

# Файлы

Лекции по курсу  
Языки программирования  
семестр 2

# Определение. Преимущества

- Файл – это именованная область данных на диске, хранящая данные
- Преимущества файлов:
  - Сохраняют данные между запусками программ
  - Могут быть большого размера
- Недостатки:
  - Скорость работы существенно меньше чем скорость доступа к оперативной памяти

# Классификация файлов

- По типу компонент:
  - текстовые (состоят из строк, разделённых разделителями строк. Символы записываются в некоторой **кодировке**)
  - двоичные (бинарные)
- По способу доступа:
  - Произвольный: время доступа -  $\Theta(1)$ , компоненты – одинакового размера (в простейшем случае - байты)
  - Последовательный: компоненты – разного размера, для перехода к  $i$ -той компоненте надо просмотреть предыдущие  $i-1$  компонент (текстовые файлы)
- По операциям чтения-записи:
  - только чтение
  - только запись
  - чтение и запись

# Операции с файлами

## С закрытыми файлами:

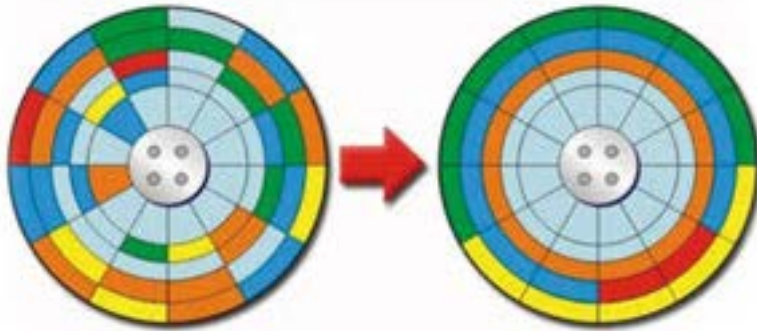
- открыть
- проверить существование
- удалить
- скопировать
- переименовать
- переместить
- проверить уровень доступа

## С открытыми файлами:

- закрыть
- записать информацию
- считать информацию
- проверить, не достигнут ли конец файла (EOF)
- переместить позицию файлового указателя в заданное место

# Фрагментация файлов на диске

- Влияет на скорость доступа
- Специальные программы-дефрагментаторы



# Основные пространства имён и классы .NET

## Пространства имён:

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

## Классы:

**File** – статический класс, содержащий операции с закрытыми файлами

**FileMode** – режим открытия файла

**FileStream** – файловый поток

**StreamReader/StreamWriter** – потоки для чтения-записи в текстовый файл

**BinaryReader/BinaryWriter** – потоки для чтения-записи в двоичный файл

**Encoding** – кодировка текстовых файлов

# Работа с закрытыми файлами

- `File.Exists("a.dat")`
- `File.Exists("d:\\a.dat")`
- `File.Exists(@"d:\a.dat")`
- `File.Delete("a.dat")`
- `File.Copy("a.dat", "b.dat")`
- `File.Copy("a.dat", @"d:\a.dat")`
- `File.Move("a.dat", "b.dat")`
- `File.Move("a.dat", @"..\a.dat")`

# Поток FileStream – файл как поток байтов

```
var path = @"d:\a.dat";
```

```
var fs = new FileStream(path, FileMode.Create);  
fs.WriteByte(1);  
fs.WriteByte(2);  
fs.WriteByte(3);  
fs.Close();
```

```
fs = new FileStream(path, FileMode.Open);  
WriteLine(fs.ReadByte());  
WriteLine(fs.ReadByte());  
WriteLine(fs.ReadByte());  
fs.Close();
```

- Поток FileStream обеспечивает только базовую функциональность – чтение и запись байтов
- Для чтения/записи структурированных данных используются обёртки:
  - **StreamReader/StreamWriter** – для чтения-записи в текстовый файл
  - **BinaryReader/BinaryWriter** – для чтения-записи в двоичный файл



