C# programming language. The beginning

Course

Programming Languages

Semester 2, FIIT

Mayer S.F.

Mikhalkovich S.S

LECTURE # 14, 15. Files

Files

Lesson # 14

Definitions. Advantages

- **File** is the named area on disk which stores data
- **Advantages**:
  - File stores data between program executions
  - The storage can be large or huge (if we use variables, without files, it can be ineffective)
- **Disadvantages**:
  - Access speed is significantly smaller than the speed of access to variables in a program

Files classification

- On component type:
  - text files (consists of strings in some encoding)
  - binary files
- On access type:
  - Arbitrary: access time = $\Theta(1)$, components have the same size (bytes in the simplest case)
  - Sequential: components have different sizes, for seeking to the i-th component it's necessary to pass the preceding i-1 components (example: text files)
- On operations type:
  - only reading
  - only writing
  - reading and writing

File operations

With closed files:

- open
- check existence
- delete
- copy
- rename
- move

With opened files:

- close
- write info
- read info
- check the end of file (EOF)
- seek to the n-th position

The main .NET classes & namespaces

Namespaces:

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

Classes:

`File` – static class, operations with closed files
`FileMode` – mode of file opening
`FileStream`
`StreamReader`/**`StreamWriter`** – streams for reading-writing to text files
`BinaryReader`/**`BinaryWriter`** - streams for reading-writing to binary files
`Encoding` – file encodings

Operations with closed files

File class

```
File.Copy(@"E:\test.txt", @"E:\testFolder\test.txt");
File.Move(@"E:\test.txt", @"E:\testFolder\test.txt");
File.Delete(@"E:\test.txt");
```

- 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")

* means any filename and any file extension

```
string Path = @"E:\";
// output the files in some folder
var files = System.IO.Directory.GetFiles(Path, "*.*",
SearchOption.TopDirectoryOnly);
foreach (var file in files)
      {
            Console.WriteLine(file);

      }
```

FileStream class

Working with bytes

FileStream class – file as a stream of bytes

WriteByte & **ReadByte** methods only

```
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()); //1
WriteLine(fs.ReadByte()); //2
WriteLine(fs.ReadByte()); //3
fs.Close();
```

Can only write byte to file

Read from file

- FileStream provides only basic functionality – reading-writing of bytes
- For reading-writing of structured data:
  - **StreamReader/StreamWriter** – For reading-writing in text files
  - **BinaryReader/BinaryWriter** – For reading-writing binary files

FileStream class – to work with text

```
FileStream fs = new FileStream(@"D:\my.txt", FileMode.Create);
string s = "Hello world\n Goodbye";
// sequence of bytes, using GetBytes method
byte[] sByte = Encoding.Default.GetBytes(s);
fs.Write(sByte, 0, sByte.Length);

fs.Position = 0; // to set the file pointer to the beginning:
byte[] bytesFromFile = new byte[fs.Length];
for (var i = 0; i < fs.Length; i++)
{
    bytesFromFile[i] = (byte)fs.ReadByte();
}
Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
fs.Close();
```

Can only write byte to file
Using System.Text is used for encoding

Read from file

- For reading-writing of structured data it is better to use:
  - **StreamReader/StreamWriter** – For reading-writing in text files
  - **BinaryReader/BinaryWriter** – For reading-writing binary files

Exception handling

Try …catch

Try … Finally

Using

Exception handling

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

try .. catch statement

```
try
{
  // code with possible exception
}
catch (Exception e)
{
  // handle of an exception
}
```

Example

```
try
{
  var path = @"d:\a.dat";
  var fs = new FileStream(path, FileMode.Open);
  ...
}
catch (FileNotFoundException e)
{
  WriteLine($"File {e.FileName} is mot found");
}
```

If exception is thrown

```
try
{
  var path = @"d:\a.dat";
  var fs = new FileStream(path, FileMode.Open);
  ...
}
catch (FileNotFoundException e)
{
  WriteLine ($"File {e.FileName} is mot found");
}
```

If exception isn't thrown

```
try
{
  var path = @"d:\a.dat";
  var fs = new FileStream(path, FileMode.Open);
  ...
}
catch (FileNotFoundException e)
{
  WriteLine($"File {e.FileName} isn't found");
}
catch (AnotherException e)
{
  ...
}
```

try .. finally

```
var path = @"d:\a.dat";
var fs = new FileStream(path, FileMode.Open);
…
var i = int.Parse("abc");
fs.Close();
```

