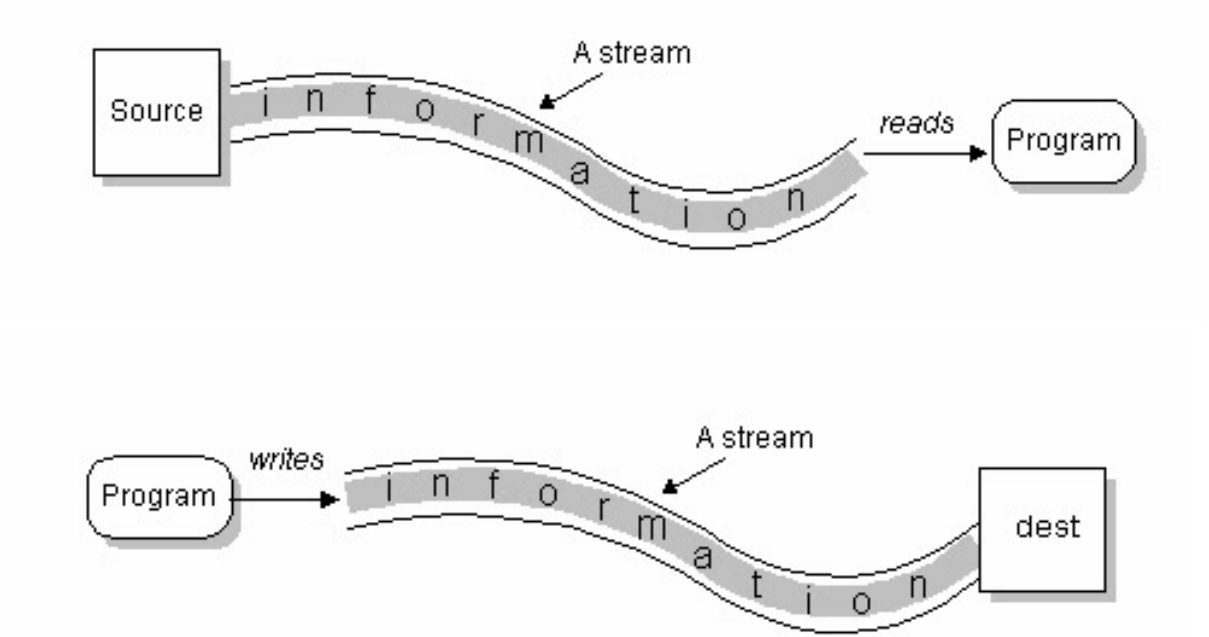
Программирование C#

6. Средства ввода-вывода. Файлы и потоки

Карбаев Д.С., 2015

Система ввода- вывода в C# на потоках

- **Поток (Stream)** — это некая абстракция производства или потребления информации.
- **Байтовые и символьные потоки**: в среде .NET Framework определено несколько классов, выполняющих превращение байтового потока в символьный с автоматическим преобразованием типа byte в тип char и обратно.



