Stream и работа с файлами

В процессе работы программам часто требуется читать, записывать и передавать данные: сохранять информацию в файлы, загружать конфигурации, обрабатывать результаты вычислений или временно хранить данные в памяти. При этом объём данных может быть большим, и загрузка их целиком в память не всегда возможна или оправдана. Для решения этой задачи в .NET используется абстракция **потоков данных (Stream)**.

Stream — это универсальный механизм последовательного доступа к данным, который позволяет работать с информацией по частям, независимо от того, где эти данные физически находятся. Поток скрывает источник данных и предоставляет единый интерфейс для чтения и записи.

Важно сразу сделать принципиальное уточнение:
Stream не имеет никакого отношения к многопоточности и потокам выполнения (`Thread`). Это разные понятия, которые просто одинаково переводятся на русский язык. В предыдущей лекции мы рассматривали потоки выполнения и асинхронность, а в этой лекции речь пойдёт исключительно о потоках данных.

1. Абстракция Stream и иерархия классов

`Stream` — это абстрактный базовый класс из пространства имён `System.IO`, который определяет общий интерфейс для:

- последовательного чтения данных;
- последовательной записи данных;
- управления позицией внутри потока.

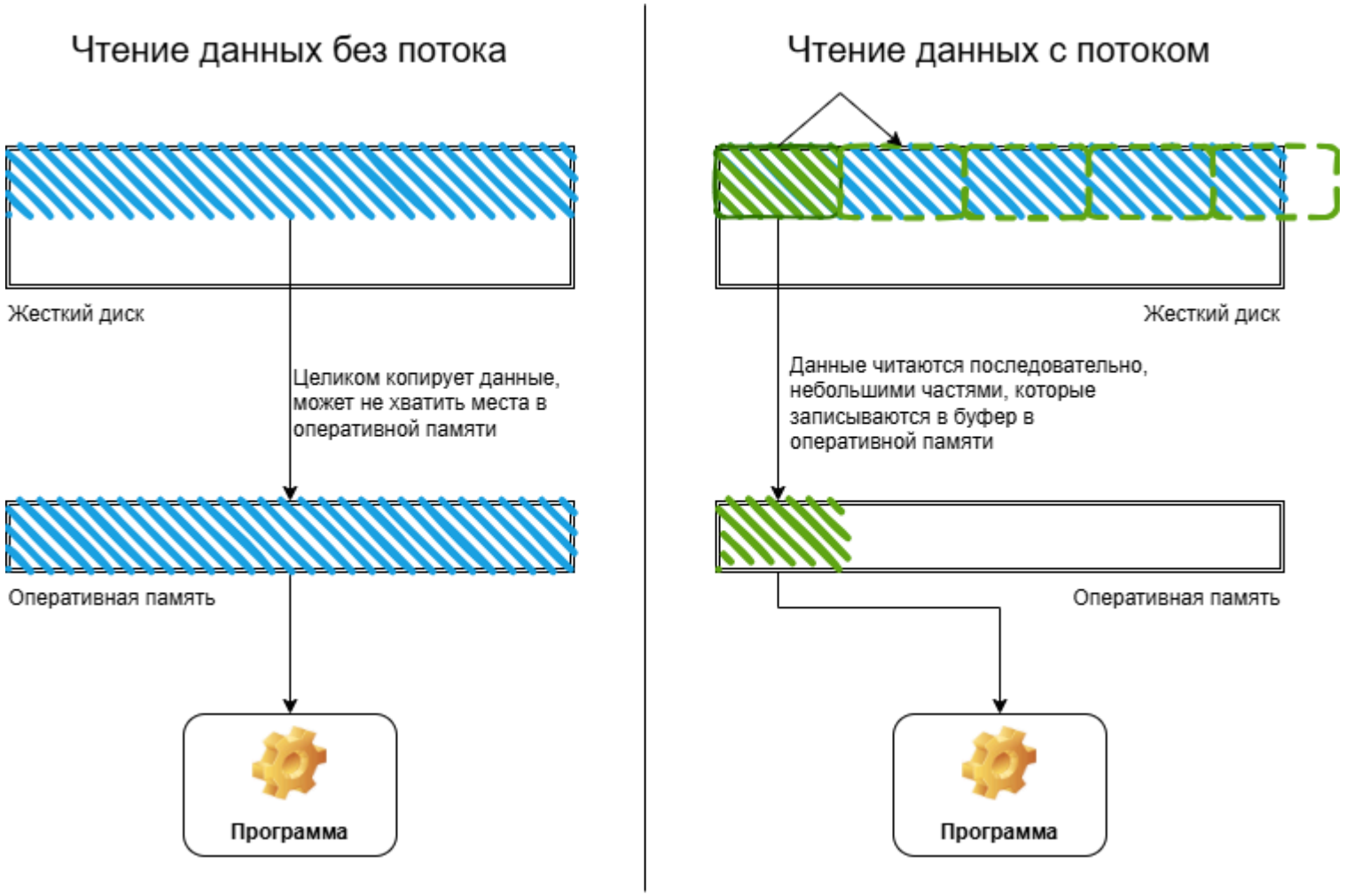
Основная идея потока заключается в том, чтобы передавать данные последовательно, без необходимости загружать их целиком в память. Они применяются при:

- Чтении и записи файлов;
- Обмене данными по сети;
- Работе с данными в оперативной памяти.

При работе с потоком мы можем контролировать:

- Направление (чтение или запись);
- Скорость (буферизация);
- Источник (файл, память, сеть).

Это делает потоки особенно полезными при работе с большими файлами и другими источниками данных.


Чтение данных без потока
Жесткий диск
Целиком копирует данные, может не хватить места в оперативной памяти
Оперативная память
Программа
Чтение данных с потоком
Жесткий диск
Данные читаются последовательно, небольшими частями, которые записываются в буфер в оперативной памяти
Оперативная память
Программа

Абстрактность Stream

`Stream` является **абстрактным классом**, поэтому:

- нельзя создать экземпляр `Stream` напрямую;
- всегда используется конкретная реализация.

Это подчёркивает роль `Stream` как **контракта**, а не готового решения. Он определяет, что можно делать с потоком, но не как именно это будет реализовано.

Одна из сильных сторон `Stream` — **абстрагирование источника данных**.

Для кода, который работает со `Stream`, не имеет значения:

- читаются ли данные из файла;
- находятся ли они в оперативной памяти;
- передаются ли они между компонентами приложения.

Важно только, что:

- данные представляют собой последовательность байтов;
- поток поддерживает операции чтения и/или записи.

Благодаря этому один и тот же код может работать с разными источниками данных без изменений.

Основные возможности класса Stream

Класс `Stream` определяет набор базовых свойств и методов, которые реализуются всеми его наследниками.

Основные свойства:

- `CanRead` — поддерживает ли поток чтение;
- `CanWrite` — поддерживает ли поток запись;
- `CanSeek` — поддерживает ли поток перемещение по данным;
- `Length` — общий размер данных (если поддерживается);
- `Position` — текущая позиция в потоке.

Основные методы:

- `Read(byte[] buffer, int offset, int count)` — чтение данных;
- `Write(byte[] buffer, int offset, int count)` — запись данных;
- `Seek(long offset, SeekOrigin origin)` — изменение позиции;
- `Flush()` — принудительная запись буферизированных данных.

Не все потоки поддерживают все операции. Например, поток может поддерживать чтение, но не поддерживать перемещение позиции.

Иерархия классов потоков

В .NET существует большое количество классов, наследующихся от `Stream`. В рамках курса нас будут интересовать прежде всего следующие группы:

Потоки-источники данных

- `FileStream` — работа с файлами на диске;
- `MemoryStream` — работа с данными в оперативной памяти.

Потоки-декораторы

Эти классы не являются самостоятельными источниками данных. Они оборачивают другой поток и добавляют функциональность:

- `BufferedStream` — добавляет буферизацию;
- `StreamReader` / `StreamWriter` — работа со строками и кодировками;
- `BinaryReader` / `BinaryWriter` — работа с бинарными данными.

Такой подход соответствует паттерну **Decorator** и позволяет комбинировать потоки.

Последовательность и позиция в потоке

Потоки предполагают **последовательный доступ** к данным:

- чтение происходит от текущей позиции вперёд;
- запись также идёт от текущей позиции.

Свойство `Position` указывает текущее смещение в потоке, а метод `Seek` позволяет изменить эту позицию (если поток это поддерживает).

Важно понимать:

- не каждый поток поддерживает `Seek`;
- код должен проверять `CanSeek`, прежде чем менять позицию.