FormatException

File will not be closed

try .. finally

Solution

```
var path = @"d:\a.dat";
var fs = new FileStream(path, FileMode.Open);
try
{
  ...
  var i = int.Parse("abc");
  ...
}
finally
{
  fs.Close();
}
```

This line will be executed regardless of whether the exception was or not

using statement

```
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());
}
```

- **using** statement provides an automatic call of **fs.Close()**
- **using** is equivalent to code with **try … finally**, so file will be closed automatically in any situation

FileMode enum type

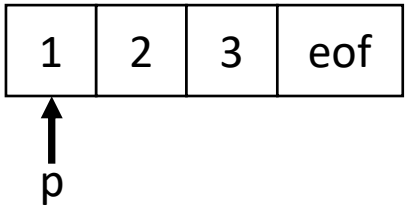
- **FileMode.Create**
- **FileMode.Open**
- **FileMode.OpenOrCreate**
- **FileMode.Append**

File pointer

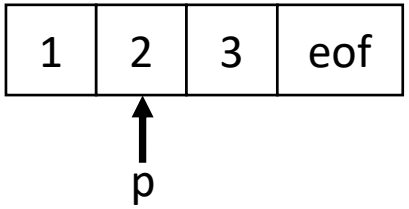
While working with binary files

File pointer

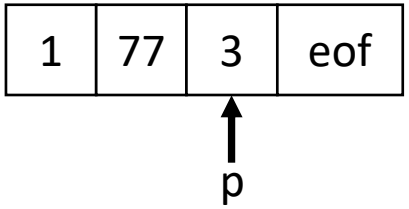
- File pointer is associated with every opened file
- File pointer gives an access to the current position in the file
- After opening the file, File pointer is located at the beginning of the file data
- After every read-write operation, File pointer is moved forward




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



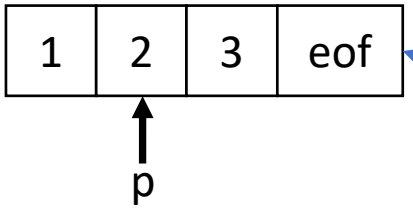

```
var a = fs.ReadByte(); // a = 1
```



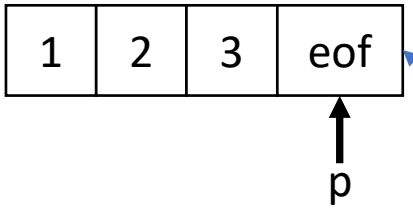

```
fs.WriteByte(77);
```

Actions with file pointer

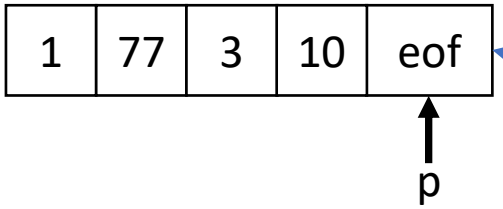
- **fs.Position** – current position of a file pointer
- **fs.Length** – stream length in bytes
- **fs.Seek(1,SeekOrigin.Begin)** – positioning of a file pointer relatively to the beginning of a file stream
- **fs.Seek(1,SeekOrigin.End)** – relatively to the end of a file stream
- **fs.Seek(1,SeekOrigin.Current)** – relatively to the current position




```
fs.Position = fs.Position + 1;
or
fs.Seek(1, SeekOrigin.Current);
```

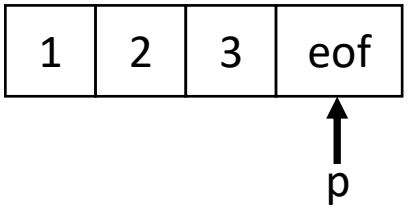



```
fs.Seek(fs.Length,SeekOrigin.Begin);
or
fs.Seek(0,SeekOrigin.End);
```




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

Loop over file elements




```
var sum = 0;
do
{
    var b = fs.ReadByte();
    if (b == -1)
        break;
    sum += b;
} while (true);
```

```
var sum = 0;
for (var i = 0; i<fs.Length; i++)
{
    var b = fs.ReadByte();
    sum += b;
}
```

- If we read after the end of file, fs.ReadByte() returns -1
  This situation is called End Of File (**EOF**)

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

- How to find out the EOF situation? It's necessary to compare a Position of the file with a file Length