# Оператор using

```
var path = @"d:\a.dat";
```

```
using (var fs = new FileStream(path, FileMode.Create))  
{  
    fs.WriteByte(1);  
    fs.WriteByte(2);  
    fs.WriteByte(3);  
}
```

```
using (var fs = new FileStream(path, FileMode.Open))  
{  
    WriteLine(fs.ReadByte());  
    WriteLine(fs.ReadByte());  
    WriteLine(fs.ReadByte());  
}
```

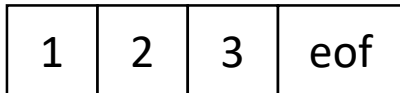
- Оператор using обеспечивает автоматический вызов **fs.Close()**
- Действие переменной fs распространяется только на блок using

# Режим открытия файла FileMode

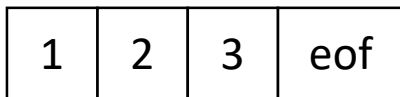
- **FileMode.Create** – создать файл. Если создан, то усечь его
- **FileMode.Open** – открыть существующий файл. Если его не существует – исключение
- **FileMode.OpenOrCreate** – открыть файл. Если его не существует, то создать
- **FileMode.Append** – открыть файл на добавление (поставив файловый указатель в конец). Если его не существует, то создать

# Понятие файлового указателя

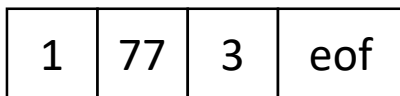
- С каждым открытым файлом связан файловый указатель – текущая позиция в файле
- При открытии файла файловый указатель устанавливается по обычно в начало файла
- При каждой операции чтения-записи файловый указатель продвигается вперёд



```
fs = new FileStream(path, FileMode.Open)
```



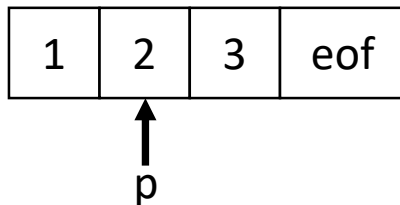
```
var a = fs.ReadByte(); // a = 1
```



```
fs.WriteByte(77);
```

# Действия с файловым указателем

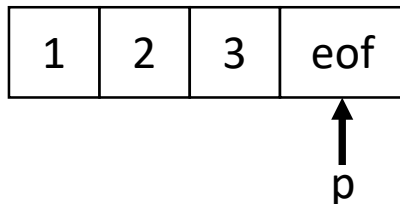
- `fs.Position` – текущая позиция файлового указателя
- `fs.Length` – длина потока в байтах
- `fs.Seek(1,SeekOrigin.Begin)` – установка позиции файлового указателя относительно начала файлового потока
- `fs.Seek(1,SeekOrigin.End)` – относительно конца
- `fs.Seek(1,SeekOrigin.Current)` – относительно текущей позиции



```
fs.Position = fs.Position + 1;
```

ИЛИ

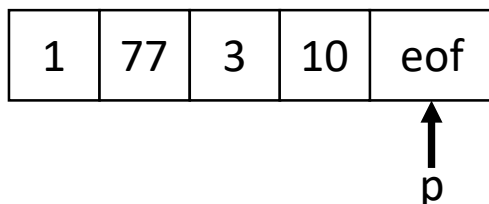
```
fs.Seek(1,SeekOrigin.Current);
```



```
fs.Seek(fs.Length,SeekOrigin.Begin);
```

ИЛИ

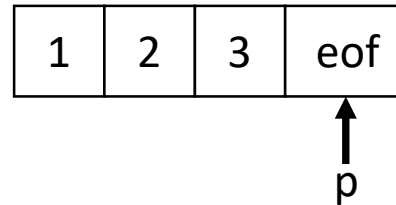
```
fs.Seek(0,SeekOrigin.End);
```



```
fs.WriteByte(10);
```

# Цикл по файлу

```
int b;  
do {  
    b = fs.ReadByte();  
    // обработка b  
} while (b != -1)
```



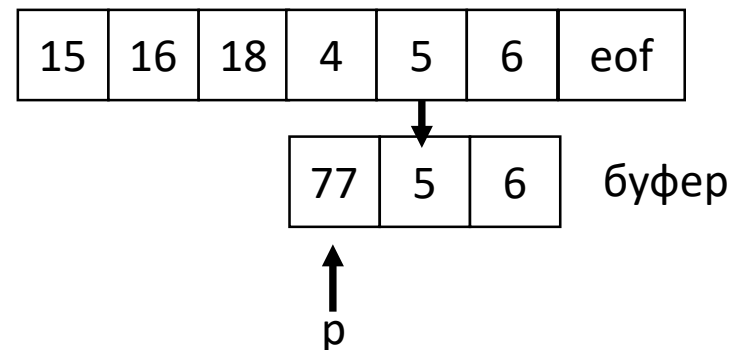
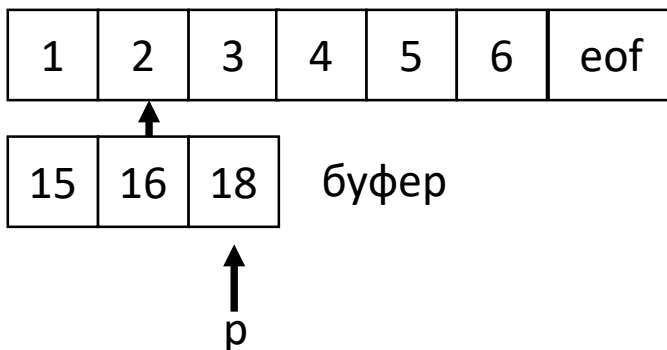
- При чтении за концом файла `fs.ReadByte()` возвращает -1  
Это является признаком конца файла

```
var end = fs.Position == fs.Length;
```

- Чтобы узнать, находимся ли мы на конце файла, необходимо сравнить позицию с длиной файла

# Буферизация файла

- С файловым потоком связан буфер – область оперативной памяти (по умолчанию около 2-4 Кб), ускоряющая операции при работе с файлами
- При чтении данные считываются из буфера, а если достигнут конец буфера, то в буфер считывается следующая порция информации из файла
- При записи данные записываются в буфер, а если буфер переполнен, то его содержимое сбрасывается в файл и в него считывается следующая порция информации из файла
- При закрытии буфер сбрасывается в файл
- Если не закрыть файл, который мы меняли, то можно потерять информацию из последнего буфера



# Потоковые адаптеры для текстовых файлов

- Потоковые **адаптеры** представляют обёртку над файловыми потоками для чтения-записи структурированных данных
- Перед созданием потокового адаптера создаётся файловый поток `FileStream`, и ссылка на него хранится внутри потокового адаптера
- Закрытие потокового адаптера приводит к закрытию файлового потока

```
var sw = new StreamWriter("a.txt");  
    // автоматически создаётся FileStream  
sw.WriteLine("Hello\nФИИТ");  
sw.Close(); // автоматически вызывается FileStream.Close()
```

```
using (var sw = new StreamWriter("a.txt"))  
{  
    sw.WriteLine("Hello\nФИИТ");  
}
```

# Вывод числовой информации в текстовые потоки

- В текстовые потоки можно выводить числовую информацию:

```
using (var sw = new StreamWriter("a.txt"))
{
    var i = 1;
    var r = 3.14;
    sw.Write(i);
    sw.Write(' ');
    sw.WriteLine(r);
}
```



# Цикл считывания строк текстового файла

- Признаком конца файла при считывании из текстового потока является то, что `sr.ReadLine()` возвращает null-строку
- Второй способ – обращение к свойству `sr.EndOfStream`

## Цикл по строкам файла - первый способ

```
using (var sr = new StreamReader("a.txt"))
{
    string line;
    while ((line = sr.ReadLine()) != null)
        Console.WriteLine(line);
}
```

## Цикл по строкам файла - второй способ

```
using (var sr = new StreamReader("a.txt"))
{
    while (!sr.EndOfStream)
        Console.WriteLine(sr.ReadLine());
}
```

