



# Лекция 10. Файлове

# Разлика между файл и поток.

## Какво представляват потоците?

- Като концепция, потоците в C# са аналогични на потоците в други обектно-ориентирани езици, например Java, C++ и Delphi (Object Pascal).
- Файлът, както знаем се дефинира като именована и подредена последователност от байтове. Характеризира се с име и място, тоест това е нещо конкретно.
- Понятието „ПОТОК“ е едно абстрактно понятие, чрез което се обозначава динамично изменяща се във времето последователност от „нещо“ (байтове, символи и др.). Най-често, това „нещо“ са „байтове“, т.е най-често „ПОТОК“ -ът е последователност от байтове, свързани с конкретно устройство на компютъра (твърд диск, дисплей, принтер, клавиатура), получавани посредством „системата за вход/изход“.
- Системата за вход/изход обезпечава за програмиста стандартни и независещи от физическите устройства средства за представяне на информацията и за управление на потоците за вход/изход, а именно **един стандартен набор от операции** (тоест един набор от едноименни функции за вход и изход, със стандартен интерфейс).



# Видове потоци

Потоците биват: байтови, символни, двоични (бинарни)

- **Байтови:** Болшинството от устройствата, предназначени за изпълнение на операции за вход/изход се явяват байт-ориентирани. Това обяснява факта, защо, на най-ниско ниво всички операции за вход/изход манипулират с байтове, в рамките на байтове потоци.
- **Символни:** От друга страна, значителен обем задачи, като редактиране на текст, запълване на екранна форма, извеждане на информацията в нагледен вид и др. изискват работа със символи, а не с байтове. Символно-ориентираните потоци са предназначени за работа със символи а не с байтове и се явяват потоци за вход/изход от по-високо ниво. В рамките на Framework .NET има съответни класове, които при реализация на операция за вход /изход обезпечават автоматично преобразуване на данни от тип `byte` в данни от тип `char` и обратно.
- **Бинарни:** В допълнение към байтовите и символни потоци в C# са определени два класа, реализиращи механизъм на четене и запис на информация, непосредствено в двоичен вид (потоци `BinaryReader` и `BinaryWriter`).



# Обща характеристика на класовете за потоци

- За езиките на .NET, класът който описва най-общите характеристики (свойства и методи) на потоците е клас **System.IO.Stream** (намира се в именовано пространство System.IO) - **абстрактен базов клас за всички потоци**.
- Той се наследява от останалите потоци, а именно: FileStream, NetworkStream...

Stream

I-----→FileStream

I-----→BufferedStream

I-----→MemoryStream

I-----→NetworkStream



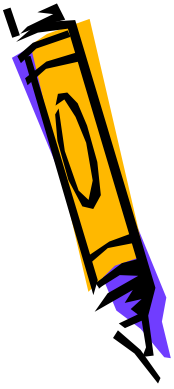
- Класовете, които имплементират **Stream** са **FileStream**, **BufferedStream**, **MemoryStream** и **NetworkStream**, като **NetworkStream** се използва при мрежови операции (не представлява обект на разглеждане тук).
- **MemoryStream** е байтов поток, използващ в качеството на източник и място за съхранение на информацията - оперативната памет.



- Класът, който реализира буфериран поток в .NET Framework, е **System.IO.BufferedStream**. Буферираните потоци използват вътрешен буфер за четене и запис на данни, с което значително подобряват производителността. Когато четем данни от някакво устройство, при заявка дори само за един байт, в буфера попадат и следващите го байтове до неговото запълване. При следващо четене, данните се взимат директно от буфера, което е много по-бързо. По този начин се извършва кеширане на данните, след което се четат кеширани данни. При запис, всички данни попадат първоначално в буфера. Когато буферът се препълни или когато програмистът извика **Flush()**, те се записват върху механизма за съхранение (пренос) на данни.



- Четенето и писането от и във файлове в .NET Framework се извършва с класа **FileStream**, който е байтов поток.
- Като наследник на **Stream**, той поддържа всичките му свойства и методи, а именно:
  - четене - `Read()`,
  - писане - `Write()`,
  - позициониране - `Seek()`



## Методите на Stream, които FileStream използва:

- Метод Read: чете данни от файла, в масив от байтове
- Приема 3 параметъра:

**int Read(byte[] array, int offset, int count)**

- и връща количеството на успешно прочетени байтове или 0 при достигне края на потока.

### Параметри:

- array - масива от байтове, в който се помещават прочетените от файла данни
- offset - представлява отместването в байтове, при запис в масива **array**
- count - максимално количество байтове за четене от входния поток.

Методът чете най-много **count** на брой байта, от текущата позиция на входния поток, записва ги в масива **array**, започвайки от индекс **offset** на масива, след което увеличава индекса на масива с (**offset+count**).





## Методите на Stream, които FileStream използва:

- Метод Write: записва във файла данните от масива с байтове.
- Приема 3 параметъра:

**Write(byte[] array, int offset, int count)**

- array - масив байтове, от който данните се записват във файла
- offset - отместването в байтове, в масива array, откъдето започва записа на байтове в потока
- count - максимално количество байтове, предназначени за запис

Методът записва в потока count количество байтове, от масива array, започвайки от array[offset]. След всеки запис, позицията в масива се увеличава с (offset+count). Методът връща количеството на успешно записани байтове.



Методите на Stream, които FileStream използва:

Метод Seek (за позициониране в поток):

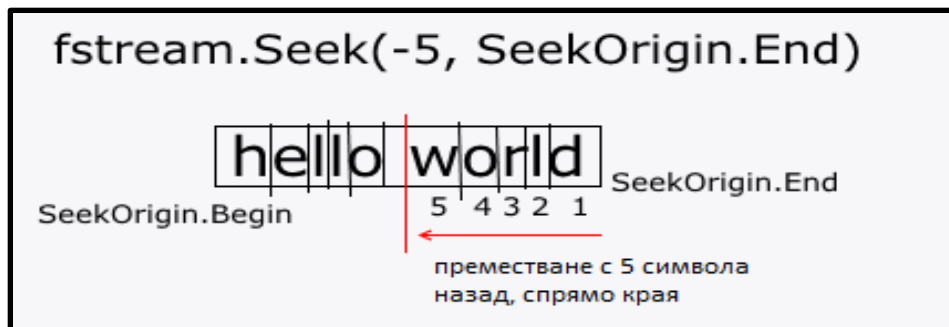
**public abstract long Seek(long offset, SeekOrigin origin)**

- премества текущата позиция на потока с offset брой байта, спрямо зададена отправна точка **origin** от тип **SeekOrigin**.
- Методът е приложим за потоците, за които CanSeek връща true, за останалите хвърля изключение NotSupportedException.
- SeekOrigin е "enum" с 3 елемента (начало, край или текуща позиция на потока):

**SeekOrigin.Begin**

**SeekOrigin.End**

**SeekOrigin.Current**



Пример: `fs.Seek(0, SeekOrigin.Begin);`

// 0 - позициониране в началото на файла



# Други методи на Stream, които FileStream използва:

// public abstract са всички методи!!

- **void WriteByte(byte b);**

Извършва запис на един байт в потока.

- **int ReadByte();**

Чете един байт от потока, връща цяло число - следващия достъпен байт



# Свойства на Stream, които FileStream използва:

- може ли да се чете от потока:

**bool CanRead**

(Ако може – резултат true);

- може ли да се пише в поток:

**bool CanWrite**

(Ако може – резултат true);



- може ли да се позиционираме в потока на зададена позиция

**bool CanSeek**

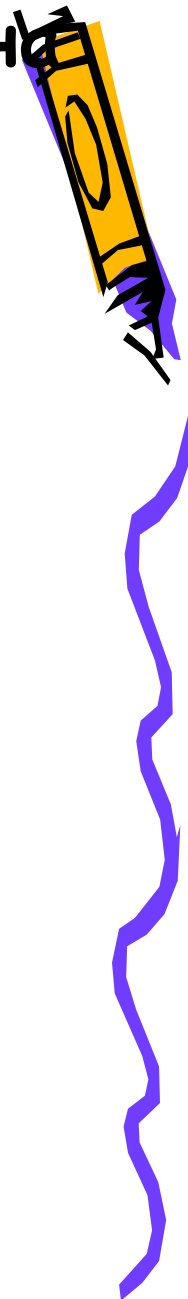
(Ако може - резултат true);

- Позицията в потока

**long Position** - връща текущата позиция в потока.

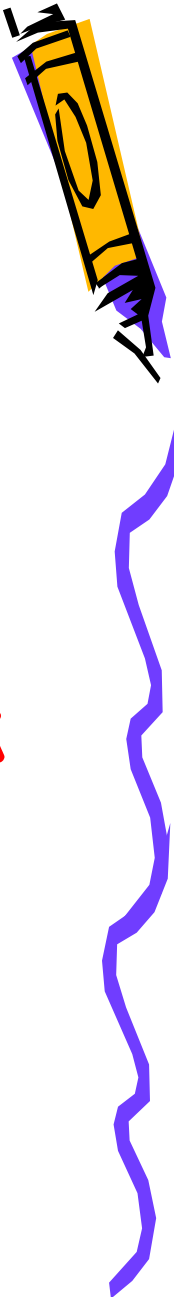
- дължината на потока

**long Length** - връща дължината на потока в байтове



# Пример:

- Ще коментираме основните методи `Read()`, `Write()`, `Seek()`, които, клас `FileStream` наследява от клас `Stream`.
- Ще започнем със **създаване на файлов поток.**



# Създаване на файлов поток

```
FileStream fs = new FileStream(string fileName,  
    FileMode [, FileAccess [, FileShare]]);
```

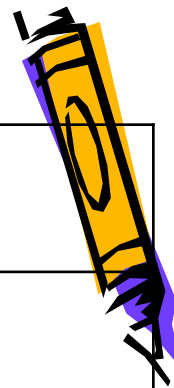
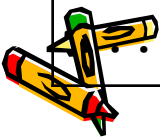
Пример 1: `FileStream fs = new FileStream("proba.txt",  
 FileMode.Open, FileAccess.Read, FileShare.Read);`

Пример 2:

```
FileStream fs = new FileStream("file.bin",  
    FileMode.Open, FileAccess.ReadWrite, FileShare.None);
```

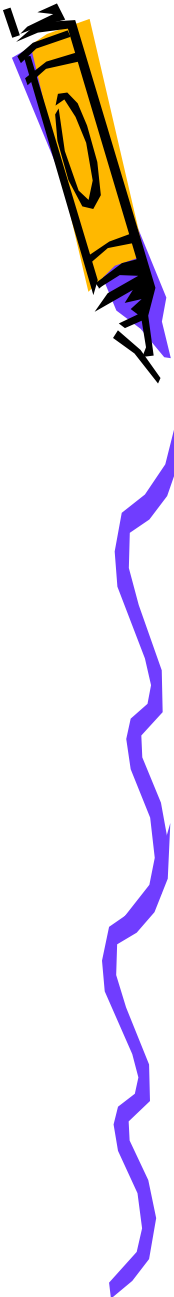
Пример 3:

```
public const string name = "test.dat";  
public const string OUTPUT_FILE = @"C:\nov.dat";  
static void Main()  
{  
    //Създаване на нов, празен файл:  
    if (File.Exists(name))  
    {  
        Console.WriteLine("The file {0} already exists!", name);  
    }  
    FileStream outFile = new FileStream(OUTPUT_FILE,  
        FileMode.Create)  
    FileStream ms = new FileStream(name, FileMode.Open,  
        FileAccess.Read);
```



При създаване на файлов поток посочваме:

- името на файла, с който свързваме потока (**fileName**),
- начина на отваряне на файла (**FileMode**),
- правата, с които го отваряме (**FileAccess**) и
- правата, които притежават другите потребители, докато ние държим файла отворен (**FileShare**).





**FileMode** може да има една от следните стойности:

- **Open** - отваря съществуващ файл.
- **Append** - отваря съществуващ файл и придвижва позицията веднага след края му.
- **Create** - създава нов файл. Ако файлът вече съществува, той се презаписва и старото му съдържание с губи.
- **CreateNew** - аналогично на **Create**, но ако файлът съществува, се хвърля изключение.
- **OpenOrCreate** - отваря файла, ако съществува, в противен случай го създава.
- **Truncate** - отваря съществуващ файл и изчиства съдържанието му, като прави дължината му 0 байта.
- **FileAccess** и **FileShare** могат да приемат стойности **Read**, **Write** и **ReadWrite**. **FileShare** може да бъде и **None**.



