoveChild(firstNode);

xDoc.Save("D://users.xml");

XPath

XPath представляет язык запросов в XML. Он позволяет выбирать элементы, соответствующие определенному селектору.

Рассмотрим некоторые наиболее распространенные селекторы:  
.  
выбор текущего узла  
..  
выбор родительского узла  
\*  
выбор всех дочерних узлов текущего узла  
user  
выбор всех узлов с определенным именем, в данном случае с именем “user”  
@name  
выбор атрибута текущего узла, после знака @ указывается название атрибута (в данном случае “name”)  
@+  
выбор всех атрибутов текущего узла  
element[3]  
выбор определенного дочернего узла по индексу, в данном случае третьего узла  
//user  
выбор в документе всех узлов с именем “user”  
user[@name='Bill Gates']  
выбор элементов с определенным значением атрибута. В данном случае выбираются все элементы “user” с атрибутом name=‘Bill Gates’  
user[company='Microsoft']  
выбор элементов с определенным значением вложенного элемента. В данном случае выбираются все элементы “user”, у которых дочерний элемент “company” имеет значение ‘Microsoft’  
//user/company  
выбор в документе всех узлов с именем “company”, которые находятся в элементах “user”

Действие запросов XPath основано на применении двух методов класса XmlElement:

* SelectSingleNode(): выбор единственного узла из выборки. Если выборка по запросу содержит несколько узлов, то выбирается первый
* SelectNodes(): выборка по запросу коллекции узлов в виде объекта XmlNodeList

Для запросов возьмем xml-документ из прошлых тем:

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

<users>

<username="Bill Gates">

<company>Microsoft</company>

<age>48</age>

</user>

<username="Larry Page">

<company>Google</company>

<age>42</age>

</user>

</users>

Теперь выберем все узлы корневого элемента, то есть все элементы user:

XmlDocument xDoc = new XmlDocument();

xDoc.Load("D://users.xml");

XmlElement xRoot = xDoc.DocumentElement;

// выбор всех дочерних узлов

XmlNodeList childnodes = xRoot.SelectNodes("\*");

foreach (XmlNode n in childnodes)

Console.WriteLine(n.OuterXml);

Выберем все узлы <user>:

XmlNodeList childnodes = xRoot.SelectNodes("user");

Выведем на консоль значения атрибутов name у элементов user:

XmlNodeList childnodes = xRoot.SelectNodes("user");

foreach (XmlNode n in childnodes)

Console.WriteLine(n.SelectSingleNode("@name").Value);

Результатом выполнения будет следующий вывод:

Bill Gates

Larry Page

Выберем узел, у которого атрибут name имеет значение “Bill Gates”:

XmlNode childnode = xRoot.SelectSingleNode("user[@name='Bill Gates']");

if (childnode != null)

Console.WriteLine(childnode.OuterXml);

Выберем узел, у которого вложенный элемент “company” имеет значение “Microsoft”:

XmlNode childnode = xRoot.SelectSingleNode("user[company='Microsoft']");

if (childnode != null)

Console.WriteLine(childnode.OuterXml);

Допустим, нам надо получить только компании. Для этого надо осуществить выборку вниз по иерархии элементов:

XmlNodeList childnodes = xRoot.SelectNodes("//user/company");

foreach (XmlNode n in childnodes)

Console.WriteLine(n.InnerText);

Linq to Xml. Создание документа XML

Еще один подход к работе с Xml представляет технология LINQ to XML. Вся функциональность LINQ to XML содержится в пространстве имен System.Xml.Linq. Рассмотрим основные классы этого пространства имен:

* XAttribute: представляет атрибут xml-элемента
* XComment: представляет комментарий
* XDocument: представляет весь xml-документ
* XElement: представляет отдельный xml-элемент

Ключевым классом является XElement, который позволяет получать вложенные элементы и управлять ими. Среди его методов можно отметить следующие:

* Add(): добавляет новый атрибут или элемент
* Attributes(): возвращает коллекцию атрибутов для данного элемента
* Elements(): возвращает все дочерние элементы данного элемента
* Remove(): удаляет данный элемент из родительского объекта
* RemoveAll(): удаляет все дочерние элементы и атрибуты у данного элемента

Итак, используем функциональность LINQ to XML и создадим новый XML-документ:

XDocument xdoc = new XDocument();

// создаем первый элемент

XElement iphone6 = new XElement("phone");

// создаем атрибут

XAttribute iphoneNameAttr = new XAttribute("name", "iPhone 6");

XElement iphoneCompanyElem = new XElement("company", "Apple");

XElement iphonePriceElem = new XElement("price", "40000");

// добавляем атрибут и элементы в первый элемент

iphone6.Add(iphoneNameAttr);

iphone6.Add(iphoneCompanyElem);

iphone6.Add(iphonePriceElem);

// создаем второй элемент

XElement galaxys5 = new XElement("phone");

XAttribute galaxysNameAttr = new XAttribute("name", "Samsung Galaxy S5");

XElement galaxysCompanyElem = new XElement("company", "Samsung");

XElement galaxysPriceElem = new XElement("price", "33000");

galaxys5.Add(galaxysNameAttr);

galaxys5.Add(galaxysCompanyElem);

galaxys5.Add(galaxysPriceElem);

// создаем корневой элемент

XElement phones = new XElement("phones");

// добавляем в корневой элемент

phones.Add(iphone6);

phones.Add(galaxys5);

// добавляем корневой элемент в документ

xdoc.Add(phones);

//сохраняем документ

xdoc.Save("phones.xml");

Чтобы создать документ, нам нужно создать объект класса XDocument. Это объект самого верхнего уровня в хml-документе.