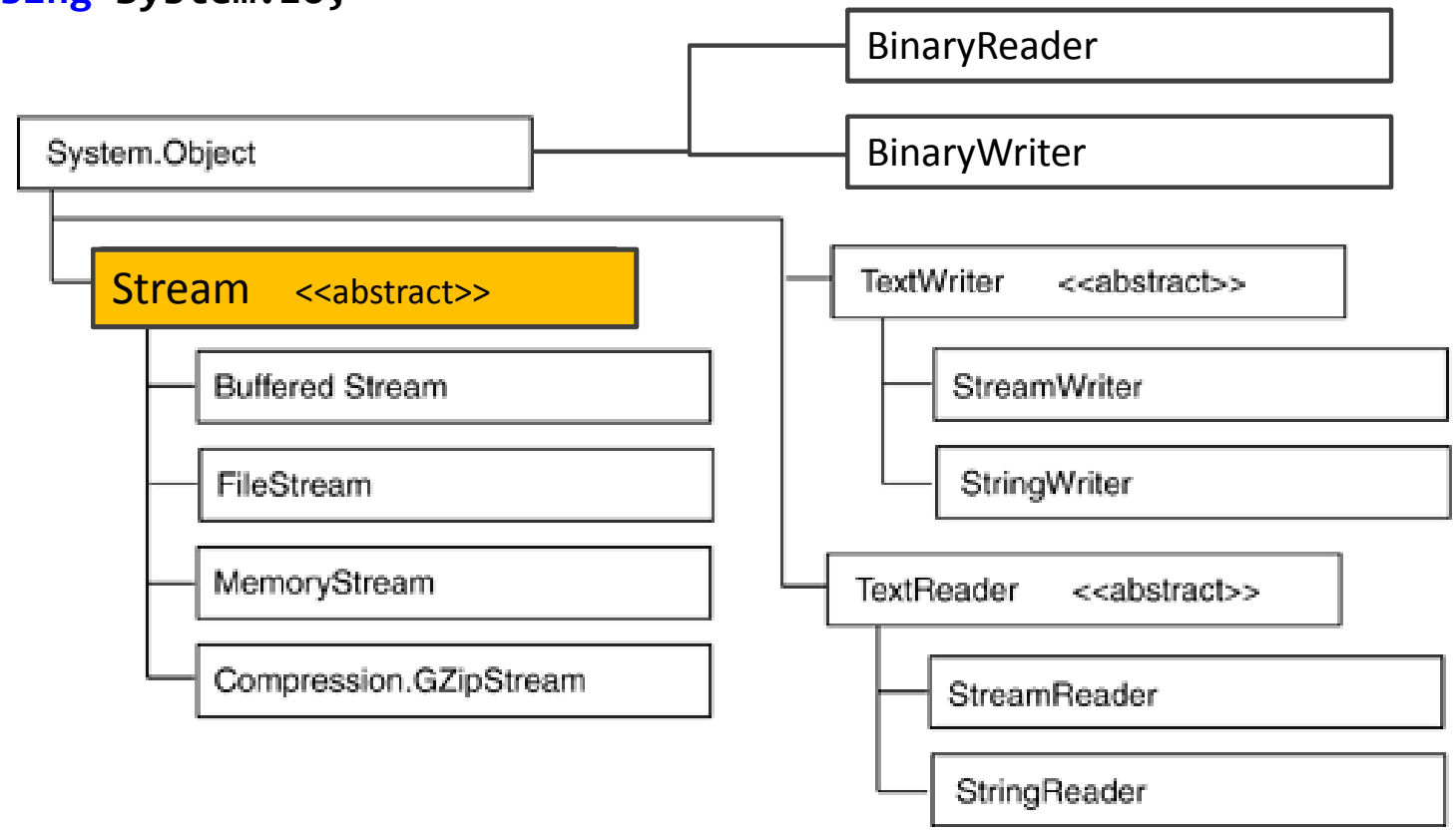
Система ввода- вывода в C# на потоках

- **Встроенные потоки**: доступны встроенные потоки, открывающиеся с помощью свойств Console.In, Console.Out и Console.Error.
  - **Свойство Console.Out - поток вывода на консоль (по умолчанию).**
  - **Свойство Console.In связано со стандартным потоком ввода, который по умолчанию осуществляется с клавиатуры.**
  - **свойство Console.Error связано со стандартным потоком сообщений об ошибках, которые по умолчанию также выводятся на консоль.**
- Но эти потоки могут быть переадресованы на любое другое совместимое устройство ввода-вывода. Стандартные потоки являются символьными.

Классы потоков

- Символьные потоки основываются на байтовых, хотя они и разделены логически.
- Основные классы потоков определены в пространстве имен System.IO:

```
using System.IO;
```



Классы потоков: класс Stream

- Основным для потоков является **класс System.IO.Stream** - представляет байтовый поток и является базовым для всех остальных классов потоков.
- **Таблица 1. Некоторые методы, определенные в классе Stream**

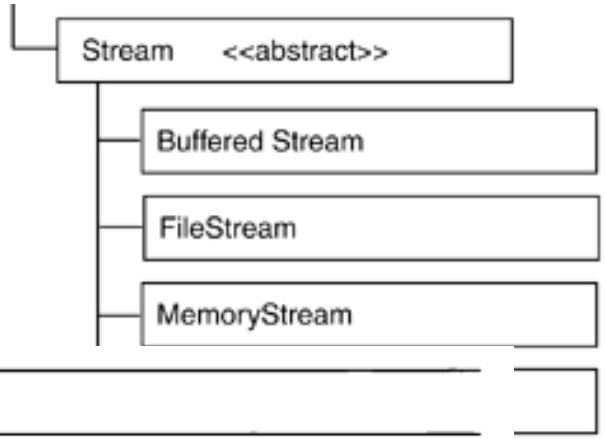
Метод	Описание
`void Close()`	Закрывает поток
`void Flush()`	Выводит содержимое потока на физическое устройство
`int ReadByte()`	Возвращает целочисленное представление следующего байта, доступного для ввода из потока. При обнаружении конца файла возвращает значение –1
`int Read(byte[] buffer, int offset, int count)`	Делает попытку ввести `count` байтов в массив `buffer`, начиная с элемента `buffer[offset]`. Возвращает количество успешно введенных байтов
`long Seek(long offset, SeekOrigin origin)`	Устанавливает текущее положение в потоке по указанному смещению `offset` относительно заданного начала отсчета `origin`. Возвращает новое положение в потоке
`void WriteByte(byte value)`	Выводит один байт в поток вывода
`void Write(byte[]buffer, int offset, int count)`	Выводит подмножество `count` байтов из массива `buffer`, начиная с элемента `buffer[offset]`. Возвращает количество выведенных байтов

Классы потоков: класс Stream

- **Таблица 2. Свойства, определенные в классе Stream**

Свойство	Описание
`bool CanRead`	Принимает значение `true`, если из потока можно ввести данные. Доступно только для чтения
`bool CanSeek`	Принимает значение `true`, если поток поддерживает запрос текущего положения в потоке. Доступно только для чтения
`bool CanWrite`	Принимает значение `true`, если в поток можно вывести данные. Доступно только для чтения
`long Length`	Содержит длину потока. Доступно только для чтения
`long Position`	Представляет текущее положение в потоке. Доступно как для чтения, так и для записи
`int ReadTimeout`	Представляет продолжительность времени ожидания в операциях ввода. Доступно как для чтения, так и для записи
`int WriteTimeout`	Представляет продолжительность времени ожидания в операциях вывода. Доступно как для чтения, так и для записи

Классы потоков



- Классы **байтовых потоков**

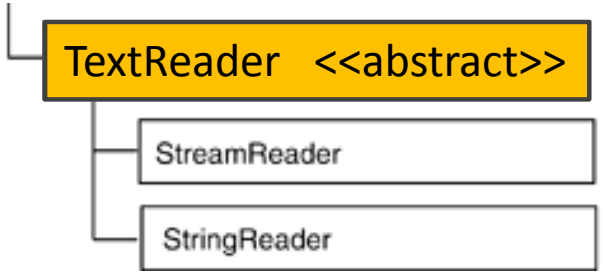
Класс потока	Описание
BufferedStream	Заключает в оболочку байтовый поток и добавляет буферизацию. Буферизация, как правило, повышает производительность
FileStream	Байтовый поток, предназначенный для файлового ввода-вывода
MemoryStream	Байтовый поток, использующий память для хранения данных
UnmanagedMemoryStream	Байтовый поток, использующий неуправляемую память для хранения данных

- **Классы-оболочки** символьных потоков: на вершине иерархии классов символьных потоков находятся абстрактные классы **TextReader** (ввод) и **TextWriter** (вывод).
- В классе TextReader имеется также метод **Close()**, определяемый следующим образом:

```
void Close ();
```

- этот метод закрывает считывающий поток и освобождает его ресурсы

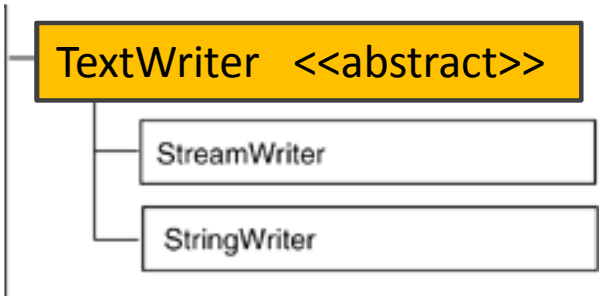
Классы потоков



- **Таблица 3. Методы ввода, определенные в классе TextReader**

Метод	Описание
`int Peek()`	Получает следующий символ из потока ввода, но не удаляет его. Возвращает значение -1, если ни один из символов не доступен
`int Read()`	Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца потока возвращает значение -1
`int Read(char[]buffer, int index, int count)`	Делает попытку ввести количество *count* символов в массив *buffer*, начиная с элемента *buffer[index]*, и возвращает количество успешно введенных символов
`int ReadBlock(char[]buffer, int index, int count)`	Делает попытку ввести количество *count* символов в массив *buffer*, начиная с элемента *buffer[index]*, и возвращает количество успешно введенных символов
`string ReadLine()`	Вводит следующую текстовую строку и возвращает ее в виде объекта типа `string`. При попытке прочитать признак конца файла возвращает пустое значение
`string ReadToEnd()`	Вводит все символы, оставшиеся в потоке, и возвращает их в виде объекта типа `string`

Классы потоков



- **Методы вывода, определенные в классе TextWriter**

Метод	Описание
`void Write(int value)`	Выводит значение типа `int`
`void Write(double value)`	Выводит значение типа `double`
`void Write(bool value)`	Выводит значение типа `bool`
`void WriteLine(string value)`	Выводит значение типа `string` с последующим символом новой строки
`void WriteLine(uint value)`	Выводит значение типа `uint` с последующим символом новой строки
`void WriteLine(char value)`	Выводит символ с последующим символом новой строки

- в классе TextWriter определены методы **Close()** и **Flush()**:

```
virtual void Close();
```

- закрывает записывающий поток и освобождает его ресурсы

```
virtual void Flush();
```

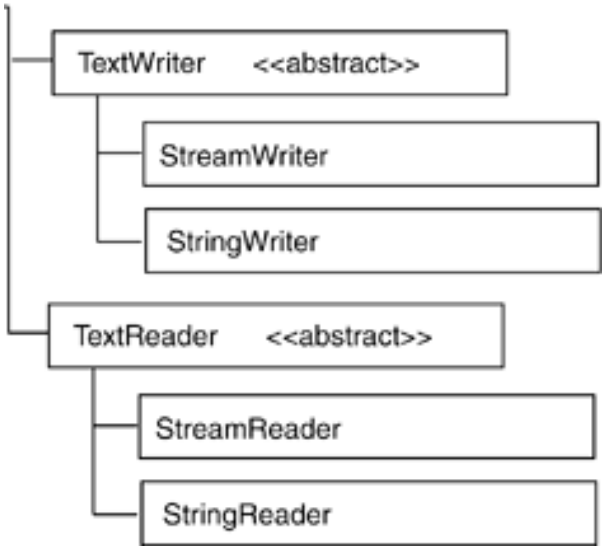
- организует вывод в физическую среду всех данных, оставшихся в выходном буфере

Классы потоков

- Классы **TextReader** и **TextWriter** реализуются несколькими классами символьных потоков, включая и те, что перечислены ниже.