# Следва пример:

Четене и писане от поток с използване  
методите Read и Write на  
System.IO.Stream (т.е без използване  
четец и писач):



Пример:

```
public const string name = "test.dat";  
public const string OUTPUT_FILE = @"C:\nov.dat";  
    static void Main()  
    {  
        //Създаване на нов, празен файл за запис:  
        if (File.Exists(name))  
        {  
            Console.WriteLine("The file {0} already exists!", name);  
            FileStream outFile = new FileStream(OUTPUT_FILE,  
                FileMode.Create)
```

```
            FileStream ms = new FileStream(name, FileMode.Open,  
                FileAccess.Read);
```

```
            byte[] buf = new byte[4096];
```

```
            ...
```

```
            int bytesRead = ms.Read(buf, 0, buf.Length);
```

```
            if (bytesRead == 0) // входният файл вече е прочетен  
                break;
```

```
                иначе:
```

```
                outFile.Write(buf, 0, bytesRead);
```

```
                //!!!bytesRead байта записваме, а не buf.Length, защото реалният  
                брой на прочетените символи може да е по-малък от буфера
```



## Изчистване на работните буфери

**public abstract void Flush()**

- Методът **Flush()** изчиства буферите на физическите устройства, като изпраща съдържащите се в тях данни към механизма за съхранение (външния файл).
- След успешното приключване на изпълнението на **Flush()** е гарантирано, че всички данни, записани в потока, са изпратени към местоназначението си, но няма гаранция, че ще пристигнат успешно до него.
- Ако при писане в поток не се извиква **Flush()**, няма гаранция, че данните, записани в потока, са изпратени.

## Затваряне на поток

Методът **Close()** извиква **Flush()**, затваря текущия поток и освобождава използваните ресурси.

**public virtual void Close()**



- Ако по време на работа с отворен поток възникне някакво изключение, операцията `Close()` няма да се изпълни и потокът ще си остане отворен.
- Правилната работа с потоци изисква затварянето им да бъде гарантирано след приключване на работата с тях или чрез **using** конструкцията в **C#** или чрез употребата на **try-finally** блок.
- Когато използваме `using`, дефинираме програмния блок, в който е видим създавания обект.
- Ето как можем да използваме конструкцията `using` и безпроблемно да освободим потока:



```
StreamReader str = new...;
```

```
// Създава се обект от тип поток
```

```
using (str)
```

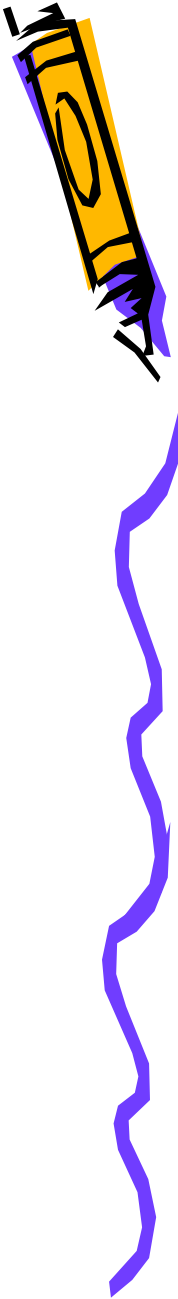
```
{ // Правим нещо със str ... }
```

```
// Потокът автоматично се затваря
```

- При достигане края на **using** блока, гарантирано се извиква методът `Dispose()` на посочения в клаузата обект (в примера обекта е **str**), а той (`Dispose()`) вътрешно извиква `Close()`.



```
StreamReader str = ...; //try-finally  
// Obtain opened stream  
try  
{  
    // Do something with the stream here  
}  
finally  
{  
    // Manually close the stream after finishing  
    working with it  
    str.Close();  
}
```



# Четци и писачи

- Четците и писачите (readers and writers) в .NET Framework са **класове**, които улесняват работата с потоците.
- Ако не използвате четец или писач (readers|writers), тоест при работа само с операциите на потока (в примерите поток FileStream), вие можете да четете и записвате **единствено байтове**, както видяхме в примерите по-горе.
- Когато, около FileStream поток се обвие в четец или писач, вече стават позволени четенето и записа на различни примитивни типове данни в двоичен вид (BinaryReader/BinaryWriter ) и текстова информация (TextReader/TextWriter).



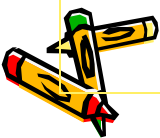


Забележка: в частност, клас `FileStream` е неудобен за работа с текстови файлове.

- За тази цел, в пространството `System.IO` са налице специални класове `StreamReader/StreamWriter`:
  - `StreamReader` - четец (наследник на абстрактния клас `TextReader`)

и

- `StreamWriter` - писач (наследник на абстрактния клас `TextWriter`) .



# Двоични и текстови Четци и писачи

Четците и писачите биват:

- Двоични: **BinaryReader/BinaryWriter** и
- Текстови: наследници на абстрактния клас **TextReader/TextWriter**).

Да ги разгледаме по-подробно:

- Двоичните четци и писачи (класове **BinaryReader и BinaryWriter**) осигуряват методи за четене и запис на примитивни типове данни в двоичен вид:



# Двоични четци и писачи - методи

ReadChar(), чете 1 символ и премества указателя с толкова байта, колкото заема символа.	Write(char),
ReadChars(), прочита масив от символи	Write(char[]),
ReadDecimal(), прочита decimal и премества указателя със 16 байта	Write(decimal)
ReadInt32(), прочита стойност int и премества указателя с 4 байта	Write(Int32),
ReadDouble(), прочита значение double и премества с 8 байта	Write(double)
ReadString(), прочита string.	Write(string)

пр.

# Двоични четци и писачи - пример

Пример - запис (**и четене**) на данни от различни типове:

```
using System;
using System.IO;
class Class1
{private const string name="d:\\test.dat";
    static void Main()
    {
FileStream ms = new FileStream(name, FileMode.Create,
    FileAccess.Write,FileShare.None);
/* FileStream ms = new FileStream(name, FileMode.Open, FileAccess.Read,
    FileShare.None); */
BinaryWriter br = new BinaryWriter(ms);
    //BinaryReader r = new BinaryReader(ms);
br.Write(123); //r.ReadInt32();
br.Write('A'); //r.ReadChar();
br.Write(12.3); //r.ReadDouble();
br.Write("Ivan Ivanov"); //r.ReadString();
br.Close();
    }
```



Абстрактен текстов четец (абстрактен базов клас за четене на текст): **TextReader** (и неговите наследници):

**TextReader**

->**StreamReader**

->**StringReader**

Абстрактен текстов писач (абстрактен базов клас за запис на текст):

**TextWriter** (и неговите наследници):

**TextWriter**

I----->**StreamWriter**

I----->**StringWriter**

- Текстовите четци и писачи осигуряват четене и запис на текстова информация, представена във вид на низове, разделени с нов ред.
- Тъй като класовете класове **TextReader** и **TextWriter** са абстрактни, за конкретни входно-изходни операции с текстови данни се използват техните посочени по-горе наследници.



Клас `StringWriter` – записва текстови данни като символен низ, а клас `StringReader` чете стринг от потока. Основните методи за четене и запис на `StreamReader` и `StreamWriter` са следните:

- `string ReadLine()` – прочита един ред текст.
- `string ReadToEnd()` – прочита всичко от текущата позиция до края на потока.
- `Write(...)` – вмъква данни в потока на текущата позиция.
- `WriteLine(...)` – вмъква данни в потока на текущата позиция и добавя символ за нов ред.

И др.



Пример:

```
private const string name = "nnn.txt";
static void Main()
{
    //Ако файлът съществува, то го прочитаме
    if (File.Exists(name))
    {
        Console.WriteLine("The file {0} already exists!", name); Console.ReadKey();
        //Създаваме Reader за файла
        StreamReader r = new StreamReader(name);
        //Четене на данните от файла
        string input;
        while ((input = r.ReadLine()) != null)
            Console.WriteLine(input);
        r.Close();
        Console.ReadKey(); return;
    }
    else //Ако файлът не съществува, то го създаваме
    {
        //Създаваме Writer за файла
        StreamWriter sw = File.CreateText(name);
        //Запис на данните във файла
        sw.WriteLine("{0},{1},{2}", "Ivan Ivanov", 1956, "Varna");
        //Затваряне на Writer и на FileStream
        sw.Close();
        Console.ReadKey();
    }
}
```



```
The file nnn.txt already exists!
Ivan Ivanov,1956,Varna
```

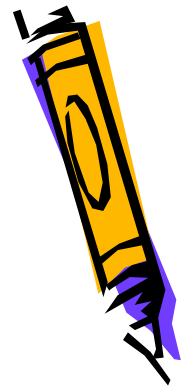
# Класове за работа с дисковете

- За представяне на дисковете, `System.IO` има клас **DriveInfo**.
- Този клас има статичен метод **GetDrives()**, който връща имената на всички логически дискове на вашия компютър.
- Клас **DriveInfo**, предоставя и редица полезни свойства:
  - **AvailableFreeSpace**: указва свободния обем в байтове
  - **DriveFormat**: получаваме името на файловата система





- **DriveType**: представя типа на диска (Fixed, Removable, CDRom)
- **IsReady**: готов ли е диска (например, DVD-диск може да не бъде поставен в дисковото устройство)
- **Name**: името на диска
- **TotalFreeSpace**: общия обем на диска в байтове
- **TotalSize**: общия размер на диска в байтове
- **VolumeLabel**: получава или поставя етикет на том

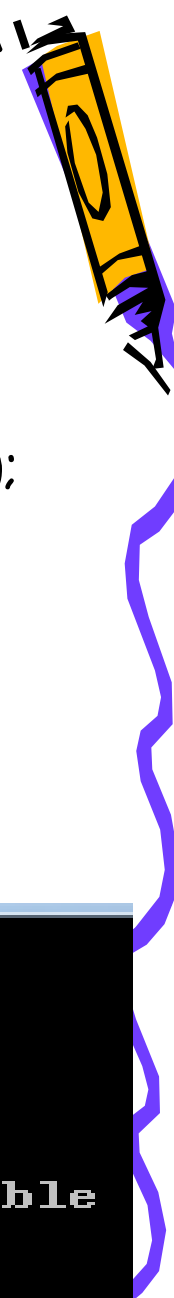


# Пример: да изведем имената и свойствата на ВСИЧКИ ДИСКОВЕ НА КОМПЮТЪРА:

using System.IO;...

static void Main()

```
{    DriveInfo[] drives = DriveInfo.GetDrives();
    foreach (DriveInfo drive in drives)
    { Console.WriteLine("Name of the drive: {0}", drive.Name);
      Console.WriteLine("Type of the drive: {0}", drive.DriveType);
      if (drive.IsReady)
      { Console.WriteLine("Volume: {0}", drive.TotalSize);
        Console.WriteLine("Free space: {0}", drive.TotalFreeSpace);
        Console.WriteLine("Label: {0}", drive.VolumeLabel);
      } Console.WriteLine();
    } Console.ReadLine();
}
```



```
Name of the drive: C:\
Type of the drive: Fixed
Volume: 159934050304
Free space: 104299753472
Label:

Name of the drive: D:\
Type of the drive: Removable
Volume: 4036231168
Free space: 663617536
Label:
```



# Класове за работа с файлове и директории

## 1. Работа с файлове

### Класове File и FileInfo (осъществяват операции с файлове)

- Класовете File и FileInfo са помощни класове за работа с файлове. Те дават възможност за стандартни операции върху файлове като създаване, изтриване, копиране, проверка за съществуване (каквато използвахме вече) и др.

В тях са дефинирани следните методи:

- Create(), CreateText() - създаване на файл, във втория случай - .txt.
- Пример 1. създаваме файл Foo.txt и го свързваме с FileStream обект

```
FileStream fs= File.CreateText("Foo.txt");
```

- Пример 2. създаваме файл Foo.txt и го свързваме с StreamWriter:

```
if (File.Exists("Foo.txt"))  
{StreamWriter writer= File.CreateText("Foo.txt");  
Writer.WriteLine("Ivan Ivanov");  
Writer.Close();}
```



- **Пример 3.** отваряме файл Foo.txt и връщаме StreamReader обект, прочитаем текста от файла и го извеждаме на конзолата

```
if (File.Exists("Foo.txt"))  
{StreamReader r= File.OpenText("Foo.txt");  
String input; //  
while ((input=r.ReadLine())!=null)  
{Console.WriteLine(input);}  
r.Close();  
}
```

- Open(), OpenRead(), OpenWrite(), AppendText() – отваряне на файл.
- CopyTo(...) – копиране на файл,  
`File.CopyTo("source.txt", "copye.txt");..`



- MoveTo(...) - преименуване на файл,  
`File.MoveTo("source.txt", "new.txt");`..
- Delete() - изтриване на файл,  
`File.Delete("source.txt");`...
- Exists(...) - проверка за съществуване, if  
`(File.Exists("test4.txt")) { ... }`
- LastAccessTime и LastWriteTime - момент на последен достъп и последен запис във файла,
- Length - свойство, връща long тип - дължината на файла, `File.Length`.
- CreationTime - свойство, връща DateTime, кога е създаден файла

В класа File, изброените методи са статични, а в класа FileInfo - достъпни чрез инстанция. Ако извършваме дадено действие еднократно (например създаваме един файл, след това го отваряме), класът File е за предпочитане. Работата с FileInfo и създаването на обект биха имали смисъл при многократното му използване.



static void Main() **//File и FileInfo - пример:**

```
{ StreamWriter writer = File.CreateText("test1.txt");
```

**//Създаваме файл test1.txt и го обвиваме с writer**

```
using (writer)
```

```
{ //чрез using безопасно използваме test1.txt в using блок  
  writer.WriteLine("Hello Students!");
```

**//гарантирано тук се извиква Dispose() на обекта writer от тип**

**// StreamWriter, който от своя страна вика Close().**

```
}
```

**//създаваме обект fileInfo от тип FileInfo и го свързваме с файл test1.txt**

```
FileInfo fileInfo = new FileInfo("test1.txt");
```

**//копиране на съдържанието на test1.txt в test2.txt и в test3.txt, като ако test2.txt и test3.txt съществуват, ще бъдат презаписани - true**

```
fileInfo.CopyTo("test2.txt", true);
```

```
fileInfo.CopyTo("test3.txt", true);
```

```
if (File.Exists("test4.txt"))
```

```
{ // проверка за съществуване на test4.txt
```

```
  File.Delete("test4.txt");           //изтриване на test4.txt
```

```
} File.Move("test3.txt", "test4.txt");
```

**//test3 става test4, тоест преименуване на test3.txt**

```
}
```



В примера, извикването на `CreateText(...)` създава файла `test1.txt` и отваря текстов писач върху него.

С този писач можем да запишем някакъв произволен текст във файла.

След създаването на `FileInfo` обект, копираме създадения файл в два други. Параметърът `true` означава, че при вече съществуващ файл `test2.txt`, респ. `test3.txt`, новият файл ще бъде записан върху стария.

След това се проверява дали съществува файл `test4.txt` и се изтрива,

Следва - `test3.txt` се преименува като `test4.txt`.



## 2. Работа с директории

### Класове Directory и DirectoryInfo

- Класовете Directory и DirectoryInfo са помощни класове за работа с директории. Ще изброим основните им методи, като отбележим, че за Directory те са статични, а за DirectoryInfo – достъпни чрез инстанция.
- Create(), CreateSubdirectory() – създава директория или поддиректория.
- GetFiles() – връща всички файлове от директорията.
- GetDirectories(...) – връща всички поддиректории на директорията.
- MoveTo(...) – премества (преименува) директория.
- Delete() – изтрива директория.
- Exists() – проверява директория дали съществува.
- Parent – връща горната директория.
- FullName – пълно име на директорията – свойство.
- CreationTime – Свойство, връща DateTime тип.





**Пример 1:** за използване на DirectoryInfo и FileInfo с цел да се извърши преглед на текущата директория и на \*.cs файловете в нея:

```
DirectoryInfo d=new DirectoryInfo(".");  
foreach (FileInfo fn in d.GetFiles("*.cs"))  
{
```

//d.GetFiles("\*.cs") - извличат се всички .cs файлове (обекти от тип FileInfo) на директория d

```
String name = fn.FullName;
```

// в обект name от тип string се запазва името на всеки файл

```
long size = fn.Length;
```

//свойство Length - връща дължината на файл

```
DateTime creation=fn.CreationTime;
```

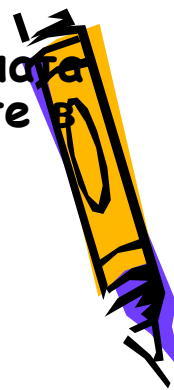
// creation - датата на създаване на файла

```
Console.WriteLine("Файлът {0} с размер {1} е създаден на {2}.", name, syze, creation);
```



Пример 2: Рекурсивно обхождане на директории: Ще разгледаме програма, която обхожда дадена директория и извежда на конзолата нейното съдържание, като рекурсивно обхожда и поддиректориите в нея:

```
using System; using System.IO;
class DirectoryTraversal
{   private static void Traverse(string aPath)
{Console.WriteLine("[{0}]", aPath);
string[] subdirs = Directory.GetDirectories(aPath);
/* Directory.GetDirectories() - ще върне всички поддиректории на
C:\WINDOWS\system32 в масив от стрингове subdirs */
foreach (string subdir in subdirs) {Traverse(subdir);}
//рекурсивно се извиква Traverse() за всички поддиректории
string[] files = Directory.GetFiles(aPath);
/*Directory.GetFiles() - ще върне всички файлове files на обработваната
директория, започвайки от C:\WINDOWS\system32) */
foreach (string f in files)
{Console.WriteLine(f);   } //всеки файл ще се разпечата на екрана
}
static void Main()
{   string winDir = Environment.SystemDirectory;
/*winDir ще съдържа SystemDirectory на класа Environment. Това е
директорията C:\WINDOWS\system32 */
Traverse(winDir); }
}
```



## Как работи примерът?

- Променливата `winDir` определя началната директория, от която започва обхождането. В случая, статичната член-променлива `SystemDirectory` на класа `Environment` определя директорията `C:\WINDOWS\system32` като начална.
- Началната директория предаваме като параметър на рекурсивния метод `Traverse(...)`, който извършва обхождане в дълбочина. Той извежда подадената му директория на екрана, след което се самоизвиква за всяка една нейна поддиректория.
- Поддиректориите на дадена директория се извличат с метода `Directory.GetDirectories(...)`, а файловете – с метода `Directory.GetFiles(...)`. И двата метода връщат като резултат масив от низове, съдържащи имена на директории или файлове, заедно с пълния път до тях.



## Класът Path

Класът `System.IO.Path` предоставя допълнителна функционалност за работа с пътища. Той обработва `string` променливи, съдържащи информация за пътя до файл или директория. Функционалността, предоставена от класа `Path` е независима от платформата.

Ето някои полезни свойства и методи на този клас:

- `DirectorySeparatorChar` – символът, който отделя директориите в пътя ("`\`" за Windows и "`/`" за UNIX и Linux файлови системи). (Свойство)
- `Combine(...)` – добавя относителен път към пълен.
- `GetExtension(...)` – извлича разширението на даден файл (ако има), връща `string`.
- `GetFileName(...)` – извлича име на файл от даден пълен път (ако има), връща `string`.

**`String File_Name = Path.GetFileName();`**

- `File_Name` ще съдържа името на файла, ако има такъв
- `GetTempFileName(...)` – създава временен файл с уникално име и нулева дължина, връща името му.



## (Работа с временни файлове - пример)

```
using System;
using System.IO;
class TempFilesDemo
{ static void Main()
  {String tempFileName = Path.GetTempFileName();
  //създава временен файл tempFileName
    try
    {
using (TextWriter writer = new StreamWriter(tempFileName))
  { writer.WriteLine("This is just a test");
  //във временния файл записваме "This is just a test"
  }
File.Copy(tempFileName, "test.txt");
//копираме съдържанието на временния файл в test.txt
  }
  finally
  {File.Delete(tempFileName);
  //изтриваме съдържанието на временния файл
  }
}
```



## Как работи примерът?

- Променливата `tempFileName` съдържа вече споменатия временен файл с уникално име.
- След като върху него отворим текстов писач и запишем текста "This is just a test", копираме временния файл в текстовия файл `test.txt`, който се създава в текущата директория (директорията `bin\Debug`) на приложението.
- След приключване на програмата, `test.txt` съдържа същия текст. Ако не изтрием временния файл във `finally` клаузата, можем да проверим, че върху хард диска остава новосъздаден файл с уникално име и разширение `.tmp`, който съдържа текста "This is just a test".



## Как работи примерът?

- Променливата `tempFileName` съдържа вече споменатия временен файл с уникално име.
- След като върху него отворим текстов писач и запишем текста "This is just a test", копираме временния файл в текстовия файл `test.txt`, който се създава в текущата директория (директорията `bin\Debug`) на приложението.
- След приключване на програмата, `test.txt` съдържа същия текст. Ако не изтрием временния файл във `finally` клаузата, можем да проверим, че върху хард диска остава новосъздаден файл с уникално име и разширение `.tmp`, който съдържа текста "This is just a test".



## Клас System.Environment

Клас System.Environment предоставя също полезни възможности (свойства и методи) за работа с файлове и директории, например:

**1. SystemDirectory** - свойство за системната директория

```
string winDir = System.Environment.SystemDirectory;
```

```
// winDir ще съдържа SystemDirectory на класа Environment
```

```
// това е директорията C:\WINDOWS\system32
```

**2. CurrentDirectory** - свойство за текущата директория

```
string currentDir = System.Environment.CurrentDirectory;
```

```
// currentDir - текущата директория
```

**3. Метода GetFolderPath()** на клас System.Environment осигурява за достъп до специални директории <special folders> на текущия потребител, тоест:

```
System.Environment.GetFolderPath(Environment.SpecialFolder.  
<special folder>), където:
```

<special folder> може да е:

Personal,

DesktopDirectory,

Favorites,

MyMusic

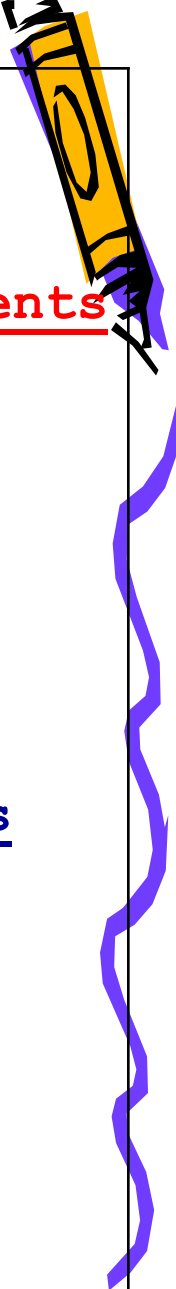
И др.





Ето и пример за достъп до някои от тях:

```
string myDocuments = Environment.GetFolderPath(  
    Environment.SpecialFolder.Personal);  
Console.WriteLine(myDocuments);  
// C:\Documents and Settings\Administrator\My Documents  
string myDesktop = Environment.GetFolderPath(  
    Environment.SpecialFolder.DesktopDirectory);  
Console.WriteLine(myDesktop);  
// C:\Documents and Settings\Administrator\Desktop  
string myFavourites = Environment.GetFolderPath(  
    Environment.SpecialFolder.Favorites);  
Console.WriteLine(myFavourites);  
// C:\Documents and Settings\Administrator\Favorites  
string myMusic = Environment.GetFolderPath(  
    Environment.SpecialFolder.MyMusic);  
Console.WriteLine(myMusic);  
// C:\Documents and Settings\Administrator\My  
Documents\My Music
```



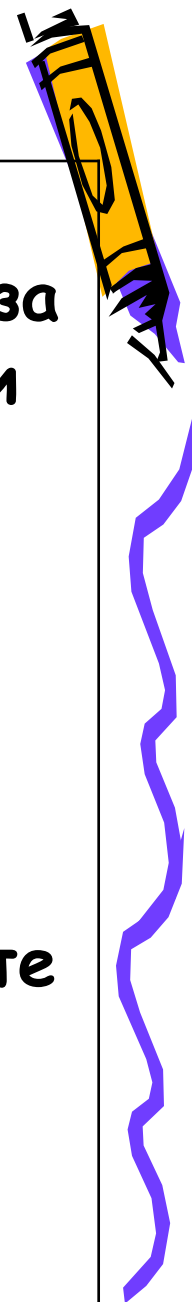
# Клас FileSystemWatcher

- Класът `FileSystemWatcher` в `System.IO` позволява наблюдение на файловата система за различни събития като създаване, промяна или преименуване на файл или директория.
- По-важните му събития и свойства са следните:

**Path** – съдържа наблюдаваната директория.

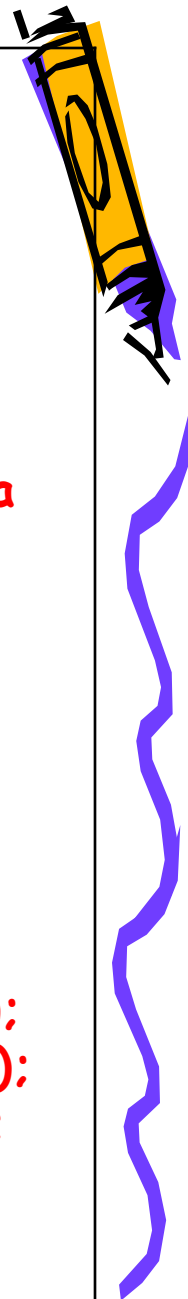
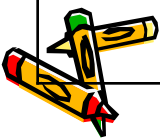
**Filter** – филтър за наблюдаваните файлове (напр. `"*.*)"` или `"*.exe"`).

**NotifyFilter** – филтър за типа на наблюдаваните събития, напр. `FileName`, `LastWrite`, `Size`, т.е. създаване, променя или преименуване на файлове...и др.



# Наблюдение на файловата система - пример

```
using System; using System.IO;
class FileSystemWatcherDemo
{
    static void Main()
    {
        string currentDir = Environment.CurrentDirectory;
        //това е bin\Debug директорията на проекта
        FileSystemWatcher w = new FileSystemWatcher(currentDir);
        //Watch all files - Следим всички файлове в bin\Debug директорията
        w.Filter = "*.*";
        // Watch the following information for the files
        //Следим за създаване, променя или преименуване на файлове
        w.NotifyFilter = NotifyFilters.FileName |
            NotifyFilters.DirectoryName |
            NotifyFilters.LastWrite;
        //Created, Changed, Renamed - събития, които се извикват при
        регистриране на създаване, променя или преименуване на файл.
        w.Created += new FileSystemEventHandler(OnCreated);
        w.Changed += new FileSystemEventHandler(OnChanged);
        w.Renamed += new RenamedEventHandler(OnRenamed);
        w.EnableRaisingEvents = true;
    }
}
```



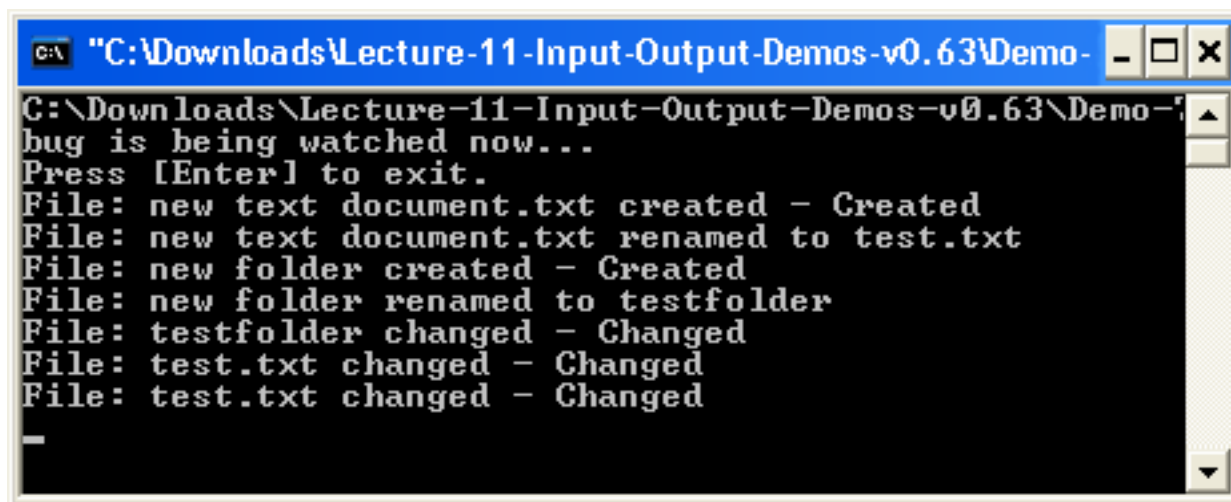
```
Console.WriteLine("{0} is being watched now...", currentDir);
Console.WriteLine("Press [Enter] to exit."); Console.ReadLine();
}
// Methods called when a file is created, changed, or renamed
//Метод, който се извиква, когато един файл се създава
private static void OnCreated(object aSource, FileSystemEventArgs aArgs)
{
    Console.WriteLine("File: {0} created - {1}", aArgs.Name, aArgs.ChangeType);
}
//Метод, който се извиква, когато един файл се променя
private static void OnChanged(object aSource, FileSystemEventArgs aArgs)
{
    Console.WriteLine("File: {0} changed - {1}", aArgs.Name,
aArgs.ChangeType);
}
//Метод, който се извиква, когато един файл се преименува
private static void OnRenamed(object aSource, RenamedEventArgs aArgs)
{
    Console.WriteLine("File: {0} renamed to {1}", aArgs.OldName, aArgs.Name);
}
}
```



Как работи примерът?

При конструирането на `FileSystemWatcher` обекта, му подаваме текущата директория – `bin\Debug` директорията на проекта. В тази директория ще наблюдаваме всички файлове, като ще следим изброените в `w.NotifyFilter` файлови операции. Събитията `Created`, `Changed` и `Renamed` свързваме с подходящи обработващи методи. Сега при създаване, модификация или преименуване на файл в наблюдаваната директория, се изпълнява съответният обработчик.

Ето примерен резултат от изпълнението на горната програма:



```
C:\Downloads\Lecture-11-Input-Output-Demos-v0.63\Demo-
bug is being watched now...
Press [Enter] to exit.
File: new text document.txt created - Created
File: new text document.txt renamed to test.txt
File: new folder created - Created
File: new folder renamed to testfolder
File: testfolder changed - Changed
File: test.txt changed - Changed
File: test.txt changed - Changed
-
```

Това изпълнение на програмата отразява създаването на текстов файл в наблюдаваната директория и неговото преименуване. Следва създаване и преименуване на поддиректория. Накрая е показано какво се случва при промяна на съдържанието на файл в наблюдаваната директория.