# Кодировки файлов

- Кодировка – таблица соответствия символов и их кодов
- Кодировки: однобайтовые (устарели) и Unicode
- Unicode – стандарт кодирования символов (с 1991 г.). Состоит из универсального набора символов (и их кодов) и семейства кодировок (*Unicode transformation format, UTF* – для преобразования кодов символов при передаче в файле)
- Кодировка задаётся типом `System.Text.Encoding`
- **Encoding.ASCII** задаёт кодировку ASCII (первые 128 символов набора Unicode – английские буквы, цифры, знаки препинания записываются в файл как 1 байт, остальные заменяются знаком ?)
- **Encoding.UTF8** задаёт кодировку Utf-8: первые 128 символов кодируются в 1 байт (совместимо с ASCII), а остальные – как правило в 2 байта. **Наиболее часто используется**
- **Encoding.Unicode** задаёт кодировку Utf-16: все символы кодируются 2 байтами (в платформе .NET)
- **Encoding.Default** – однобайтовая кодировка, принятая в ОС по умолчанию
- При открытии файла может указываться кодировка

```
var sr = new StreamReader("a.txt", Encoding.Default);  
var sr = new StreamReader("a.txt", Encoding.UTF8);  
var sw = new StreamWriter("a.txt", false, Encoding.UTF8)
```



режим append

# Использование методов класса File для создания текстовых потоков

- File.OpenText("a.txt");
- File.CreateText("a.txt");
- File.AppendText("a.txt");

Все эти методы открывают файл в кодировке **Utf-8**

```
using (StreamWriter sw = File.CreateText("a.txt"))
{
    sw.WriteLine("Hello");
}

using (StreamWriter sw = File.AppendText("a.txt"))
{
    sw.WriteLine("ФИИТ");
}

using (StreamReader sr = File.OpenText("a.txt"))
{
    while (!sr.EndOfStream)
        WriteLine(sr.ReadLine());
}
```

# Обработка исключений

- Исключение – объект, который генерируется в ответ на ошибку времени выполнения программы
- Все исключения – наследники класса `System.Exception`
- Механизм обработки исключений вводился с целью единообразной обработки ошибок времени выполнения
- Исключение возникает в одном месте, а может быть перехвачено в другом (объемлющем) блоке. В момент возникновения исключения программист может не знать, что с ним делать. Исключение обрабатывается в тот момент, когда понятно, что с ним делать
- Исключения – это не обязательно ошибки в программе. Например, ошибки ввода-вывода, связанные с отсутствием файла на диске, не вина программиста
- Пример обработки исключения:

```
try
{
    var sr = new StreamReader("d:\\b.txt");
    ...
}
catch (FileNotFoundException e)
{
    WriteLine(e.Message);
    WriteLine(e.FileName);
    WriteLine(e.StackTrace);
}
```

# Потоковые адаптеры для двоичных файлов

- Двоичные адаптеры BinaryReader/BinaryWriter пишут данные базовых типов byte, int, char, double, bool и т.д. так как они хранятся в оперативной памяти (что более эффективно)
- Метод bw.Write() перегружен для всех базовых типов
- Двоичному адаптеру в конструкторе обязательно передаётся FileStream
- Закрытие двоичного адаптера автоматически приводит к закрытию файлового потока. Повторный вызов FileStream.Close() не порождает ошибку

```
using (FileStream fs = File.Create(@"d:\a.dat"))
using (var bw = new BinaryWriter(fs))
{
    bw.Write(1);
    bw.Write(3.14);
    bw.Write('z');
    bw.Write("ФИИТ");
    bw.Write(true);
}
```

# Считывание из двоичных файлов

- Для считывания из двоичных файлов в потоке `BinaryReader` предусмотрены методы `ReadInt32()`, `ReadDouble()`, `ReadChar()`, `ReadBoolean()`, `ReadString()` и т.д.
- Если типы разные, то важно читать данные в том порядке, в котором они были записаны

```
using (var fs = File.Open(@"d:\a.dat", FileMode.Open))
using (var br = new BinaryReader(fs))
{
    var i = br.ReadInt32();
    var r = br.ReadDouble();
    var c = br.ReadChar();
    var s = br.ReadString();
    var b = br.ReadBoolean();
}
```

# Цикл по двоичному файлу

- Если двоичный файл содержит одинаковые типы, то по нему возможен цикл
- Условие конца файла проверяется хитрым образом: `br.PeekChar() != -1`

```
var path = @"d:\a.dat";
using (var fs = File.Create(path))
using (var bw = new BinaryWriter(fs))
{
    for (int i = 0; i < 10; i++)
        bw.Write(i*1.0);
}
```

```
using (var fs = File.Open(path, FileMode.Open))
using (var br = new BinaryReader(fs))
{
    while (br.PeekChar() != -1)
    {
        WriteLine(br.ReadDouble());
    }
}
```

# Исключение EndOfStreamException

- При попытке считывания за концом двоичного файла возникает исключение EndOfStreamException

```
using (var fs = File.Open(path, FileMode.Open))
using (var br = new BinaryReader(fs))
{
    while (br.PeekChar() != -1)
    {
        WriteLine(br.ReadDouble());
    }
    br.ReadDouble();
}
```

**Необработанное исключение: System.IO.EndOfStreamException: Чтение после конца потока невозможно.**



# Исключение EndOfStreamException

- При попытке считывания за концом двоичного файла возникает исключение EndOfStreamException

```
using (var fs = File.Open(path, FileMode.Open))
using (var br = new BinaryReader(fs))
{
    try
    {
        while (br.PeekChar() != -1)
        {
            WriteLine(br.ReadDouble());
        }
        br.ReadDouble();
    }
    catch (EndOfStreamException e)
    {
        // обработать
    }
}
```

# Возведение всех элементов в квадрат

```
var path = @"d:\a.dat";
using (var fs = File.Create(path))
using (var bw = new BinaryWriter(fs))
{
    for (int i = 0; i < 10; i++)
        bw.Write(i*1.0);
}

using (var fs = File.Open(path, FileMode.Open))
using (var br = new BinaryReader(fs))
using (var bw = new BinaryWriter(fs))
{
    var len = fs.Length / sizeof(double);

    for (int i=0; i<len; i++)
    {
        var x = br.ReadDouble();
        bw.Seek(-sizeof(double), SeekOrigin.Current);
        bw.Write(x * x);
    }
}
```

# Текстовые файлы как последовательности строк

С помощью методов `File.ReadLines`, `File.ReadAllLines` `File.WriteAllLines` текстовый файл преобразуется в последовательность строк и обратно. При этом файл автоматически открывается, последовательность строк считывается (записывается) и затем файл закрывается

```
IEnumerable<string> q = File.ReadLines("a.txt"); // UTF-8
q = q.Reverse();
// другие преобразования
File.WriteAllLines("b.txt", q);
File.Erase("a.txt");
File.Move("b.txt", "a.txt");
```

ИЛИ

```
string[] q = File.ReadAllLines("a.txt");
q = q.Reverse();
// другие преобразования
File.WriteAllLines("a.txt", q);
```

# Добавление строки в начало (конец) файла

**В конец файла:**

```
string s = "add";  
File.AppendAllLines("a.txt", new string[] { s });
```

**В начало файла:**

```
var q = File.ReadLines("a.txt");  
q = q.Reverse();  
File.WriteAllLines("b.txt", new string[] { s }.Concat(q));  
File.Erase("a.txt");  
File.Move("b.txt", "a.txt");
```

последовательность из одной строки



слияние двух последовательностей



# Парсинг строк в текстовом файле

Парсинг – разбор и перевод во внутреннее представление

Пример – в текстовом файле записаны строки вида

```
Иванов 62 77 80
Попова 98 93 89
Сидоров 63 99 100
```

**Программа :**

```
class Person {
    string Name;
    public int Age;
    public Person(string Name, int Age) {
        this.Name = Name;
        this.Age = Age;
    }
}
...
var q = File.ReadLines("a.txt").Select(s => s.Split())
    .Select(ss => new Person(ss[0], int.Parse(ss[1])));
WriteLine(q.Max(p => p.Age));
```

← Конкретный запрос

# Q & A