

# ПОТОКИ ВВОДУ- ВИВОДУ

---

# Абстрактный класс Stream

---

К этому моменту было показано несколько способов получения объектов `FileStream`, `StreamReader` и `StreamWriter`, но еще нужно читать данные и записывать их в файл, использующий эти типы.

- Чтобы понять, как это делается, необходимо изучить концепцию потока.

В мире манипуляций вводом-выводом поток (stream) представляет порцию данных, протекающую от источника к цели.

- Потoki предоставляют общий способ взаимодействия с последовательностью байтов, независимо от того, какого рода устройство (файл, сеть, соединение, принтер и т.п.) хранит или отображает эти байты.

В абстрактном классе `System.IO.Stream` определен набор членов, которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, файлом или областью памяти).

- Потомки класса `Stream` представляют данные как низкоуровневые потоки байт, а непосредственная работа с низкоуровневыми потоками может оказаться довольно загадочной.
- Некоторые типы, унаследованные от `Stream`, поддерживают поиск, что означает возможность получения и изменения текущей позиции в потоке.

# Члены абстрактного класса Stream

Член	Описание
CanRead CanWrite CanSeek	Определяют, поддерживает ли текущий поток чтение, поиск и/или запись
Close()	Закрывает текущий поток и освобождает все ресурсы (такие как сокеты и файловые дескрипторы), ассоциированные с текущим потоком. Внутренне этот метод является псевдонимом <code>Dispose()</code> . Поэтому закрытие потока функционально эквивалентно <i>освобождению потока</i>
Flush()	Обновляет лежащий в основе источник данных или репозиторий текущим состоянием буфера с последующей очисткой буфера. Если поток не реализует буфер, метод ничего не делает
Length	Возвращает длину потока в байтах
Position	Определяет текущую позицию в потоке
Read() ReadByte()	Читает последовательность байт (или одиночный байт) из текущего потока и перемещает текущую позицию потока вперед на количество прочитанных байтов
Seek()	Устанавливает позицию в текущем потоке
SetLength()	Устанавливает длину текущего потока
Write() WriteByte()	Записывает последовательность байтов (или одиночный байт) в текущий поток и перемещает текущую позицию вперед на количество записанных байтов

# Работа с классом FileStream

---

Класс `FileStream` предоставляет реализацию абстрактного члена `Stream` в манере, подходящей для потоковой работы с файлами.

- Это элементарный поток, и он может записывать или читать только один байт или массив байтов. Тем не менее, взаимодействовать с членами типа `FileStream` придется нечасто.
- Вместо этого, скорее всего, будут использоваться разнообразные оболочки потоков, которые облегчают работу с текстовыми данными или типами `.NET`. Однако в целях иллюстрации полезно поэкспериментировать с возможностями синхронного чтения/записи типа `FileStream`.

Предположим, что имеется консольное приложение под названием `FileStreamApp` (и в файле кода `C#` выполнен импорт пространств имен `System.IO` и `System.Text`).

- Цель заключается в записи простого текстового сообщения в новый файл по имени `myMessage.dat`.
- Однако, учитывая, что `FileStream` может работать только с низкоуровневыми байтами, придется закодировать тип `System.String` в соответствующий байтовый массив.
- К счастью, в пространстве имен `System.Text` определен тип по имени `Encoding`, который содержит члены, способные кодировать и декодировать строки в массивы байтов.

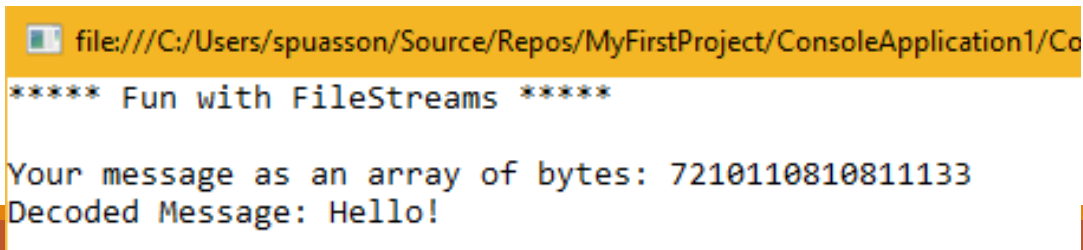
После кодирования байтовый массив сохраняется в файле с помощью метода `FileStream.Write()`. Чтобы прочитать байты обратно в память, необходимо сбросить внутреннюю позицию потока (через свойство `Position`) и вызвать метод `ReadByte()`.

- И, наконец, на консоль выводится содержимое низкоуровневого байтового массива и декодированная строка.