Потоки и работа с памятью

Потоки работают с **байтами**, а не с типами данных высокого уровня.
Преобразование байтов в более удобные типы выполняется:

- вручную (через кодировки);
- либо с помощью потоков-декораторов, которые мы рассмотрим далее.

Это ещё раз подчёркивает универсальность и низкоуровневый характер `Stream`.

2. Базовые операции с потоками

Независимо от конкретной реализации, все потоки работают по одному принципу: данные читаются и записываются **последовательно**, начиная с текущей позиции.

Чтение выполняется методом `Read`:

```
int Read(byte[] buffer, int offset, int count)
```

- `buffer` — массив байтов, в который будут записаны данные;
- `offset` — позиция в `buffer`, с которой начинается запись;
- `count` — максимальное количество байтов для чтения.

Метод возвращает фактическое количество прочитанных байтов. Если возвращено `0`, поток достиг конца. Важно: `Read` **не гарантирует** чтение ровно `count` байтов, поэтому читать нужно в цикле:

```
byte[] buffer = new byte[1024];
int bytesRead;

while ((bytesRead = stream.Read(buffer, 0, buffer.Length)) > 0)
{
    // обработка прочитанных данных
}
```

Запись выполняется методом `Write` с теми же параметрами, но данные идут **из массива в поток**:

```
void Write(byte[] buffer, int offset, int count)
```

Позиция в потоке доступна через свойство `Position` и автоматически смещается после каждой операции чтения или записи. Если поток поддерживает перемещение (`CanSeek == true`), позицию можно изменить явно с помощью `Seek` :

```
stream.Seek(0, SeekOrigin.Begin); // перейти в начало
```

`SeekOrigin` определяет точку отсчёта: `Begin` — начало потока, `Current` — текущая позиция, `End` — конец.

Flush принудительно записывает все буферизированные данные в конечный источник. Особенно важен перед чтением из того же потока после записи:

```
stream.Flush();
```

3. Управление ресурсами и IDisposable

Работа с потоками данных всегда связана с использованием **внешних ресурсов**. Эти ресурсы не управляются сборщиком мусора напрямую, поэтому требуют явного и корректного освобождения. В этом разделе мы разберём, почему это важно и как правильно управлять жизненным циклом потоков в C#.

В .NET все объекты в памяти делятся на два типа:

- **Управляемые ресурсы** — объекты, за освобождение которых отвечает сборщик мусора (GC).
- **Неуправляемые ресурсы** — ресурсы операционной системы, которые GC освобождать не умеет, например открытые файлы.

Потоки (`Stream`) почти всегда работают с неуправляемыми ресурсами, поэтому:

- просто дождаться сборки мусора **недостаточно**;
- ресурс может оставаться занятым неопределённо долго.

Если поток не закрыт корректно, могут возникнуть следующие проблемы:

- файл остаётся заблокированным и недоступным для других процессов;
- данные могут быть записаны не полностью (не сброшен буфер);
- приложение расходует ограниченные системные ресурсы;
- возникают трудноуловимые ошибки при повторном доступе к файлам.

Интерфейс IDisposable

Для явного освобождения неуправляемых ресурсов в .NET используется интерфейс `IDisposable`.

Он определяет единственный метод:

```
void Dispose();
```

Классы потоков (`Stream` и все его наследники):

- реализуют `IDisposable`;
- обязаны освобождать ресурсы в методе `Dispose`.

Вызов `Dispose`:

- закрывает поток;
- освобождает системные ресурсы;
- делает дальнейшую работу с потоком недопустимой.

Использование Dispose вручную

Простейший, но небезопасный способ:

```
FileStream fs = new FileStream("data.txt", FileMode.Open);
// работа с файлом
fs.Dispose();
```

Но у такого способа есть недостаток, если между созданием и `Dispose` возникнет исключение, ресурс не будет освобождён.

По этой причине такой подход **не рекомендуется**.

Конструкция using

Правильный и рекомендуемый способ управления ресурсами — конструкция `using`.

Классический синтаксис:

```
using (FileStream fs = new FileStream("data.txt", FileMode.Open))
{
    // работа с потоком
}
```

Гарантии `using`:

- метод `Dispose` будет вызван автоматически;
- ресурс освобождается даже при возникновении исключения.

