**Файлы. Потоки. Сериализация.**

**1.**

using System;

using System.Linq;

using System.IO;

// 1a. В текстовом файле "data.csv" записана информация о прямоугольных деталях в формате: "длина;ширина".

// Найти суммарную площадь деталей.

namespace files {

class MainClass {

public static void Main(string[] args) {

// Открываем файл

// При необходимости указываем путь к файлу: @"d:\data.csv"

using (StreamReader data = File.OpenText(@"data.csv")) {

double S = 0.0; // Сумма площадей

while (!data.EndOfStream) { // Пока не конец файла

string line = data.ReadLine(); // Читаем строку

double[] numbers;

try {

// Разбиваем строку по ; и преобразуем в массив чисел

numbers = (from x in line.Split(';') select double.Parse(x.Trim())).ToArray();

} catch(FormatException) {

continue; // Строки, не содержащие чисел, игнорируем

}

try {

S += numbers[0] \* numbers[1];

} catch(IndexOutOfRangeException) {

continue; // Если чисел меньше двух, пропускаем строку

}

}

Console.WriteLine("S = {0}", S);

}

}

}

}

using System;

using System.Linq;

using System.IO;

// 1b. Написать программу, записывающую в двоичный файл N целых чисел

// и находящую сумму целых чисел в двоичном файле.

// В этом примере будем записывать случайные числа до 100

namespace files {

class MainClass {

public static void Main(string[] args) {

const int N = 5;

// Записываем числа в файл

using (FileStream data = File.Create(@"data.bin")) {

var rnd = new Random();

for (int i = 0; i < N; i++) {

int number = rnd.Next(100);

Console.WriteLine(number);

// Преобразуем число в массив байтов

byte[] number\_bytes = BitConverter.GetBytes(number);

// Записываем байты в файл

data.Write(number\_bytes, 0, number\_bytes.Length);

}

}

// Суммируем числа из файла

using (FileStream data = File.OpenRead(@"data.bin")) {

int S = 0;

// Буфер для хранения прочитанных чисел

byte[] buffer = new byte[sizeof(int)];

while(true) {

// Читаем следующее число

int read\_bytes = data.Read(buffer, 0, sizeof(int));

// Если данные закончились, выходим

if (read\_bytes < sizeof(int))

break;

// Преобразуем байты в int

int number = BitConverter.ToInt32(buffer, 0);

S += number;

}

Console.WriteLine("S = {0}", S);

}

}

}

}

**2.**

using System;

using System.IO;

// 2. Написать программу, дописывающую слева к имени всех файлов с расширением txt в указанной папке год создания.

namespace files {

class MainClass {

public static void Main(string[] args) {

// Получаем информацию о папке

// Например, @"d:\103\texts\"

DirectoryInfo dir = new DirectoryInfo(@"texts");

// Получаем список файлов с расширением txt

FileInfo[] txt\_files = dir.GetFiles("\*.txt");

// Обрабатываем каждый файл

foreach (var file in txt\_files) {

// Получаем год создания

var year = file.CreationTime.Year;

// Дописываем год к имени

file.MoveTo(

Path.Combine(

dir.FullName,

year.ToString() + " " + file.Name

) // Объединяем путь и новое имя файла

);

}

}

}

}

**3.**

using System;

using System.IO;

using System.Collections.Generic;

using System.Xml.Serialization;

// 3. Написать программу, запрашивающую список студентов (определяется фамилий и именем)

// и записывающую его в XML-файл. Написать программу, выводящую список из XML-файла.

namespace files {

public class Student {

public string Name;

public string Surname;

}

public class StudentList {

public List<Student> Students = new List<Student>();

}

class MainClass {

public static void Main(string[] args) {

const string filename = @"students.xml";

XmlSerializer xml = new XmlSerializer(typeof(StudentList));

// Вместо запроса студентов с клавиатуры просто создадим пару объектов сами

var students = new StudentList();

students.Students.Add(

new Student {Name = "Иван", Surname = "Иванов"}

);

students.Students.Add(

new Student {Name = "Петр", Surname = "Петров"}

);

// Сериализуем список

using (FileStream file = File.Create(filename)) {

xml.Serialize(file, students);

}

// Читаем список

using (FileStream file = File.OpenRead(filename)) {

StudentList students2 = xml.Deserialize(file) as StudentList;

foreach (var s in students2.Students) {

Console.WriteLine("{0} {1}", s.Name, s.Surname);

}

}

}

}

}

**Чтение и запись файла. Класс FileStream**

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

Рассмотрим наиболее важные его свойства и методы:

* Свойство **Length**: возвращает длину потока в байтах
* Свойство **Position**: возвращает текущую позицию в потоке
* Метод **Read**: считывает данные из файла в массив байтов. Принимает три параметра: int Read(byte[] array, int offset, int count) и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:
* 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.
* Метод **long Seek(long offset, SeekOrigin origin)**: устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
* Метод **Write**: записывает в файл данные из массива байтов. Принимает три параметра: Write(byte[] array, int offset, int count)
* 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

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 полезен для извлечения определенных порций информации и ее обработки.