# Відповідний КОД

```
Console.WriteLine("***** Fun with FileStreams *****\n");
// Получить объект FileStream.
using (FileStream fStream = File.Open(@"C:\myMessage.dat", FileMode.Create))
{
    // Закодировать строку в виде массива байт.
    string msg = "Hello!";
    byte[] msgAsByteArray = Encoding.Default.GetBytes(msg);
    // Записать byte[] в файл.
    fStream.Write(msgAsByteArray, 0, msgAsByteArray.Length);
    // Сбросить внутреннюю позицию потока.
    fStream.Position = 0;
    // Прочитать типы из файла и вывести на консоль.
    Console.Write("Your message as an array of bytes: ");
    byte[] bytesFromFile = new byte[msgAsByteArray.Length];
    for (int i = 0; i < msgAsByteArray.Length; i++) {
        bytesFromFile[i] = (byte)fStream.ReadByte();
        Console.Write(bytesFromFile[i]);
    }
    // Вывести декодированные сообщения.
    Console.Write("\nDecoded Message: ");
    Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
}
Console.ReadLine();
```

В примере производится не только наполнение файла данными, но также демонстрируется основной недостаток прямой работы с типом FileStream: приходится оперировать низкоуровневыми байтами. Другие унаследованные от Stream типы работают аналогично.



file:///C:/Users/spuasson/Source/Repos/MyFirstProject/ConsoleApplication1/Co

```
***** Fun with FileStreams *****

Your message as an array of bytes: 7210110810811133
Decoded Message: Hello!
```

# Работа с классами StreamWriter и StreamReader

---

Классы StreamWriter и StreamReader удобны во всех случаях, когда нужно читать или записывать символьные данные (например, строки).

- Оба типа работают по умолчанию с символами Unicode; однако это можно изменить предоставлением правильно сконфигурированной ссылки на объект System.Text.Encoding.
- Чтобы не усложнять пример, предположим, что стандартная кодировка Unicode вполне устраивает.

Класс StreamReader, как и StringReader, унаследован от абстрактного класса по имени TextReader.

- Базовый класс предлагает очень ограниченный набор функциональности каждому из своих наследников, в частности — возможность читать и “заглядывать” (peek) в символьный поток.

Класс StreamWriter (а также и StringWriter) порожден от абстрактного базового класса по имени TextWriter.

- В этом классе определены члены, позволяющие производным типам записывать текстовые данные в заданный символьный поток.

# Основные члены TextWriter

---

Член	Описание
Close()	Этот метод закрывает средство записи и освобождает все связанные с ним ресурсы. В процессе автоматически сбрасывается буфер (опять-таки, этот член функционально эквивалентен методу <code>Dispose()</code> )
Flush()	Этот метод очищает все буферы текущего средства записи и записывает все буферизованные данные на лежащее в основе устройство, но не закрывает его
NewLine	Это свойство задает константу новой строки для унаследованного класса средства записи. По умолчанию ограничителем строки в Windows является возврат каретки, за которым следует перевод строки ( <code>\r\n</code> )
Write()	Этот перегруженный метод записывает данные в текстовый поток без добавления константы новой строки
WriteLine()	Этот перегруженный метод записывает данные в текстовый поток с добавлением константы новой строки

WriteAsync(), WriteLineAsync()

---

Последние два члена класса `TextWriter`, скорее всего, покажутся знакомыми.

- Если помните, тип `System.Console` имеет члены `Write()` и `WriteLine()`, которые выталкивают текстовые данные на стандартное устройство вывода.
- Фактически свойство `Console.In` хранит `TextWriter`, а `Console.Out` — `TextWriter`.

Унаследованный класс `StreamWriter` предоставляет соответствующую реализацию методов `Write()`, `Close()` и `Flush()`, а также определяет дополнительное свойство `AutoFlush`.

- Когда это свойство установлено в `true`, оно заставляет `StreamWriter` выталкивать данные при каждой операции записи.
- Следует отметить, что можно обеспечить более высокую производительность, установив `AutoFlush` в `false`, но при этом всегда вызывать `Close()` по завершении работы с `StreamWriter`.

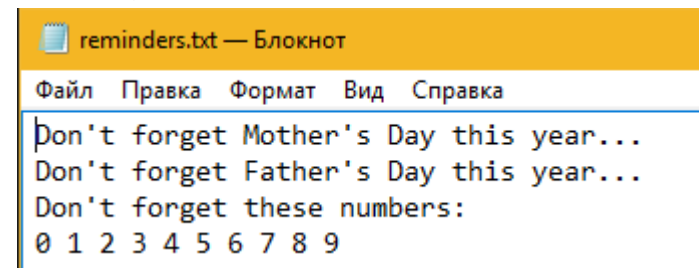


# Запись в текстовый файл

В показанном ниже методе Main () создается новый файл по имени reminders.txt с помощью метода File.CreateText ().