Упрощённый using (C# 8.0+)

В современных версиях C# доступен сокращённый синтаксис:

```
using var fs = new FileStream("data.txt", FileMode.Open);

// работа с потоком
```

В этом случае:

- `Dispose` вызывается автоматически при выходе из текущей области видимости;
- код становится компактнее и читаемее.

Dispose и Flush

Важно понимать связь между `Dispose` и `Flush`:

- `Flush` — сбрасывает буферизированные данные в поток;
- `Dispose` — закрывает поток и освобождает ресурсы.

Как правило:

- `Dispose` **автоматически вызывает** `Flush`;
- отдельный вызов `Flush` нужен только в долгоживущих потоках.

4. FileStream — работа с файлами

`FileStream` — основная реализация `Stream` для работы с файлами на диске. Он предоставляет низкоуровневый доступ к файловой системе и позволяет читать и записывать данные в виде последовательности байтов. Используется, когда нужно читать или записывать файлы, работать с большими файлами без загрузки их целиком в память, а также управлять правами доступа к файлу.

Создаётся `FileStream` через конструктор с несколькими параметрами:

```
using var fs = new FileStream("data.txt", FileMode.Open, FileAccess.Read, FileShare.None);
```

`FileMode` определяет, как именно файл будет открыт:

- `Create` — создать новый или перезаписать существующий;
- `CreateNew` — создать новый, ошибка если файл уже существует;
- `Open` — открыть существующий, ошибка если не найден;
- `OpenOrCreate` — открыть или создать, если отсутствует;
- `Append` — открыть и установить позицию в конец;
- `Truncate` — очистить существующий файл.

`FileAccess` указывает, какие операции разрешены: `Read`, `Write` или `ReadWrite`. При попытке выполнить недопустимую операцию будет выброшено исключение.

`FileShare` определяет, что могут делать с файлом другие процессы, пока он открыт: `None` — полный эксклюзивный доступ, `Read` — другие могут читать, `Write` — другие могут записывать, `ReadWrite` — разрешены оба действия, `Delete` — файл можно удалить. Этот параметр особенно важен в многопользовательских и серверных приложениях.

Примеры работы с FileStream

Запись строки в файл:

```
using var fs = new FileStream("data.txt", FileMode.Create, FileAccess.Write);

byte[] data = Encoding.UTF8.GetBytes("Пример записи в файл");
fs.Write(data, 0, data.Length);
```

Чтение файла:

```
using var fs = new FileStream("data.txt", FileMode.Open, FileAccess.Read);

byte[] buffer = new byte[fs.Length];
int bytesRead = fs.Read(buffer, 0, buffer.Length);

string text = Encoding.UTF8.GetString(buffer, 0, bytesRead);
Console.WriteLine(text);
```

Для больших файлов не стоит читать данные целиком — лучше читать частями в цикле, как показано в примере выше. Свойство `fs.Length` доступно только для потоков, поддерживающих `Seek`.

Запись и чтение в одном потоке — при совмещении важно явно управлять позицией:

```
using var fs = new FileStream("data.txt", FileMode.Create, FileAccess.ReadWrite);

// Запись
byte[] writeBuffer = Encoding.UTF8.GetBytes("Пример текста");
fs.Write(writeBuffer, 0, writeBuffer.Length);
fs.Flush();

// Сброс позиции перед чтением
fs.Seek(0, SeekOrigin.Begin);

// Чтение
byte[] readBuffer = new byte[fs.Length];
int readBytes = fs.Read(readBuffer, 0, readBuffer.Length);
string text = Encoding.UTF8.GetString(readBuffer, 0, readBytes);
Console.WriteLine(text);
```

Производительность и типичные ошибки

`FileStream` поддерживает внутреннюю буферизацию. Размер буфера можно задать явно через конструктор — по умолчанию он составляет 4096 байт:

```
using var fs = new FileStream(
    "data.txt",
    FileMode.Open,
    FileAccess.Read,
    FileShare.Read,
    bufferSize: 4096
);
```

Типичные ошибки при работе с `FileStream`: отсутствие `using` (файл остаётся открытым), неправильный выбор `FileMode` (можно случайно перезаписать файл), забытый `Seek` перед чтением после записи, а также загрузка больших файлов целиком в память вместо посекционного чтения.