Stream adapters

For binary files

Stream adapters for binary files

- Binary adapters **BinaryReader**/**BinaryWriter** write data of base types byte, int, char, double, bool etc. in the same form as they are stored in memory
- We can open BinaryReader and BinaryWriter for one file at the same time (read-write access)
- The method bw.Write() is overridden for all base types

```
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("FIIT");
    bw.Write(true);
}
```

Reading from binary files

```
using (FileStream fs = File.Open("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();
    Console.WriteLine($"{i} {r} {c} {s} {b}");
}
```

Reading from binary files

- Reading from binary files the BinaryReader has methods ReadInt32(), ReadDouble(), ReadChar(), ReadBoolean(), ReadString() etc.
- It's necessary to read in the same order that we write these data

```
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();
}
```

Loop over binary file

- If the binary file contains the same type data then we can do a loop by the elements of a file
- For file of reals: f number of reals can be found by the formula:

```
var sz = br.BaseStream.Length / sizeof(double);
```

where br is `BinaryReader`:

```
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))
{
    var sz = fs.Length / sizeof(double);
    for (var i=0; i<sz; i++)
    {
        Console.WriteLine(br.ReadDouble());
    }
}
```

Loop over file elements

```
while (br.PeekChar() != -1)
{
    var x = br.ReadInt32();
    sum+ = x;
}
```

- If we use BinaryReader, we have PeekChar() method. Returns **-1** if there is no any more character to read

Stream adapters

For text files

Stream adapters for the text files

- Stream adapters are the file stream wrappers for reading-writing of a structured data
- Stream adapters for the text files: **StreamReader**/**StreamWriter**
- (!) It can be opened as `StreamWriter`, or `StreamReader`, but not both at the same time (!!)

```
var sw = new StreamWriter("a.txt");
sw.WriteLine("Hello\nFIIT");
sw.Close();
```

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

Numerical information in the text streams

- Numeric information can be output to text streams (analogous to Console.Write и Console.WriteLine):

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

a.txt:
1 3,14

- Numeric information can be input from the text streams by parsing the corresponding values (to give an example)

Reading from text streams

```
using (var sr = new StreamReader("a.txt"))
{
    var s = sr.ReadLine();
    var a = s.Split();
    string s1 = a[0];
    string s2 = a[1];
    int i = Convert.ToInt32(s1); // 1
    double r = Convert.ToDouble(s2);   // 3,14
    Console.WriteLine($"{i+r}"); // 4,14
}
```

a.txt:
1 3,14

Loop over a text file

The 1-st way

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

The 2-nd way

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

File encodings

- Encoding is a correspondence between symbols and their codes
- Unicode – is the ba (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)
- Type **System.Text.Encoding**
- **Encoding.ASCII** (The first 128 symbols of the Unicode table – english letters, digits, punctuations are stored in a file as 1 byte, the others change by the "?" char)
- **Encoding.UTF8**: The first 128 symbols are coded by 1 byte, the others are coded by 2 bytes. **The most common**
- **Encoding.Unicode** (Utf-16): all the symbols are coded by 2 bytes
- **Encoding.Default** – The default 1-byte encoding
- In the constructor we can specify the file encoding (UTF8 as a default)

```
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);
```

if true - the append mode

Static methods of File class

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

All files are opened in **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());
}
```

The EndOfStreamException

- The EndOfStreamException is thrown when reading after the end of file

```
using (var fs = File.Open(path, FileMode.Open))
using (var br = new BinaryReader(fs))
{
    br.BaseStream.Position = br.BaseStream.Length;
    WriteLine(br.ReadDouble());
}
```

```
Unhandled exception: System.IO.EndOfStreamException: reading
after the end of stream
```

Problem. Square all the elements in the file of doubles

```
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();
        fs.Position -= sizeof(double);
        bw.Write(x * x);
    }
}
```

Problem. Square all the elements in the file of reals

```
0 1 4 3 4  5  6  7  8  9
    @

0 1 4 9 16 25 36 49 64 81
```

Text files as sequences of strings

Using methods File.ReadLines, File.ReadAllLines, File.WriteAllLines we can transform text file into a sequence of strings and vise versa

- file is opened
- the sequence of strings is getting from (taking into) a file
- file is closed

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

IEnumerable<string> q = File.ReadLines("a.txt");
q = q.Where(s => s.Length > 5);
File.WriteAllLines("b.txt", q);
```

Lecture tasks

Lesson #14, 15

https://labs-org.ru/c-sharp14-eng/
https://labs-org.ru/c-sharp15-eng/

Q & A