Используя полученный объект StreamWriter, в новый файл будут добавлены некоторые текстовые данные.

```
Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
// Получить StreamWriter и записать строковые данные.
using (StreamWriter writer = File.CreateText("reminders.txt"))
{
    writer.WriteLine("Don't forget Mother's Day this year...");
    writer.WriteLine("Don't forget Father's Day this year...");
    writer.WriteLine("Don't forget these numbers:");
    for (int i = 0; i < 10; i++)
        writer.Write(i + " ");
    // Вставить новую строку.
    writer.Write(writer.NewLine);
}
Console.WriteLine("Created file and wrote some thoughts...");
Console.ReadLine();
```



# Чтение из текстового файла.

## Основные члены TextReader

---

посмотрим, как программно прочитать данные из файла, используя соответствующий тип StreamReader.

Как вы помните, этот класс унаследован от абстрактного класса TextReader,

Член	Описание
Peek()	Возвращает следующий доступный символ, не изменяя текущей позиции средства чтения. Значение -1 указывает на достижение конца потока
Read()	Читает данные из входного потока
ReadBlock()	Читает указанное максимальное количество символов из текущего потока и записывает данные в буфер, начиная с заданного индекса
ReadLine()	Читает строку символов из текущего потока и возвращает данные в виде строки (строка null указывает на признак конца файла)
ReadToEnd()	Читает все символы от текущей позиции до конца потока и возвращает их в виде одной строки

---

Если теперь расширить текущий пример приложения, чтобы в нем применялся класс `StreamReader`, то можно будет прочитать текстовые данные из файла `reminders.txt`:

```
// Прочитать данные из файла.
Console.WriteLine("Here are your thoughts:\n");
using (StreamReader sr = File.OpenText("reminders.txt"))
{
    string input = null;
    while ((input = sr.ReadLine()) != null)
    {
        Console.WriteLine(input);
    }
}
Console.ReadLine();
```

 `file:///C:/Users/spuasson/Source/Repos/MyFirstProject/ConsoleApplicati`

\*\*\*\*\* Fun with StreamWriter / StreamReader \*\*\*\*\*

Created file and wrote some thoughts...

Here are your thoughts:

Don't forget Mother's Day this year...

Don't forget Father's Day this year...

Don't forget these numbers:

0 1 2 3 4 5 6 7 8 9

# Прямое создание экземпляров классов StreamWriter/StreamReader

---

Одним из сбивающих с толку аспектов работы с типами, которые входят в пространство имен System.IO, является то, что одного и того же результата часто можно достичь с использованием разных подходов.

- Например, ранее уже было показано, что с помощью метода CreateText() можно получить объект StreamWriter с типом File или FileInfo.
- В действительности есть и еще один способ работы с объектами StreamWriter и StreamReader: создавать их напрямую.
- Например, текущее приложение можно было бы переделать следующим образом:

```
Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
// Получить StreamWriter и записать строковые данные.
using (StreamWriter writer = new StreamWriter("reminders.txt"))
{
    ...
}
// Прочитать данные из файла.
using (StreamReader sr = new StreamReader("reminders.txt"))
{
    ...
}
```

# Работа с классами StringWriter и StringReader

---

Классы StringWriter и StringReader можно использовать для трактовки текстовой информации как потока символов, находящихся в памяти.

- Это полезно, когда требуется добавить символьную информацию к лежащему в основе буферу.
- Для иллюстрации в следующем консольном приложении (по имени StringReaderWriterApp) блок строковых данных записывается в объект StringWriter вместо файла на локальном жестком диске:

```
Console.WriteLine("***** Fun with StringWriter / StringReader *****\n");  
// Создать StringWriter и записать символьные данные в память.  
using (StringWriter strWriter = new StringWriter())  
{  
    strWriter.WriteLine("Don't forget Mother's Day this year...");  
    // Получить копию содержимого (хранящегося в строке)  
    // и вывести на консоль.  
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);  
}  
Console.ReadLine();
```

---

Поскольку и `StringWriter`, и `StreamWriter` порождены от одного и того же базового класса (`TextWriter`), логика записи в какой-то мере похожа.