5. Буферизация и производительность

При работе с потоками данных производительность приложения во многом зависит от того, **как именно выполняются операции чтения и записи**. Одним из ключевых факторов здесь является буферизация. В этом разделе мы разберём, что такое буфер, зачем он нужен и как его использование влияет на скорость работы с файлами.

Буферизация — это приём, при котором данные временно накапливаются в памяти и обрабатываются крупными блоками, а не по одному байту.

Без буферизации:

- каждое чтение или запись может приводить к обращению к файловой системе;
- такие операции являются дорогими с точки зрения времени выполнения.

С буферизацией:

- данные читаются и записываются пакетами;
- количество системных вызовов существенно уменьшается;
- производительность возрастает.

Буферизация в FileStream

`FileStream` по умолчанию использует **внутренний буфер**. Однако:

- его размер можно контролировать;
- неправильный размер буфера может негативно повлиять на производительность.

Размер буфера задаётся в конструкторе:

```
using var fs = new FileStream(
    "data.txt",
    FileMode.Open,
    FileAccess.Read,
    FileShare.Read,
    bufferSize: 4096
);
```

Типичные значения буфера:

- 4 КБ — стандартный размер страницы памяти;
- 8–16 КБ — часто оптимальны для файлов;
- слишком маленький буфер — плохо для производительности;
- слишком большой — может быть избыточным по памяти.

Пример: чтение без эффективной буферизации

Пример неэффективного чтения:

```
using var fs = new FileStream("data.txt", FileMode.Open);

int value;
while ((value = fs.ReadByte()) != -1)
{
    // обработка одного байта
}
```

Недостатки:

- каждый вызов `ReadByte` может обращаться к диску;
- код работает медленно даже для небольших файлов.

Пример: чтение с буфером

Эффективный вариант:

```
using var fs = new FileStream("data.txt", FileMode.Open);

byte[] buffer = new byte[4096];
int bytesRead;

while ((bytesRead = fs.Read(buffer, 0, buffer.Length)) > 0)
{
    // обработка блока данных
}
```

Преимущества:

- данные читаются крупными блоками;
- значительно меньше системных вызовов;
- заметный прирост производительности.

Сравнение производительности

В первом случае размер буфера будет равен 1, то есть для чтения каждого байта в файле, будет происходить физическое чтение с диска. Во втором случае, мы будем считывать данные блоками размером 4096 байт за раз.

```
var sw = new Stopwatch();
sw.Start();
using var fs1 = new FileStream("file.txt", FileMode.Open, FileAccess.ReadWrite, bufferSize: 1);
byte[] singleByte = new byte[1];
for (int i = 0; i < fs1.Length; i++)
    fs1.Read(singleByte, 0, 1);
sw.Stop();
Console.WriteLine($"Размер буфера 1: {sw.ElapsedMilliseconds}мс");

sw.Restart();
using var fs2 = new FileStream("file.txt", FileMode.Open, FileAccess.ReadWrite, bufferSize: 4096);
byte[] buffer = new byte[4096];
for (int i = 0; i < fs2.Length; i+=4096)
    fs2.Read(buffer, 0, buffer.Length);
sw.Stop();
Console.WriteLine($"Размер буфера 4096: {sw.ElapsedMilliseconds}мс");
```

С использованием большого буфера, файл был прочитан в 4 тысячи раз быстрее.

```
Размер буфера 1: 4195мс
Размер буфера 4096: 1мс
```

Буферизация и запись данных

Буферизация важна не только для чтения, но и для записи:

- данные сначала записываются в буфер;
- фактическая запись на диск происходит позже;
- `Flush` принудительно сбрасывает буфер.

Это позволяет:

- уменьшить количество операций записи;
- повысить производительность.

Баланс между памятью и скоростью

При выборе размера буфера необходимо учитывать:

- объём обрабатываемых данных;
- доступную память;
- частоту операций ввода-вывода.

Практические рекомендации:

- не использовать слишком маленькие буферы, избегать чтения по одному байту;
- начинать с 4–8 КБ;
- измерять производительность при необходимости;

6. Декораторы потоков

Работа напрямую с байтовыми массивами неудобна и подвержена ошибкам, особенно когда требуется обрабатывать строки, числа или другие типы данных. Для решения этой проблемы в .NET используются **декораторы потоков** — классы, которые оборачивают существующий поток и добавляют дополнительную функциональность, не изменяя сам источник данных.

