# Лекция №7. Потоки

## Stream

**Поток (Stream)** — это абстракция для последовательной передачи данных между источником и приемником. Его не нужно путать с потоками (Thread), который мы рассматривали в лекции про многопоточность, это разные понятия, которые просто одинаково переводятся на русский.

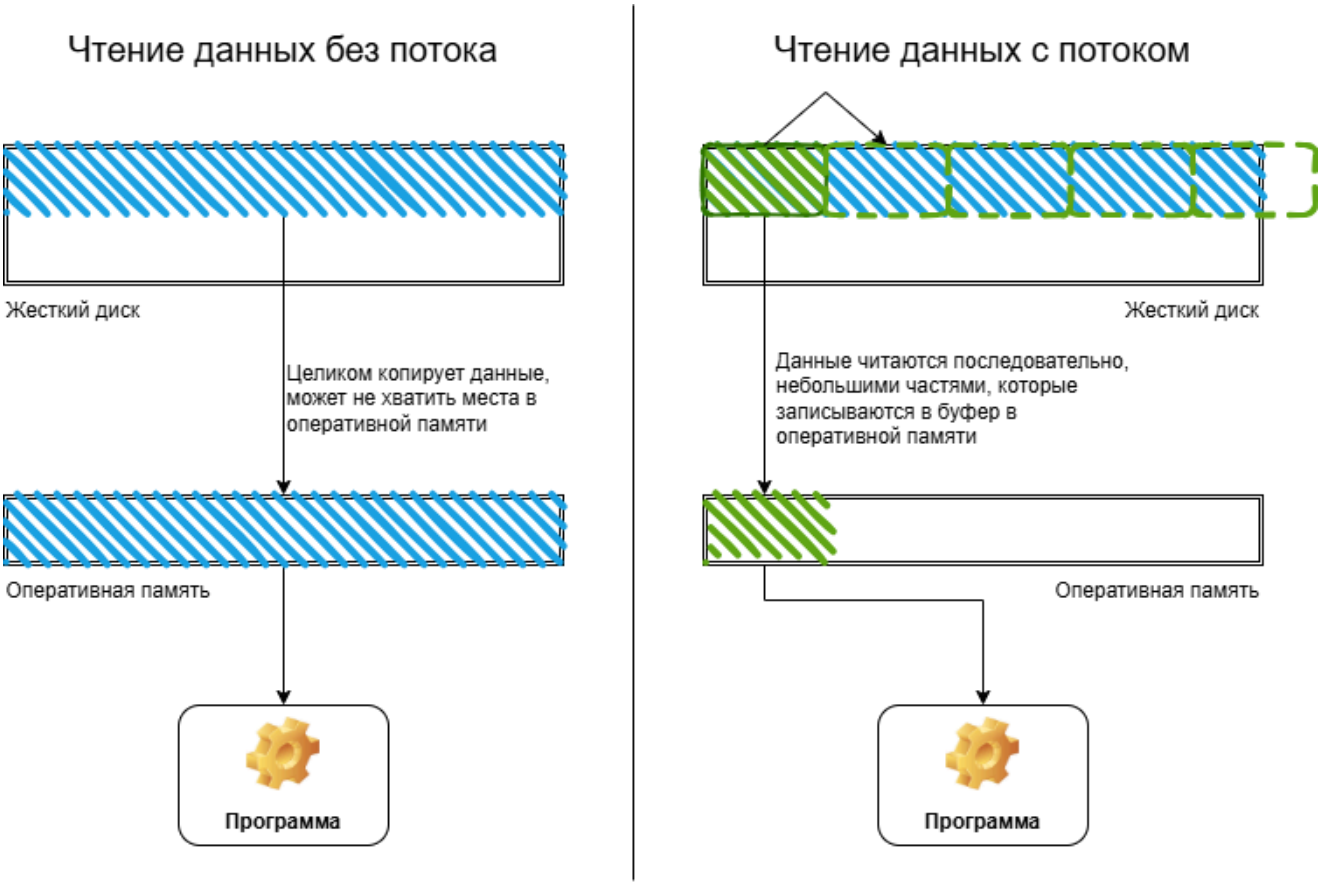
Основная идея потока заключается в том, чтобы передавать данные последовательно, без необходимости загружать их целиком в память. Они применяются при:

- 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).

Все специализированные классы для работы с потоками наследуются от абстрактного класса `Stream` из пространства имен `System.IO`.




## Базовые операции с потоками

В классе `Stream` есть несколько базовых методов:

- `Read(byte[] buffer, int offset, int count)` — читает данные из потока в массив `buffer` (начиная с позиции `offset`, не более `count` байт) и возвращает количество фактически прочитанных байт.
- `Write(byte[] buffer, int offset, int count)` — работает аналогично `Read`, но он записывает данные в поток.
- `Seek(long offset, SeekOrigin origin)` — изменяет позицию в потоке (например, для пропуска части данных или перезаписи).
- `Flush()` — принудительно записывает буферизированные данные в конечный источник.