- Однако, учитывая природу `StringWriter`, имейте в виду, что этот класс позволяет использовать метод `GetStringBuilder()` для извлечения объекта `System.Text.StringBuilder`:

```
using (StringWriter strWriter = new StringWriter())
{
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
    // Получить внутренний StringBuilder.
    StringBuilder sb = strWriter.GetStringBuilder();
    sb.Insert(0, "Hey!! ");
    Console.WriteLine("-> {0}", sb.ToString());
    sb.Remove(0, "Hey!! ".Length);
    Console.WriteLine("-> {0}", sb.ToString());
}
```

---

Когда необходимо выполнить чтение из потока строковых данных, используйте соответствующий тип `StringReader`, который (как и можно было ожидать) функционирует идентично классу `StreamReader`.

- На самом деле в классе `StringReader` всего лишь переопределяются унаследованные члены для чтения из блока символьных данных, а не из файла:

```
using (StringWriter strWriter = new StringWriter()) {
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
    // Читать данные из StringWriter.
    using (StringReader strReader = new StringReader(strWriter.ToString())) {
        string input = null;
        while ((input = strReader.ReadLine()) != null)
        {
            Console.WriteLine(input);
        }
    }
}
```

# Работа с классами BinaryWriter и BinaryReader

---

BinaryWriter и BinaryReader являются прямыми наследниками System.Object.

- Эти типы позволяют читать и записывать дискретные типы данных в потоки в компактном двоичном формате.
- В классе BinaryWriter определен многократно перегруженный метод Write () для помещения типов данных в лежащий в основе поток.
- В дополнение к Write (), класс BinaryWriter предоставляет дополнительные члены, позволяющие получать или устанавливать объекты унаследованных от Stream типов, а также поддерживает произвольный доступ к данным

Член	Описание
BaseStream	Это свойство, предназначенное только для чтения, обеспечивает доступ к лежащему в основе потоку, используемому объектом BinaryWriter
Close ()	Этот метод закрывает двоичный поток
Flush ()	Этот метод выталкивает буфер двоичного потока
Seek ()	Этот метод устанавливает позицию в текущем потоке
Write ()	Этот метод записывает значение в текущий поток



# Основные члены класса BinaryReader

---

Член	Описание
BaseStream	Это свойство, предназначенное только для чтения, обеспечивает доступ к лежащему в основе потоку, используемому объектом <code>BinaryReader</code>
<code>Close()</code>	Этот метод закрывает двоичный поток
<code>PeekChar()</code>	Этот метод возвращает следующий доступный символ без перемещения текущей позиции потока
<code>Read()</code>	Этот метод читает заданный набор байт или символов и сохраняет их в переданном ему массиве
<code>ReadXXXX()</code>	В классе <code>BinaryReader</code> определено множество методов чтения, которые извлекают из потока объекты различных типов ( <code>ReadBoolean()</code> , <code>ReadByte()</code> , <code>ReadInt32()</code> и т.д.)

---

```
Console.WriteLine("***** Fun with Binary Writers / Readers *****\n");
// Open a binary writer for a file.
FileInfo f = new FileInfo("BinFile.dat");
using (BinaryWriter bw = new BinaryWriter(f.OpenWrite()))
{
    // Print out the type of BaseStream.
    // (System.IO.FileStream in this case).
    Console.WriteLine("Base stream is: {0}", bw.BaseStream);
    // Create some data to save in the file.
    double aDouble = 1234.67;
    int anInt = 34567;
    string aString = "A, B, C";
    // Write the data.
    bw.Write(aDouble);
    bw.Write(anInt);
    bw.Write(aString);
}
Console.WriteLine("Done!");
Console.ReadLine();
```

В примере объекты данных разных типов записываются в файл \*.dat

Обратите внимание, что объект FileStream, возвращенный методом FileInfo.OpenWrite(), передается конструктору типа BinaryWriter.

Используя эту технику, очень просто организовать по уровням поток перед записью данных.

- Имейте в виду, что конструктор BinaryWriter принимает любой тип, унаследованный от Stream (например, FileStream, MemoryStream или BufferedStream).

Таким образом, если необходимо записать двоичные данные в память, просто используйте объект MemoryStream.

---

Для чтения данных из файла BinFile.dat в классе BinaryReader предлагается ряд опций.

- Например, ниже будут вызываться различные члены, выполняющие чтение, для извлечения каждого фрагмента данных из файлового потока:

```
static void Main(string[] args)
{
    ...
    FileInfo f = new FileInfo("BinFile.dat");
    ...
    // Читать двоичные данные из потока.
    using(BinaryReader br = new BinaryReader(f.OpenRead()))
    {
        Console.WriteLine(br.ReadDouble());
        Console.WriteLine(br.ReadInt32());
        Console.WriteLine(br.ReadString());
    }
    Console.ReadLine();
}
```

# Программное отслеживание файлов

---

следующая задача заключается в исследовании роли класса `FileSystemWatcher`.