Пример считывания-записи в текстовый файл:

using System;

using System.IO;

using System.Text;

class Program

{

public static void Main()

{

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

string text = Console.ReadLine();

// запись в файл

using (FileStream fstream = new FileStream(@"D:\mydocs\myFile.txt", FileMode.Create))

{

// преобразуем строку в байты

byte[] array = 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(@"D:\mydocs\myFile.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 = Encoding.Default.GetString(array);

Console.WriteLine("Текст из файла: {0}", textFromFile);

}

}

}

Разберем этот пример. 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.

Объект fstream создается двумя разными способами: через конструктор и через один из статических методов класса File.

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 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.

Статический метод OpenRead класса File открывает файл для чтения и возвращает объект FileStream.

Конструктор класса FileStream также имеет ряд перегруженных версий, позволяющий более точно настроить создаваемый объект.

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 myFile. 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.

**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 определены специальные классы: **StreamReader** и **StreamWriter**.

**Чтение из файла и StreamReader**

Класс StreamReader позволяет нам легко считывать весь текст или отдельные строки из текстового файла. Среди его методов можно выделить следующие:

* **Close**: закрывает считываемый файл и освобождает все ресурсы.
* **Peek**: возвращает следующий доступный символ, если символов больше нет, то возвращает -1.
* **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.
* **ReadLine**: считывает одну строку в файле.
* **ReadToEnd**: считывает весь текст из файла.

Считаем текст из файла различными способами:

using System;

using System.IO;

using System.Text;

class Program

{

public static void Main()

{

string path = @"D:\mydocs\myFile.txt";

try

{

Console.WriteLine("\*\*\*\*\*\*считываем весь файл\*\*\*\*\*\*\*\*");

using (StreamReader sr = new StreamReader(path))

{

Console.WriteLine(sr.ReadToEnd());

}

Console.WriteLine();

Console.WriteLine("\*\*\*\*\*\*считываем построчно\*\*\*\*\*\*\*\*");

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

{

string line;

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

{

Console.WriteLine(line);

}

}

Console.WriteLine();

Console.WriteLine("\*\*\*\*\*\*считываем блоками\*\*\*\*\*\*\*\*");

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

{

char[] array = new char[4];

// считываем 4 символа

sr.Read(array, 0, 4);

Console.WriteLine(array);

}

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

Console.ReadLine();

}

}

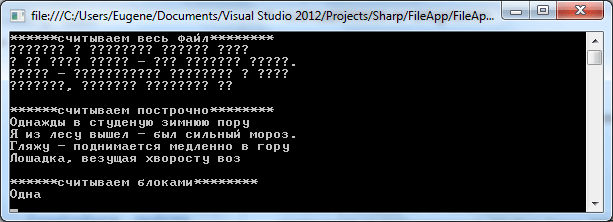
Как и в случае с классом FileStream здесь используется конструкция using.

В первом случае мы разом считываем весь текст с помощью метода ReadToEnd().

Во втором 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.

В третьем случае считываем в массив четыре символа.

Обратите внимание, что в последних двух случаях в конструкторе StreamReader указывалась кодировка Encoding.Default. Свойство Default класса Encoding получает кодировку для текущей кодовой страницы ANSI. Также через другие свойства мы можем указать другие кодировки. Если кодировка не указана, то при чтении используется UTF8. Иногда важно указывать кодировку, так как она может отличаться от UTF8, и тогда мы получим некорректный вывод. Например:



**Запись в файл и StreamWriter**

Для записи в текстовый файл используется класс StreamWriter. Свою функциональность он реализует через следующие методы:

* **Close**: закрывает записываемый файл и освобождает все ресурсы
* **Flush**: записывает в файл оставшиеся в буфере данные и очищает буфер.
* **Write**: записывает в файл данные простейших типов, как int, double, char, string и т.д.
* **WriteLine**: также записывает данные, только после записи добавляет в файл символ окончания строки

Рассмотрим запись в файл на примере:

using System;

using System.IO;

using System.Text;

class Program

{

public static void Main()

{

string readPath = @"D:\mydocs\myFile1.txt";

string writePath = @"D:\mydocs\myFile2.txt";

string text = "";

try

{

using (StreamReader sr = new StreamReader(readPath, Encoding.Default))

{

text = sr.ReadToEnd();

}

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

{

sw.WriteLine(text);

}

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

{

sw.WriteLine("Дозапись");

sw.Write(4.5);

}

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

Console.ReadLine();

}

}

Здесь сначала мы считываем файл в переменную text, а затем записываем эту переменную в файл, а затем через объект StreamWriter записываем в новый файл.

Класс StreamWriter имеет несколько конструкторов. Здесь мы использовали один из них: new StreamWriter(writePath, false, System.Text.Encoding.Default). В качестве первого параметра передается путь к записываемому файлу. Второй параметр представляет булевую переменную, которая определяет, будет файл дозаписываться или перезаписываться. Если этот параметр равен true, то новые данные добавляются в конце к уже имеющимся данным. Если false, то файл перезаписывается. И если в первом случае файл перезаписывается, то во втором делается дозапись в конец файла.

Третий параметр указывает кодировку, в которой записывается файл.