Также класс `Stream`, а значит и все остальные потоки, реализуют интерфейс `IDisposable`, так как они работают с неуправляемыми ресурсами (подключения к файлам, базам данных, сетевые подключения и т.д.), которые не могут быть автоматически удалены сборщиком мусора. Поэтому после завершения использования потоков, мы должны вызывать метод `Dispose()`, либо использовать `using`.

Рассмотрим использование потоков для работы с файлами на примере класса `FileStream`:

```
// Открытие файла для записи (существующий файл будет перезаписан).
using (FileStream fs = new FileStream("data.txt", FileMode.Create,
FileAccess.ReadWrite, FileShare.ReadWrite))
{
    // Запись в файл
    byte[] writeBuffer = Encoding.UTF8.GetBytes("Пример записи в файл.");
    fs.Write(writeBuffer, 0, writeBuffer.Length); //Записываем текст в файл
    fs.Flush(); // Отчищаем буфер

    // Сбрасываем позицию в начало файла перед чтением!
    fs.Seek(0, SeekOrigin.Begin);

            // Чтение из файла
    byte[] readBuffer = new byte[fs.Length]; //fs.Length - размер файла
    int readBytesCount = fs.Read(readBuffer, 0, readBuffer.Length); //читаем
весь текст из файла


    Console.WriteLine($"Прочитано: {readBytesCount} байтов");
    Console.WriteLine($"Содержимое файла:
{Encoding.UTF8.GetString(readBuffer)}");
} // Автоматический вызов fs.Dispose(), неуправляемые ресурсы удаляются.
```

В результате получим в консоли

```
Прочитано: 38 байтов
Содержимое файла: Пример записи в файл.
```

При создании `FileStream` можно указать:

- `FileMode` — указывает, каким образом нужно открыть файл (обязательный);
- `FileAccess` — указывает, что можно делать с файлом (читать или записывать);
- `FileShare` — указывает, какие операции доступны для других процессов, когда файл открыт.

# Буфер

**Буфер потока** — это временная область памяти (обычно массив байтов), которая используется для оптимизации операций ввода-вывода (I/O). Он действует как промежуточный слой между программой и источником данных (файлом, сетью, устройством), уменьшая количество обращений к физическому устройству.

У многих потоков при создании можно указать параметр `bufferSize`, который отвечает за размер буфера, по умолчанию он равняется 4096 байтам. Давайте рассмотрим пример для того, чтобы понять как использование буфера влияет на производительность. 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 (FileStream fs = new FileStream("file.txt", FileMode.Open,
FileAccess.ReadWrite, FileShare.ReadWrite, bufferSize: 1))
{
    byte[] singleByte = new byte[1];
    for (int i = 0; i < fs.Length; i++)
    {
        fs.Read(singleByte, 0, 1);
    }
}
sw.Stop();
Console.WriteLine($"Размер буфера 1: {sw.ElapsedMilliseconds}мс");
sw.Restart();
using (FileStream fs = new FileStream("file.txt", FileMode.Open,
FileAccess.ReadWrite, FileShare.ReadWrite, bufferSize: 4096))
{
    byte[] buffer = new byte[4096];
    for (int i = 0; i < fs.Length; i+=4096)
    {
        fs.Read(buffer, 0, buffer.Length);
    }
}
sw.Stop();
Console.WriteLine($"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
```

## Декораторы потоков

Как вы могли заметить `FileStream`, работает с массивами байт, поэтому для того, чтобы записать и прочитать строку нам приходилось каждый раз использовать `Encoding.UTF8.GetBytes` и `Encoding.UTF8.GetString`. Это не очень неудобно, поэтому в C# есть специальные классы, которые упрощают этот процесс:

- `StreamReader` читает байты из потока и декодирует их в строки;
- `StreamWriter` кодирует строки в байты и записывает в поток.

Давайте перепишем прошлый пример с помощью этих классов:

```
// В новых версиях C# можно не писать фигурные скобки после using
// Компилятор сам определит, когда нужно вызывать Dispose()
using var fs = new FileStream("data.txt", FileMode.Create,
FileAccess.ReadWrite, FileShare.ReadWrite);

// Запись
using var sw = new StreamWriter(fs, Encoding.UTF8);
sw.WriteLine("Пример записи в файл.");
sw.Flush(); // сбрасываем буфер в поток

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

// Чтение
using var sr = new StreamReader(fs, Encoding.UTF8, leaveOpen: true);
string readString = sr.ReadLine();

Console.WriteLine($"Содержимое файла: {readString}");
```

Для чтения/записи примитивных типов данных ( `int` , `double` , `bool` и т.д.) в бинарном формате, нужно использовать `BinaryReader` и `BinaryWriter` . Рассмотрим пример записи

```
using (FileStream fs = new FileStream("numbers.bin", FileMode.Create))
using (BinaryWriter writer = new BinaryWriter(fs))
{
    writer.Write(42);          // int
    writer.Write(3.14);        // double
    writer.Write(true);        // bool
}
```

и чтения

```
using (FileStream fs = new FileStream("numbers.bin", FileMode.Open))
using (BinaryReader reader = new BinaryReader(fs))
{
    int number = reader.ReadInt32();
    double pi = reader.ReadDouble();
    bool flag = reader.ReadBoolean();
}
```

Для более гибкой буферизации используется `BufferedStream` . Пример использования:

```
using (FileStream fs = new FileStream("data.bin", FileMode.Open))
using (BufferedStream buffered = new BufferedStream(fs, 4096)) // Буфер 4 КБ
{
```

```
    byte[] buffer = new byte[1024];
    int bytesRead = buffered.Read(buffer, 0, buffer.Length);
}
```

Ключевой особенностью декораторов является то, что их можно комбинировать. Например:

```
// FileStream → BufferedStream → StreamReader

using (var fs = new FileStream("file.txt", FileMode.Open))
using (var buffered = new BufferedStream(fs))
using (var reader = new StreamReader(buffered))
{
    // Чтение с буферизацией
}
```

## Асинхронная работа с потоками

Кроме обычных синхронных методов у потоков есть и их асинхронные версии.

- `ReadAsync(byte[] buffer, int offset, int count)`
- `WriteAsync(byte[] buffer, int offset, int count)`
- `FlushAsync()`

Пример записи и чтения из файла асинхронно:

```
// В новых версиях C# можно не писать фигурные скобки после using
// Компилятор сам определит, когда нужно вызывать Dispose()
using var fs = new FileStream("data.txt", FileMode.Create,
FileAccess.ReadWrite, FileShare.ReadWrite);

// Запись
using var sw = new StreamWriter(fs, Encoding.UTF8);
await sw.WriteLineAsync("Пример записи в файл.");
await sw.FlushAsync(); // сбрасываем буфер в поток

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

// Чтение
using var sr = new StreamReader(fs, Encoding.UTF8, leaveOpen: true);
string readString = await sr.ReadLineAsync();

Console.WriteLine($"Содержимое файла: {readString}");
```

Всегда используйте `await`, т.к. он не блокирует основной поток.

# Специализированные потоки

`MemoryStream` — работа с данными в памяти, он используется для:

- Хранение данных в оперативной памяти вместо файлов.
- Промежуточный буфер для обработки данных (например, сериализация).
- Тестирование без реального I/O.

Пример использования:

```
using (MemoryStream memoryStream = new MemoryStream())
{
    // Запись данных в поток
    byte[] data = Encoding.UTF8.GetBytes("Hello, MemoryStream!");
    memoryStream.Write(data, 0, data.Length);

    // Чтение данных
    memoryStream.Seek(0, SeekOrigin.Begin); // Сброс позиции
    byte[] buffer = new byte[memoryStream.Length];
    memoryStream.Read(buffer, 0, buffer.Length);
    string content = Encoding.UTF8.GetString(buffer);
    Console.WriteLine(content); // Hello, MemoryStream!
}
```

`NetworkStream` — сетевые взаимодействия. Он оборачивает сокет ( `TcpClient` / `TcpListener` ) для передачи данных по сети. Мы рассмотрим его более подробно в лекции посвященной сетевому взаимодействию.

`GZipStream` — сжатие данных. Используется для сжатия (GZip) и распаковка данных "на лету". Часто его комбинируют с `FileStream` или `MemoryStream`. Рассмотрим пример сжатия.

```
using (FileStream source = new FileStream("data.txt", FileMode.Open))
using (FileStream compressed = new FileStream("data.gz", FileMode.Create))
using (GZipStream gzip = new GZipStream(compressed,
CompressionMode.Compress))
{
    source.CopyTo(gzip); // Сжатие и запись в файл
}
```

и распаковки данных.

```
using (FileStream compressed = new FileStream("data.gz", FileMode.Open))
using (GZipStream gzip = new GZipStream(compressed,
CompressionMode.Decompress))
using (MemoryStream decompressed = new MemoryStream())
```

```
{
    gzip.CopyTo(decompressed); // Распаковка в MemoryStream
    string content = Encoding.UTF8.GetString(decompressed.ToArray());
}
```