- Этот тип полезен, когда требуется программно отслеживать состояние файлов в системе.
- В частности, с помощью `FileSystemWatcher` можно организовать мониторинг файлов на предмет любых действий, указанных в перечислении `System.IO.NotifyFilters`.

```
public enum NotifyFilters
{
    Attributes, CreationTime,
    DirectoryName, FileName,
    LastAccess, LastWrite,
    Security, Size,
}
```

---

Первый шаг, который необходимо предпринять при работе с типом `FileSystemWatcher` — это установить свойство `Path`, чтобы оно указывало имя (и местоположение) каталога, содержащего файлы, которые нужно отслеживать, а также свойство `Filter`, определяющее расширения отслеживаемых файлов.

В настоящий момент можно выбрать обработку событий `Changed`, `Created` и `Deleted` — все они работают в сочетании с делегатом `FileSystemEventHandler`. Этот делегат может вызывать любой метод, соответствующий следующей сигнатуре:

```
// Делегат FileSystemEventHandler должен указывать
```

```
// на метод, соответствующий следующей сигнатуре.
```

```
void MyNotificationHandler(object source, FileSystemEventArgs e)
```

---

Событие `RenamedEventArgs` может быть обработано делегатом типа `EventHandler`, который может вызывать методы с такой сигнатурой:

```
// Делегат EventHandler<EventArgs> должен указывать
```

```
// на метод, соответствующий следующей сигнатуре.
```

```
void MyNotificationHandler(object source, EventArgs e)
```

Хотя для обработки каждого события можно было бы применить традиционный синтаксис делегат/событие, вы, несомненно, воспользуетесь синтаксисом лямбда-выражений (как это сделано в загружаемом коде этого проекта).

Чтобы проиллюстрировать процесс мониторинга файлов, предположим, что на диске C: создан новый каталог по имени `MyFolder`, содержащий различные файлы `*.txt` (с произвольными именами). Следующее консольное приложение (под названием `MyDirectoryWatcher`) будет выполнять мониторинг файлов `*.txt` внутри каталога `MyFolder` и выводить на консоль сообщения при создании, удалении, модификации или переименовании файлов:

---

```
Console.WriteLine("***** The Amazing File Watcher App *****\n");
// Establish the path to the directory to watch.
FileSystemWatcher watcher = new FileSystemWatcher();
try
{
    watcher.Path = @"C:\MyFolder";
} catch (ArgumentException ex) {
    Console.WriteLine(ex.Message);
    return;
}
// Set up the things to be on the lookout for.
watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
    | NotifyFilters.FileName | NotifyFilters.DirectoryName;
// Only watch text files.
watcher.Filter = "*.txt";
// Add event handlers.
watcher.Changed += new FileSystemEventHandler(OnChanged);
watcher.Created += new FileSystemEventHandler(OnChanged);
watcher.Deleted += new FileSystemEventHandler(OnChanged); watcher.Renamed += new
RenamedEventHandler(OnRenamed);
// Begin watching the directory.
watcher.EnableRaisingEvents = true;
// Wait for the user to quit the program.
Console.WriteLine(@"Press 'q' to quit app.");
while (Console.Read() != 'q') ;
```

# Следующие два обработчика событий просто сообщают о модификациях файлов

---

```
static void OnChanged(object source, FileSystemEventArgs e)
{
    // Показать, что сделано, если файл изменен, создан или удален.
    Console.WriteLine("File: {0} {1}!", e.FullPath, e.ChangeType);
}

static void OnRenamed(object source, RenamedEventArgs e)
{
    // Показать, что файл был переименован.
    Console.WriteLine("File: {0} renamed to {1}", e.OldFullPath, e.FullPath);
}
```

Чтобы протестировать эту программу, запустите ее и откройте проводник Windows.

- Попробуйте переименовать файлы, создать файл \*.txt, удалить файл \*.txt и т.д.

```
***** The Amazing File Watcher App *****
```

```
Press 'q' to quit app.
```

```
File: C:\MyFolder\New Text Document.txt Created!
```

```
File: C:\MyFolder\New Text Document.txt renamed to C:\MyFolder\Hello.txt
```

```
File: C:\MyFolder\Hello.txt Changed!
```

```
File: C:\MyFolder\Hello.txt Changed!
```

```
File: C:\MyFolder\Hello.txt Deleted!
```



Дякую за увагу!

---