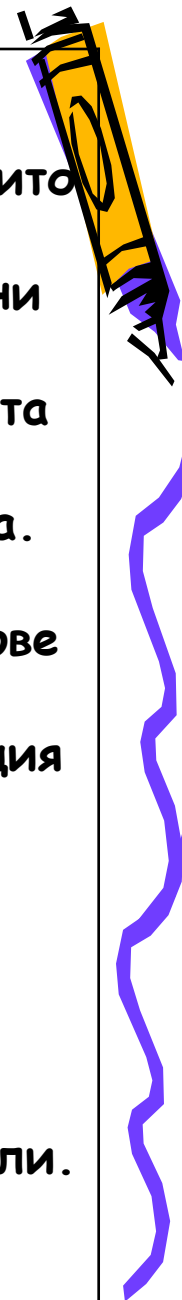
# Работа с IsolatedStorage

**IsolatedStorage** е технология, която се използва за приложения, които нямат достъп до локалния хард диск, но изискват локално съхраняване на файлове, напр. при работа с приложения, стартирани от Интернет, които по подразбиране работят с намалени права до локалния хард диск и не могат да осъществяват достъп до файловата система.

**IsolatedStorage** представлява изолирана виртуална файлова система. Тя е ограничена по обем (до 10 MB по подразбиране) и приложението, което я използва, няма достъп до останалите файлове на локалното устройство. Когато работи с **IsolatedStorage**, всяко приложение съхранява данни на място, уникално за него и за текущия потребител, а именно:

в директория `\Documents and Settings\<user>\Local Settings\Application Data\IsolatedStorage`.

Класовете за достъп до изолирани файлови системи се намират в пространството от имена `System.IO.IsolatedStorage`. Нивата на изолация са две – първо, името на потребителя, стартирал приложението, тоест `<user>` и второ – името на стартираното асембли. В много случаи, името на асемблито съответства на URL адреса, откъдето то е било заредено и стартирано.

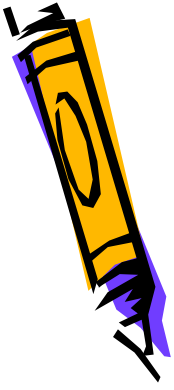


# Създаване и четене на компресирани файлове - `GZipStream` и `DeflateStream`

- .NET предоставя класове, които позволяват компресиране на файлове, а след това - възстановяване на файла от компресирано в изходно състояние.
- Това са класовете `DeflateStream` и `GZipStream` на `System.IO.Compression`.
- Те представляват реализация на алгоритми за компресиране, съответно `Deflate` или `GZip`.
- Да разгледаме използването на клас `GZipStream` с примера:

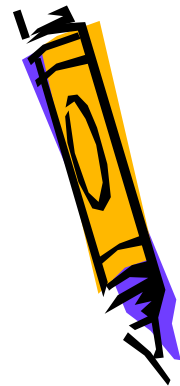


```
using System.IO;
using System.IO.Compression;
using System;
class Program
{
    static void Main(string[] args)
    {
        string sourceFile =
@"C:\Users\Vili\Desktop\book.pdf"; // изходен файл
        string compressedFile =
@"C:\Users\Vili\Desktop\book.gz"; // компресиран файл
        string targetFile =
@"C:\Users\Vili\Desktop\book_new.pdf"; // възстановен файл
        // създаване на компресиран файл
        Compress(sourceFile, compressedFile);
        // четене от компресирания файл
        Decompress(compressedFile, targetFile);
        Console.ReadLine();
    }
}
```





```
public static void Compress(string sourceFile, string
compressedFile)
    { // поток за четене на изходния файл
using (FileStream sourceStream = new FileStream(sourceFile,
FileStream.OpenOrCreate))
    { // поток за запис на компресирания файл
using (FileStream targetStream = File.Create(compressedFile))
    { // поток за архивация
using (GZipStream compressionStream = new
GZipStream(targetStream, CompressionMode.Compress))
    {
sourceStream.CopyTo(compressionStream);
        // копираме байтове от единия поток в другия
Console.WriteLine("The file {0} compressing finished. Initial
Size: {1} Compressed Size: {2}.", sourceFile,
sourceStream.Length.ToString(), targetStream.Length.ToString());
    }
    }
    }
    }
    }
```



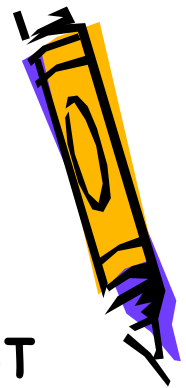


- Метод `Compress` получава името на изходния файл, който трябва да бъде компресиран и името на бъдещия компресиран файл.
- Отначало се създава поток за четене на изходния файл - `FileStream sourceStream`. След това се създава поток за запис в компресирания файл - `FileStream targetStream`.
- Потокът за архивация `GZipStream compressionStream` се инициализира с потока `targetStream` и с помощта на метода `CopyTo()` получава данните от потока `sourceStream`.

```
using (GZipStream compressionStream = new  
    GZipStream(targetStream,  
        CompressionMode.Compress))  
sourceStream.CopyTo(compressionStream);
```



- Метод `Decompress` извършва обратната операция по възстановяването на компресирания файл в изходно състояние.
- Той приема в качеството на параметри: път към компресирания файл и път към бъдещия възстановен файл.
- Сега, в началото се създава поток за четене от компресирания файл `FileStream sourceStream`, след което поток за запис в възстановения файл `FileStream targetStream`. В края се създава поток `GZipStream decompressionStream`, който с помощта на метода `CopyTo()` копира възстановените данни в поток `targetStream`.



- Използва се `Compress` и `Decompress` за параметър `CompressionMode` в конструктора на потока `GZipStream`, за да се укаже именно за какво е предназначен този поток – компресия или декомпресия.
- Ако желаем друг алгоритъм за компресиране (клас `DeflateStream`), то просто може да заменим в кода `GZipStream` със `DeflateStream`, без изменение на останалия код.

**Забележка:** Чрез използване на тези класове може да компресираме само един файл. За архивация на група файлове се използват други инструменти.