Элементы создаются с помощью конструктора класса XElement. Конструктор имеет ряд перегруженных версий. Первый параметр конструктора передает название элемента, например, phone. Второй параметр передает значение этого элемента.

Создание атрибута аналогично созданию элемента. Затем все атрибуты и элементы добавляются в элементы phone с помощью метода Add().

Так как документ xml должен иметь один корневой элемент, то затем все элементы phone добавляются в один контейнер - элемент phones.

В конце корневой элемент добавляется в объект XDocument, и этот объект сохраняется на диске в xml-файл с помощью метода Save().

Если мы откроем сохраненный файл phones.xml, то увидим в нем следующее содержание:

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

<phones>

<phone name="iPhone 6">

<company>Apple</company>

<price>40000</price>

</phone>

<phone name="Samsung Galaxy S5">

<company>Samsung</company>

<price>33000</price>

</phone>

</phones>

Конструктор класса XElement позволяют задать набор объектов, которые будут входить в элемент. И предыдущий пример мы могли бы сократить следующим способом:

XDocument xdoc = new XDocument(new XElement("phones",

new XElement("phone",

new XAttribute("name", "iPhone 6"),

new XElement("company", "Apple"),

new XElement("price", "40000")),

new XElement("phone",

new XAttribute("name", "Samsung Galaxy S5"),

new XElement("company", "Samsung"),

new XElement("price", "33000"))));

xdoc.Save("phones.xml");

Выборка элементов в LINQ to XML

Возьмем xml-файл, созданный в прошлой теме:

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

<phones>

<phonename="iPhone 6">

<company>Apple</company>

<price>40000</price>

</phone>

<phonename="Samsung Galaxy S5">

<company>Samsung</company>

<price>33000</price>

</phone>

</phones>

Переберем его элементы и выведем их значения на консоль:

XDocument xdoc = XDocument.Load("phones.xml");

foreach (XElement phoneElement in xdoc.Element("phones").Elements("phone"))

{

XAttribute nameAttribute = phoneElement.Attribute("name");

XElement companyElement = phoneElement.Element("company");

XElement priceElement = phoneElement.Element("price");

if (nameAttribute != null && companyElement!=null && priceElement!=null)

{

Console.WriteLine($"Смартфон: {nameAttribute.Value}");

Console.WriteLine($"Компания: {companyElement.Value}");

Console.WriteLine($"Цена: {priceElement.Value}");

}

Console.WriteLine();

}

И мы получим следующий вывод:

Чтобы начать работу с имеющимся xml-файлом, надо сначала загрузить его с помощью статического метода XDocument.Load(), в который передается путь к файлу.

Поскольку xml хранит иерархически выстроенные элементы, то и для доступа к элементам надо идти начиная с высшего уровня в этой иерархии и далее вниз. Так, для получения элементов phone и доступа к ним надо сначала обратиться к корневому элементу, а через него уже к элементам phone: xdoc.Element(“phones”).Elements(“phone”)

Метод Element(“имя\_элемента”) возвращает первый найденный элемент с таким именем. Метод Elements(“имя\_элемента”) возвращает коллекцию одноименных элементов. В данном случае мы получаем коллекцию элементов phone и поэтому можем перебрать ее в цикле.

Спускаясь дальше по иерархии вниз, мы можем получить атрибуты или вложенные элементы, например, XElement companyElement = phoneElement.Element(“company”)

Значение простых элементов, которые содержат один текст, можно получить с помощью свойства Value: string company = phoneElement.Element(“company”).Value

Сочетая операторы Linq и LINQ to XML можно довольно просто извлечь из документа данные и затем обработать их. Например, имеется следующий класс:

class Phone

{

public string Name { get; set; }

public string Price { get; set; }

}

Создадим на основании данных в xml объекты этого класса:

XDocument xdoc = XDocument.Load("phones.xml");

var items = from xe in xdoc.Element("phones").Elements("phone")

where xe.Element("company").Value=="Samsung"

select new Phone

{

Name = xe.Attribute("name").Value,

Price = xe.Element("price").Value

};

foreach (var item in items)

Console.WriteLine($"{item.Name} - {item.Price}");

Изменение документа в LINQ to XML

Возьмем xml-файл из прошлой темы:

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

<phones>

<phone name="iPhone 6">

<company>Apple</company>

<price>40000</price>

</phone>

<phone name="Samsung Galaxy S5">

<company>Samsung</company>

<price>33000</price>

</phone>

</phones>

И отредактируем его содержимое:

using System;

using System.Xml.Linq;

using System.Linq;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

XDocument xdoc = XDocument.Load("phones.xml");

XElement root = xdoc.Element("phones");

foreach (XElement xe in root.Elements("phone").ToList())

{

// изменяем название и цену

if (xe.Attribute("name").Value == "Samsung Galaxy S5")

{

xe.Attribute("name").Value = "Samsung Galaxy Note 4";

xe.Element("price").Value = "31000";

}

//если iphone - удаляем его

else if (xe.Attribute("name").Value == "iPhone 6")

{

xe.Remove();

}

}

// добавляем новый элемент

root.Add(new XElement("phone",

new XAttribute("name", "Nokia Lumia 930"),

new XElement("company", "Nokia"),

new XElement("price", "19500")));

xdoc.Save("pnones1.xml");

// выводим xml-документ на консоль

Console.WriteLine(xdoc);

Console.Read();

}

}

}

Для изменения содержимого простых элементов и атрибутов достаточно изменить их свойство Value: xe.Element(“price”).Value = “31000”

Если же нам надо редактировать сложный элемент, то мы можем использовать комбинацию методов Add/Remove для добавления и удаления вложенных элементов.

В результате сформируется и сохранится на диск новый документ: