# ЛЕКЦИЯ 4

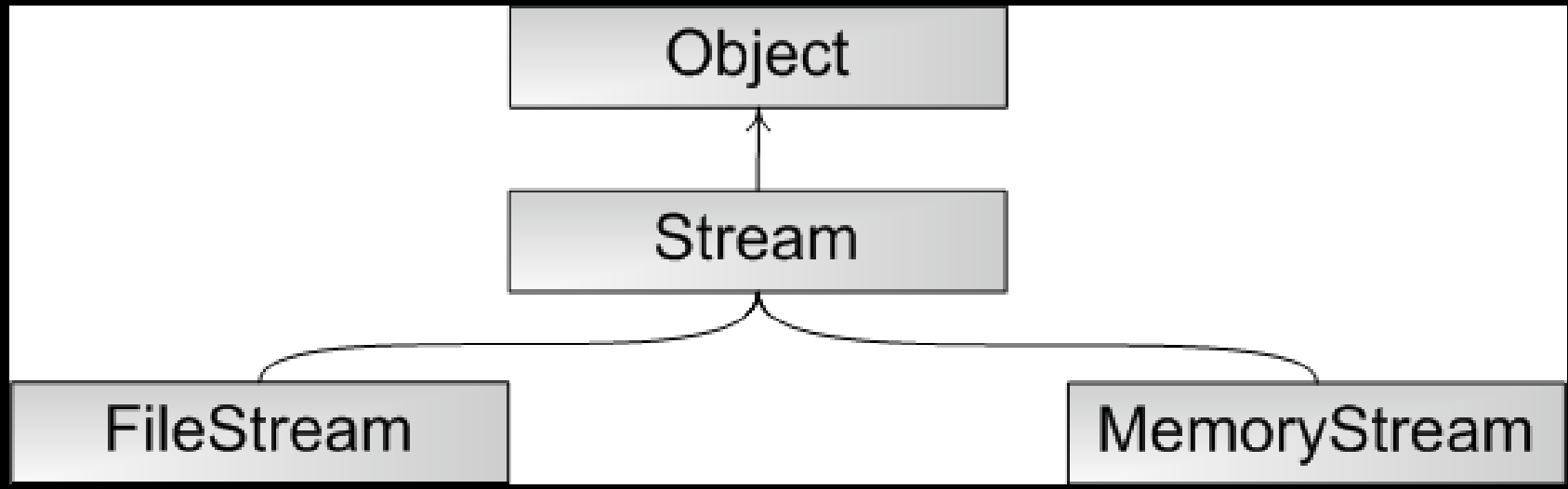
- Модуль 3
- 17.01.2024
- Работа с файлами и потоками данных

# ЦЕЛИ ЛЕКЦИИ

- Познакомиться с концепцией потока ввода и вывода данных
- Изучить общие принципы иерархии классов-потоков в C#
- Разобраться с файловыми потоками

# ПОТОКИ ДАННЫХ

- **Поток данных** – абстракция, обозначающая источник получения или средство приема данных, позволяющая унифицировать процессы обмена данными между программой и внешним окружением




```
public abstract class Stream : MarshalByRefObject, IAsyncDisposable, IDisposable
```

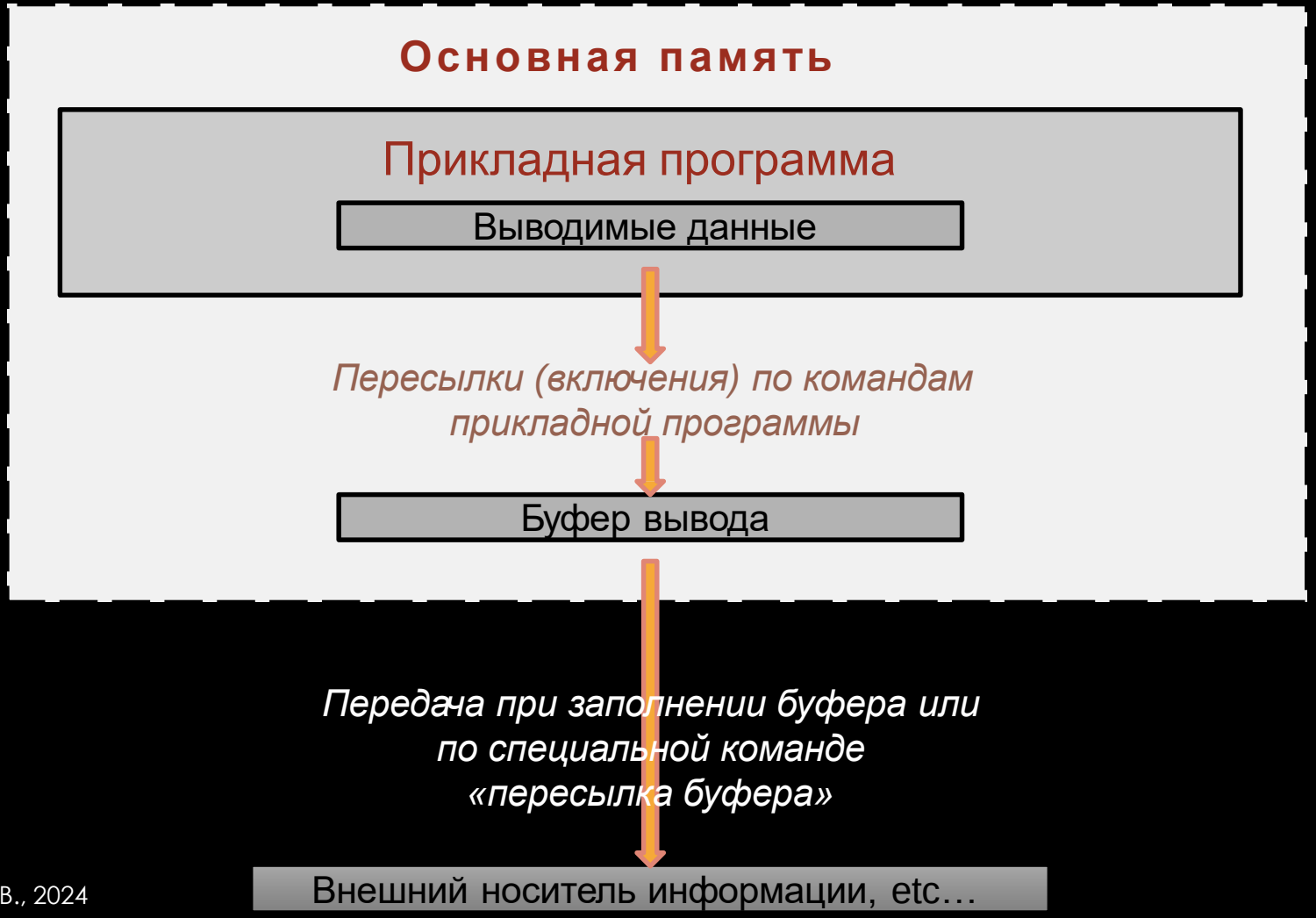
# МЕТОДЫ АБСТРАКТНОГО КЛАССА STREAM

| Имя | Назначение |
|---|---|
| **Close()** | Закрывает поток и освобождает ассоциированные с ним ресурсы. |
| **Flush()** | Освобождает буфер обмена, предварительно передав из него данные в информационный источник. |
| **Read()**<br>**ReadByte()** | Читает байт или последовательность байтов из потока и перемещает текущую позицию на прочитанное число байтов |
| **Seek()** | Устанавливает текущую позицию в потоке |
| **SetLength()** | Устанавливает длину потока |
| **Write()**<br>**WriteByte()** | Записывает в поток байт или последовательность байтов, перемещает указатель на количество записанных байтов |

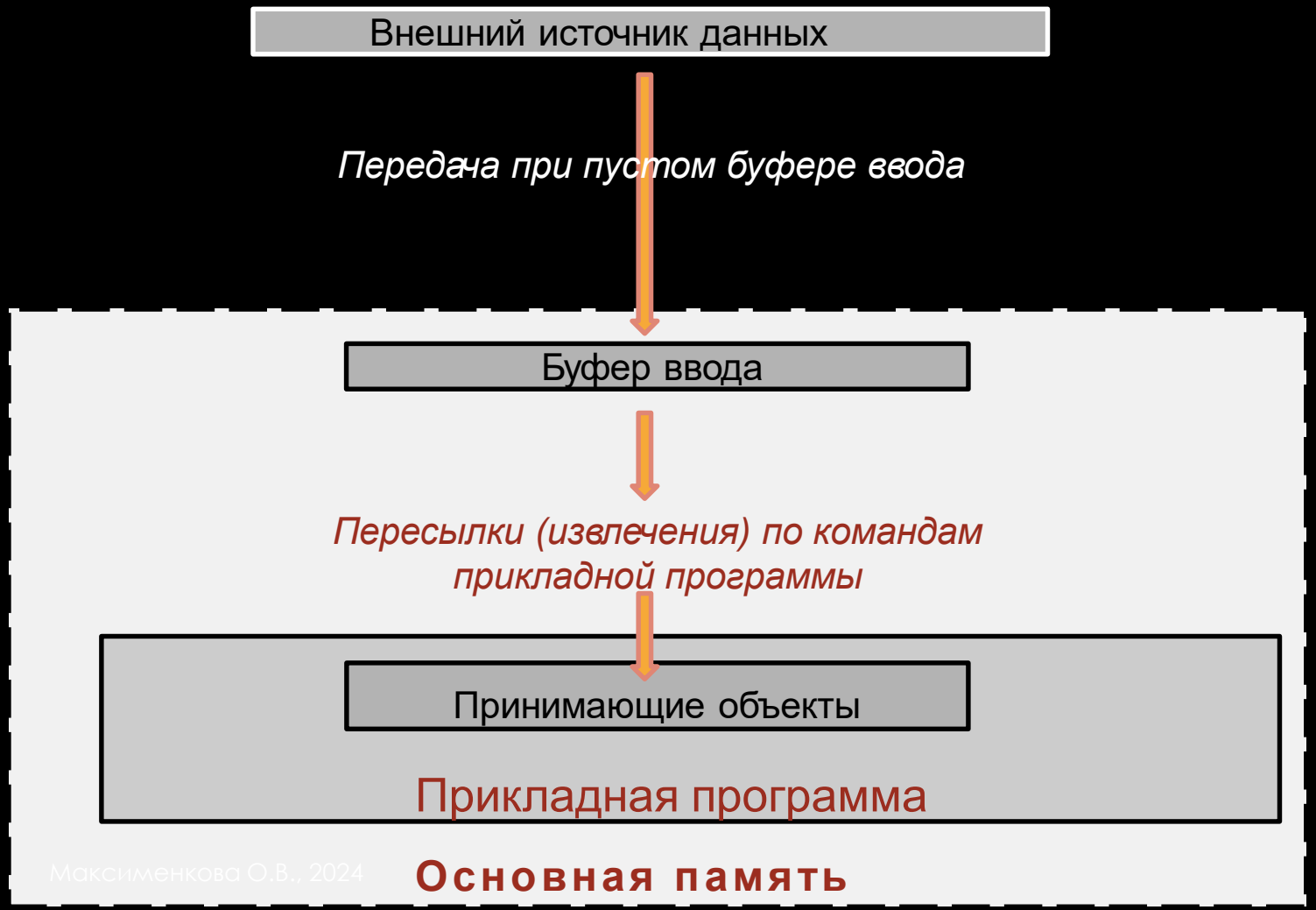
# СВОЙСТВА АБСТРАКТНОГО КЛАССА STREAM

| Имя | Назначение |
|---|---|
| **CanRead**<br>**CanSeek**<br>**CanWrite** | Проверяют для потока возможность чтения, записи и возможность изменения текущей позиции |
| **Length** | Длина потока в байтах |
| **Position** | Текущая позиция в байтах |

# ВЫХОДНОЙ ПОТОК ДАННЫХ



# ВХОДНОЙ ПОТОК ДАННЫХ



# КЛАСС STREAM

Абстрактный класс, обеспечивающий чтение и запись байтов

- Все классы, являющиеся потоками, являются наследниками `Stream`
- Служат для обеспечения общего способа просмотра источников данных
- Поток «экранирует» программиста от специфических особенностей операционной системы и физических устройств

# ОПЕРАЦИИ ПОТОКА

**Потоки поддерживают операции:**

- Чтение – перенос данных из потока в структуры данных
- Запись – перенос данных в поток из источника данных
- Поиск – определение и изменение текущей позиции внутри потока

# ПРИМЕРЫ ИНТЕРФЕЙСОВ РАЗНЫХ ПОТОКОВ

## `FileStream` Класс

### Свойства

| | |
|---|---|
| CanRead | Получает значение, указывающее, поддерживает ли текущий поток чтение. |
| CanSeek | Получает значение, указывающее, поддерживает ли текущий поток поиск. |
| CanTimeout | Возвращает значение, которое показывает, может ли для данного потока истечь время ожидания. (Унаследовано от Stream) |
| CanWrite | Получает значение, указывающее, поддерживает ли текущий поток запись. |
| Handle | **Является устаревшей.** |

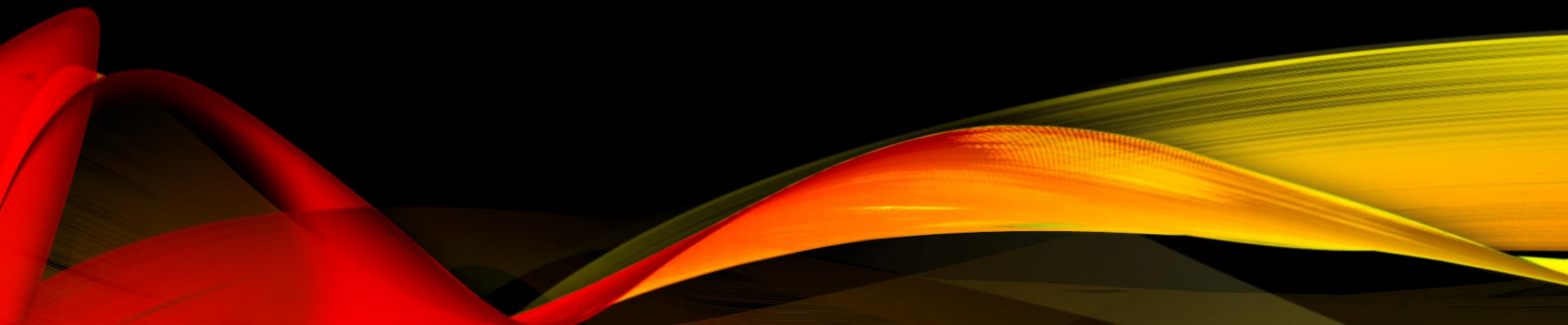
## `NetworkStream` Класс

### Свойства

| | |
|---|---|
| CanRead | Возвращает значение, указывающее, поддерживает ли объект NetworkStream чтение. |
| CanSeek | Получает значение, указывающее, поддерживает ли поток поиск. Данное свойство в настоящий момент не поддерживается. Данное свойство всегда возвращает значение `false`. |
| CanTimeout | Указывает, применимы ли для объекта NetworkStream свойства тайм-аута. |
| CanWrite | Получает значение, указывающее, поддерживает ли объект NetworkStream запись. |
| DataAvailable | Возвращает значение, указывающее, имеются ли в объекте NetworkStream данные, доступные для чтения. |

# ФАЙЛОВЫЕ ПОТОКИ

## FileStream



# ВОЗМОЖНОСТИ КЛАССА FILESTREAM

```
public class FileStream : System.IO.Stream
```

Для создания экземпляра класса `FileStream` необходимо явно или неявно указать:

- Файл, с которым будет ассоциирован поток
- Способ доступа к файлу (`FileAccess`)
- Режим открытия файла (`FileMode`)
- Допуск к совместному использованию (`FileShare`)

# КОНСТРУКТОРЫ КЛАССА FILESTREAM

**FileStream (string имя_файла, FileMode режим)**

**FileStream (string имя_файла, FileMode режим, FileAccess доступ)**

```
FileStream f1 = new FileStream("data.dat", FileMode.Create);

FileStream f2 = new FileStream("data.bin", FileMode.Open,  FileAccess.Write);
```

# МЕТОДЫ КЛАССОВ FILE И FILEINFO, ВОЗВРАЩАЮЩИЕ ССЫЛКУ НА ЭКЗЕМПЛЯР FILESTREAM

| Имя | Назначение |
|---|---|
| **Create()** | Создает новый файл и поток, ассоциированный с созданным файлом. Либо создает поток, ассоциированный с существующим файлом. |
| **Open()** | Открывает файл и возвращает ссылку на поток, ассоциированный с ним. С помощью параметра может быть указано назначение потока (чтение, запись и.т.д) |
| **OpenRead()** | Открывает файл и создает поток, предназначенный только для чтения. |
| **OpenWrite()** | Открывает файл и создает поток, предназначенный только для записи. |

# ПЕРЕЧИСЛЕНИЕ FILEACCESS

| Имя элемента | Смысл |
|---|---|
| **Read** | Разрешено только чтение |
| **ReadWrite** | Разрешены и запись и чтение |
| **Write** | Разрешена только запись |

# ПЕРЕЧИСЛЕНИЕ FILEMODE (РЕЖИМ ОТКРЫТИЯ ФАЙЛА)

| Имя элемента | Смысл |
|---|---|
| **Append** | Открыть существующий файл для дополнений или создать новый. Указатель позиции установить в конец потока |
| **Create** | Создать новый файл для записи. Если существует одноименный, уничтожить. |
| **CreateNew** | Создать новый файл для записи. Если существует одноименный, генерируется исключение. |
| **Open** | Открыть существующий файл. Если файл отсутствует, генерируется исключение. |
| **OpenOrCreate** | Если файл существует, открыть его сохранив информацию. Иначе – создать новый. |
| **Truncate** | Открыть существующий файл, очистив его. Если файл отсутствует, генерируется исключение. |

# ИСКЛЮЧЕНИЯ, ВОЗНИКАЮЩИЕ ПРИ ПОПЫТКЕ СОЗДАНИЯ ОБЪЕКТА

| **Имя** | **Причина генерации** |
|---|---|
| **IOException** | Файл невозможно открыть из-за ошибки ввода-вывода. |
| **FileNotFoundException** | Файл невозможно открыть по причине его отсутствия. |
| **ArgumentNullException** | Имя файла представляет собой null- значение. |
| **ArgumentException** | Параметр Mode некорректен. |
| **SecurityException** | Пользователь не обладает правами доступа к файлу. |
| **DirectoryNotFoundException** | Имя каталога задано некорректно. |

# ПРИМЕНЕНИЕ МЕТОДОВ КЛАССОВ FILEINFO, FILESTREAM

```
// создаём объект, но не файл
FileInfo fi1 = new FileInfo("fileTest.txt");

// создаём файл и байтовый поток
FileStream fs1 = fi1.Open(FileMode.OpenOrCreate);

// создаются и потоковый объект и файл
FileStream fs = File.Create(@"C:\!\test.txt");

fi1.Delete();   // удалить файл, представленный объектом fi1
fs1.Close();    //закрыть поток
fs.Dispose();   // освобождаем память от ресурсов потока
```

FileStream резализует интерфейс IDisposable

После использования потока, выделенные ресурсы удаляются прямо (вручную) или косвенно

using

Прямое освобождение ресурсов потока. Так не делают. Требуется использовать try / catch, где и вызывать Dispose()

# ПРИМЕР РАБОТЫ С КЛАССОМ FILESTREAM

```
using System;
using System.IO;
```

```
FileInfo fi = new FileInfo("Alphabet.txt");
using (FileStream fs = fi.Open(FileMode.OpenOrCreate)) {
    long len = fs.Length;   // Размер файла
    if (len == 26)
        Console.WriteLine("Алфавит собран!");
    else {
        if (len == 0)
            Console.WriteLine("Файл пуст!");
        fs.Seek(len, SeekOrigin.Begin);
        byte bt = (byte)('A' + len);
        fs.WriteByte(bt);
        Console.WriteLine("Добавляем в файл букву " + (char)bt);
    }
    Console.WriteLine("Буквы в файле:");
    fs.Seek(0, SeekOrigin.Begin);
    int u;
    while ((u = fs.ReadByte()) != -1)
        Console.Write((char)u + " ");
    Console.WriteLine();
}
```

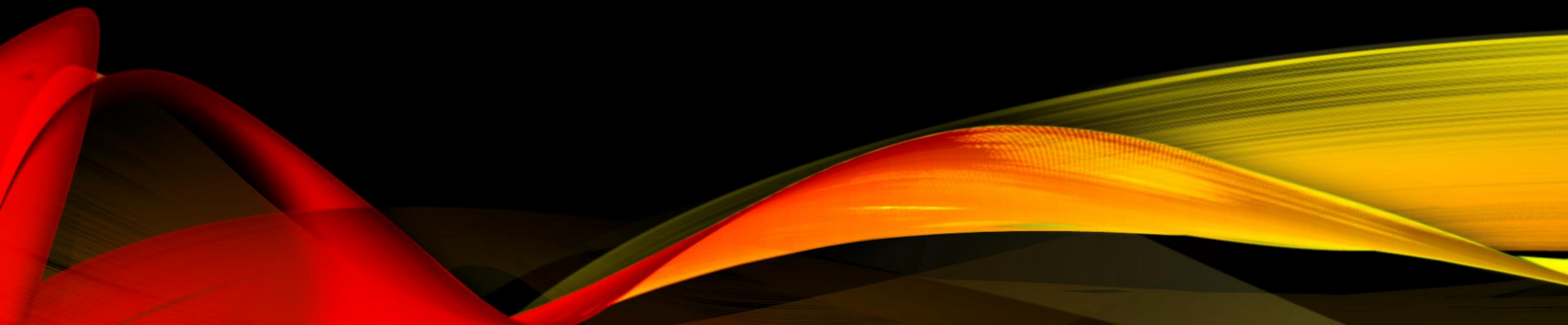
Используем **using** для косвенного освобождения ресурсов потока. **Dispose()** неявно будет вызван перед выходом из блока, связанного с **using**

```
Нет необходимости в:
fs.Flush();
fs.Close();
fs = null;
```

На уровне IL никаких using нет, есть только try / catch
using – синтаксический сахар

# КОНТРОЛИРУЕМОЕ УПРАВЛЕНИЕ РЕСУРСОВ

## IDisposable



# ИНТЕРФЕЙС IDISPOSABLE

- Интерфейс `IDisposable` и подход к его реализации предназначены для обеспечения возможности контролируемого освобождения ресурсов без необходимости ожидания момента сборки мусора
- Это бывает полезно, когда некоторый тип захватывает какие-либо внешние ресурсы

```
// Для освобождения неуправляемых ресурсов.
[ComVisible(true)]
public interface IDisposable
{
// Выполняем задачи по освобождению ресурсов.
    void Dispose();
}
```