Декоратор потока:

- сам является потоком или работает поверх потока;
- принимает в конструкторе другой `Stream`;
- добавляет новый уровень абстракции.

Это соответствует шаблону проектирования **Decorator**:

- функциональность расширяется без наследования;
- поведение можно комбинировать.

Схематично:

```
FileStream → BufferedStream → StreamReader
```

При этом:

- `FileStream` отвечает за работу с файлом;
- `BufferedStream` — за буферизацию;
- `StreamReader` — за работу со строками и кодировками.

StreamReader и StreamWriter

Классы `StreamReader` и `StreamWriter` предназначены для работы с **текстовыми данными**.

Основные задачи:

- преобразование байтов в строки и обратно;
- управление кодировкой;
- построчное чтение и запись.

Кодировки и StreamReader

`StreamReader` работает с конкретной кодировкой (`Encoding`).

Важно:

- кодировка должна совпадать при чтении и записи;
- неверная кодировка приводит к искажению текста.

По умолчанию:

- используется UTF-8;
- но лучше указывать кодировку явно для надёжности.

Пример: чтение текста

```
using var fs = new FileStream("data.txt", FileMode.Open);
using var reader = new StreamReader(fs, Encoding.UTF8);

string line;
while ((line = reader.ReadLine()) != null)
{
    Console.WriteLine(line);
}
```

Пример: запись текста

```
using var fs = new FileStream("data.txt", FileMode.Create);
using var writer = new StreamWriter(fs, Encoding.UTF8);

writer.WriteLine("Первая строка");
writer.WriteLine("Вторая строка");
```

Преимущества:

- удобство работы со строками;
- автоматическая буферизация;
- корректная обработка кодировок.

BinaryReader и BinaryWriter

`BinaryReader` и `BinaryWriter` используются для работы с **бинарными данными**, а не текстом.

Они позволяют:

- читать и записывать примитивные типы (`int` , `double` , `bool` , `string`);
- сохранять данные в компактном бинарном формате;
- избегать ручной сериализации байтов.

Пример: запись бинарных данных

```
using var fs = new FileStream("data.bin", FileMode.Create);
using var writer = new BinaryWriter(fs);

writer.Write(42);
writer.Write(3.14);
writer.Write("Пример");
```

Пример: чтение бинарных данных

```
using var fs = new FileStream("data.bin", FileMode.Open);
using var reader = new BinaryReader(fs);

int number = reader.ReadInt32();
double value = reader.ReadDouble();
string text = reader.ReadString();
```

BufferedStream как декоратор

`BufferedStream` — это декоратор, который добавляет буферизацию поверх любого потока.

Пример:

```
using var fs = new FileStream("data.txt", FileMode.Open);
using var bs = new BufferedStream(fs, 8192);
// работа с bs
```

Используется, когда:

- необходимо явно управлять размером буфера;
- базовый поток не предоставляет нужной буферизации;
- требуется повысить производительность без изменения логики кода.

Комбинирование декораторов

Декораторы можно комбинировать в цепочку:

```
using var fs = new FileStream("data.txt", FileMode.Open);
using var bs = new BufferedStream(fs);
using var reader = new StreamReader(bs);

// чтение данных
```

Рекомендации:

- внешний декоратор закрывается первым;
- базовый поток закрывается последним;
- всегда использовать `using`.

7. MemoryStream — работа с данными в памяти

Не все задачи работы с потоками связаны с файлами на диске. Во многих случаях данные существуют только во время выполнения программы и не требуют постоянного хранения. Для таких сценариев в .NET используется класс `MemoryStream`, который представляет поток данных, полностью размещённый в оперативной памяти.

`MemoryStream` — это реализация `Stream`, которая:

- хранит данные в массиве байтов в памяти;
- не использует файловую систему;
- обеспечивает быстрый доступ к данным.

`MemoryStream` часто применяется:

- для временного хранения данных;
- как промежуточный буфер между компонентами;
- для обработки данных перед сохранением;
- в тестировании, когда работа с файлами нежелательна.

Ключевые различия:

MemoryStream	**FileStream**
Работает в памяти	Работает с диском
Очень быстрый	Ограничен скоростью I/O
Не блокирует файлы	Может блокировать файлы
Не зависит от ОС	Зависит от файловой системы

Создание MemoryStream

Простейшее создание пустого потока:

```
using var ms = new MemoryStream();
```

Также можно создать поток поверх существующего массива байтов:

```
byte[] data = Encoding.UTF8.GetBytes("Пример");
using var ms = new MemoryStream(data);
```

Во втором случае:

- поток инициализируется готовыми данными;
- позиция устанавливается в начало.