# РЕАЛИЗАЦИЯ IDISPOSABLE – ПРОСТОЙ СЛУЧАЙ

- Данная реализация предназначена для опечатанных классов без неуправляемых ресурсов:

```
public sealed class SealedClass : IDisposable
{
    public void Dispose()
    {
        // Освободить управляемые ресурсы, т.е.
        // вызвать Dispose() на всех членах.
    }
}
```

# РЕАЛИЗАЦИЯ IDISPOSABLE – ОБЩИЙ СЛУЧАЙ (1)

```
class BaseClass : IDisposable
{
    // Флаг для проверки: был ли уже вызван Dispose()?
    bool disposed = false;

    // Общедоступная реализация Dispose(),
    // вызываемая пользовательским кодом:
    public void Dispose()
    {
        // См. реализацию на следующем слайде.
        Dispose(true);
        // не вызывать финализатор сборщику мусора:
        GC.SuppressFinalize(this);
    }

    // Продолжение на следующем слайде…
}
```

# РЕАЛИЗАЦИЯ IDISPOSABLE – ОБЩИЙ СЛУЧАЙ (2)

```
// Защищённая реализация Dispose(bool), доступная для переопределения:
protected virtual void Dispose(bool disposing)
{
    if (disposed)
        return;
    if (disposing) {
        // Освободить управляемые ресурсы…
    }
    // Освободить неуправляемые ресурсы…
    disposed = true; // Установить флаг, что очистка ресурсов выполнена.
}

// Финализатор имеет смысл только при использовании
// неуправляемых ресурсов непосредственно в BaseClass:
~BaseClass() {
    Dispose(false); // По сути ~BaseClass – и есть Finalize().
}
```

# ОПЕРАТОР USING

```
using (IDisposable аргумент)
{
    // операторы
}

// Эквивалент оператора using:
try { .     .     .     .     }
finally     {
    аргумент.Dispose();
}
```

Аргумент `using` должен иметь тип, реализовавший интерфейс `System.IDisposable`

# ПРИМЕР ИСПОЛЬЗОВАНИЯ USING

```
using (FileStream inFi = new FileStream(@"..\..\..\Program.cs", FileMode.Open))
{
    int t;       // числовое значение прочитанного байта
    int k = 0;   // позиция байта в потоке (в файле)
    while ((t = inFi.ReadByte()) != -1)
    {
        if (t >= '0' && t <= '9')
            Console.WriteLine(t + " - " + (char)t + " - " + k);
        k++;
    }   // End of while.
}
```

# ЭКВИВАЛЕНТ ОПЕРАТОРА USING

```
FileStream inFi = null;
try
{
    inFi = new FileStream(@"..\..\..\Program.cs", FileMode.Open);
    int t;      // числовое значение прочитанного байта
    int k = 0;  // позиция байта в потоке (в файле)
    while ((t = inFi.ReadByte()) != -1)
    {
        if (t >= '0' && t <= '9')
            Console.WriteLine(t + " - " + (char)t + " - " + k);
        k++;
    }   // while
}
finally
{
    (inFi as IDisposable)?.Dispose();
}
```

Вызов Dispose()

# ССЫЛКИ

- Класс Stream (https://learn.microsoft.com/ru-ru/dotnet/api/system.io.stream?view=net-7.0)
- FileStream класс (https://learn.microsoft.com/ru-ru/dotnet/api/system.io.filestream?view=net-7.0)
- System.BitConverter (https://learn.microsoft.com/ru-ru/dotnet/api/system.bitconverter?view=net-7.0)
- Класс MarshalByRefObject (https://learn.microsoft.com/ru-ru/dotnet/api/system.marshalbyrefobject?view=net-7.0)