Запись данных в MemoryStream

Пример записи:

```
using var ms = new MemoryStream();

byte[] buffer = Encoding.UTF8.GetBytes("Текст в памяти");
ms.Write(buffer, 0, buffer.Length);
```

Поведение аналогично `FileStream`:

- запись идёт последовательно;
- позиция смещается автоматически.

Чтение данных из MemoryStream

Для чтения необходимо сбросить позицию:

```
ms.Seek(0, SeekOrigin.Begin);

byte[] readBuffer = new byte[ms.Length];
int bytesRead = ms.Read(readBuffer, 0, readBuffer.Length);

string text = Encoding.UTF8.GetString(readBuffer, 0, bytesRead);
Console.WriteLine(text);
```

Важно:

- `MemoryStream` всегда поддерживает `Seek`;
- доступ к `Length` возможен без ограничений.

Получение данных из MemoryStream

После записи данные можно получить в виде массива байтов:

```
byte[] result = ms.ToArray();
```

Следует учитывать:

- `ToArray` создаёт **копию** данных;
- для больших объёмов данных это может быть дорого по памяти.

Использование декораторов с MemoryStream

`MemoryStream` можно комбинировать с любыми декораторами потоков:

```
using var ms = new MemoryStream();
using var writer = new StreamWriter(ms);

writer.WriteLine("Строка 1");
writer.WriteLine("Строка 2");
writer.Flush();

ms.Seek(0, SeekOrigin.Begin);

using var reader = new StreamReader(ms);
string content = reader.ReadToEnd();
```

Это позволяет:

- работать со строками и бинарными данными;
- не создавая файлов.

8. Асинхронная работа с потоками

При работе с файлами операции чтения и записи относятся к задачам ввода-вывода и могут занимать значительное время. Если выполнять такие операции синхронно, поток выполнения будет блокироваться до их завершения. Для повышения отзывчивости приложений и более эффективного использования ресурсов в .NET предусмотрена **асинхронная работа с потоками**.

Операции работы с файлами относятся к классу **I/O-bound задач**:

- процессор большую часть времени ожидает завершения операции;
- вычисления практически не выполняются.

Асинхронность позволяет:

- не блокировать поток выполнения;
- освободить поток для выполнения других задач;
- повысить масштабируемость приложения.

Важно помнить:

- асинхронность **не означает** создание нового потока;
- `async/await` — это механизм неблокирующего ожидания.

Асинхронные методы Stream

Класс `Stream` и его наследники предоставляют асинхронные версии основных операций:

- `ReadAsync`
- `WriteAsync`
- `FlushAsync`

Сигнатура `ReadAsync`:

```
Task<int> ReadAsync(
    byte[] buffer,
    int offset,
    int count
);
```

Возвращается `Task<int>`, который:

- завершается после окончания операции;
- содержит количество прочитанных байтов.

Пример: асинхронная запись в файл

```
using var fs = new FileStream(
    "data.txt",
    FileMode.Create,
    FileAccess.Write,
    FileShare.None
);

byte[] data = Encoding.UTF8.GetBytes("Асинхронная запись");

await fs.WriteAsync(data, 0, data.Length);
await fs.FlushAsync();
```

Пока выполняется `WriteAsync`:

- поток выполнения не блокируется;
- управление может быть передано другим задачам.

Пример: асинхронное чтение из файла

```
using var fs = new FileStream(
    "data.txt",
    FileMode.Open,
    FileAccess.Read,
    FileShare.Read
);

byte[] buffer = new byte[4096];
int bytesRead;

while ((bytesRead = await fs.ReadAsync(buffer, 0, buffer.Length)) > 0)
{
    // обработка данных
}
```

Принцип чтения:

- полностью аналогичен синхронному варианту;
- используется цикл до конца потока.

Асинхронные декораторы потоков

Декораторы потоков также поддерживают асинхронные методы:

- `StreamReader.ReadLineAsync`
- `StreamWriter.WriteLineAsync`

Пример:

```
using var fs = new FileStream("data.txt", FileMode.Open);
using var reader = new StreamReader(fs);

string line;
while ((line = await reader.ReadLineAsync()) != null)
{
    Console.WriteLine(line);
}
```

Асинхронность сохраняется на всех уровнях цепочки потоков.

9. Обзор специализированных потоков

В этой лекции мы рассмотрели: `FileStream`, `MemoryStream`, `BufferedStream`, `BinaryReader` / `BinaryWriter` и `StreamReader` / `StreamWriter`, но кроме них в .NET существует ряд специализированных потоков для конкретных сценариев. Все они наследуют `Stream` и работают по тем же принципам чтения и записи, но отличаются источником данных и дополнительными возможностями.

Здесь кратко перечислены некоторые из них, которые могу вам понадобиться в дальнейшем:

`NetworkStream` — поток для работы с данными, передаваемыми по сети. Не поддерживает `Seek`, так как данные поступают последовательно по мере их получения. Активно используется совместно с асинхронными методами.

`GZipStream` / **`DeflateStream`** / **`BrotliStream`** — потоки-декораторы для сжатия и распаковки данных на лету. Оборачивают другой поток и прозрачно преобразуют данные при чтении или записи. Например, `GZipStream` поверх `FileStream` позволяет читать и писать сжатые файлы без промежуточной буферизации.

`CryptoStream` — поток-декоратор для шифрования и дешифрования данных. Аналогично потокам сжатия, оборачивает любой другой поток и преобразует данные на лету с использованием заданного алгоритма.

При работе со специализированными потоками важно учитывать, что не все из них поддерживают `Seek` и `Length`, поэтому перед использованием стоит проверять свойства `CanRead`, `CanWrite` и `CanSeek`.

Практическое задание

Исходная ситуация

В проекте уже существует статический класс `FileManager`, который:

- напрямую использует `File.ReadAllLines` / `File.WriteAllLines`
- напрямую работает с файловой системой
- содержит в себе:
  - логику сериализации
  - логику хранения
  - логику доступа к данным приложения (`AppInfo`)
- не поддерживает шифрование
- плохо тестируется

Что нужно сделать

Полностью переписать `FileManager` с использованием `Stream`

Требования к реализации

1. Запрещено использовать

   ```
   File.ReadAllLines
   File.WriteAllLines
   File.ReadAllText
   File.WriteAllText
   ```

   Работа с файлами должна вестись **только через потоки**.

2. Используемые классы и типы

   В реализации **обязательно** должны использоваться:

   - `FileStream`
   - `StreamReader`
   - `StreamWriter`
   - `BufferedStream`
   - `CryptoStream`

Шифрование данных

Для шифрования данных используйте класс CryptoStream

1. Все данные профилей и задач должны **храниться в зашифрованном виде**.
2. При чтении данные должны автоматически расшифровываться.
3. Алгоритм шифрования `AES`
4. Ключ и IV:
   - хранятся централизованно
   - не генерируются при каждом сохранении

Пример цепочки потоков (декораторы)

При **сохранении**:

```
FileStream
 → BufferedStream
   → CryptoStream (Encrypt)
     → StreamWriter
```

При **загрузке**:

```
FileStream
 → BufferedStream
   → CryptoStream (Decrypt)
     → StreamReader
```

Архитектурные изменения

1. Убрать статичность FileManager
2. Отделить FileManager от AppInfo

   Запрещено внутри `FileManager`:

   ```
   AppInfo.Profiles
   AppInfo.CurrentProfileId
   ```

   Вместо этого все данные передаются **через конструктор**
3. Ввести абстракцию для тестирования

   Создать интерфейс, например:

   ```
   public interface IDataStorage
   {
       void SaveProfiles(IEnumerable<Profile> profiles);
       IEnumerable<Profile> LoadProfiles();

       void SaveTodos(Guid userId, IEnumerable<TodoItem> todos);
       IEnumerable<TodoItem> LoadTodos(Guid userId);
   }
   ```

`FileManager` должен **реализовывать этот интерфейс**.

Это позволит:

- заменить файловое хранилище на БД
- писать unit-тесты
- использовать mock-реализации для тестов

Обработка ошибок

1. Корректно закрывать потоки (`using`)
2. Обрабатывать:
   - ошибки доступа к файлам
   - ошибки расшифровки
   - повреждённые данные
3. Выбрасывать **осмысленные исключения**, а не просто `Exception`