Класс потока	Описание
`StreamReader`	Предназначен для ввода символов из байтового потока. Этот класс является оболочкой для байтового потока ввода
`StreamWriter`	Предназначен для вывода символов в байтовый поток. Этот класс является оболочкой для байтового потока вывода
`StringReader`	Предназначен для ввода символов из символьной строки
`StringWriter`	Предназначен для вывода символов в символьную строку

- Следовательно, в этих классах потоков предоставляются методы и свойства, определенные в классах TextReader и TextWriter



Консольный ввод-вывод

- Осуществляется с помощью стандартных потоков, представленных свойствами Console.In, Console.Out и Console.Error
- **Чтение данных из потока ввода с консоли:**
- Поток Console.In является экземпляром объекта класса TextReader, и поэтому для доступа к нему могут быть использованы методы и свойства, определенные в классе TextReader.
- В классе Console определены три метода ввода: Read(), ReadLine(), ReadKey().

```
static int Read();
```

- Метод Read () возвращает очередной символ, считанный с консоли. Он ожидает до тех пор, пока пользователь не нажмет клавишу, а затем возвращает результат. Возвращаемый символ относится к типу int и поэтому должен быть приведен к типу char. Если при вводе возникает ошибка, то метод Read () возвращает значение -1.

Консольный ввод-вывод

```
// Считать символ, введенный с клавиатуры.
using System;
class ReadChar {
    static void Main() {
        char ch;
        Console.Write("Введите символ: ");
        ch = (char) Console.Read(); // получить значение типа char
        Console.WriteLine("Символ: " + ch);
    }
}
```

- Результат

```
Введите символ: t
Символ: t
```

Консольный ввод-вывод

- Для, считывания строки символов служит приведенный ниже метод ReadLine().

```
static string ReadLine();
```

- Символы считываются методом ReadLine () до тех пор, пока пользователь не нажмет клавишу <Enter>, а затем этот метод возвращает введенные символы в виде объекта типа string

```
// Ввод с консоли с помощью метода ReadLine().
using System;

class ReadString {
    static void Main() {
        string str;
        Console.WriteLine("Введите несколько символов.");
        str = Console.ReadLine();
        Console.WriteLine("Вы ввели: " + str);
    }
}
```

Консольный ввод-вывод

- Для чтения данных можно обратиться и к методам базового класса TextReader

```
// Прочитать введенную строку непосредственно из потока Console.In.
using System;
class ReadChars2 {
 static void Main() {
   string str;
   Console.WriteLine("Введите несколько символов.");
   str = Console.In.ReadLine(); //вызвать метод ReadLine() класса TextReader
   Console.WriteLine("Вы ввели: " + str);
 }
}
```

Консольный ввод-вывод

- **Применение метода ReadKey ():** две формы объявления метода ReadKey().

```
static ConsoleKeyInfo ReadKey(); //возвращает введенный с клавиатуры символ и выводит его на экран
static ConsoleKeyInfo ReadKey(bool intercept); //если значение параметра intercept равно true, то введенный символ не отображается; если значение параметра intercept равно false, то введенный символ отображается
```

- Метод **ReadKey ()** возвращает информацию о нажатии клавиши в объекте типа ConsoleKeyInfo, который представляет собой структуру, состоящую из приведенных ниже свойств, доступных только для чтения.

```
char KeyChar;
ConsoleKey Key;
ConsoleModifiers Modifiers;
```

- Свойство **KeyChar** содержит эквивалент char введенного с клавиатуры символа, свойство Key — значение из перечисления **ConsoleKey** всех клавиш на клавиатуре, а свойство **Modifiers** — описание одной из модифицирующих клавиш (<Alt>, <Ctrl> или <Shift>), которые были нажаты.
- Эти модифицирующие клавиши представлены в перечислении **ConsoleModifiers** следующими значениями: Control, Shift и Alt.

Консольный ввод-вывод

```
// Считать символы, введенные с консоли, используя метод
//ReadKey().
ConsoleKeyInfo keypress;
Console.WriteLine("Введите несколько символов, " +
"a по окончании - <Q>.");
do{
  // считать данные о нажатых клавишах
  keypress = Console.ReadKey();
  Console.WriteLine(" Вы нажали клавишу: " + keypress.KeyChar);
  // Проверить нажатие модифицирующих клавиш.
  if ((ConsoleModifiers.Alt & keypress.Modifiers) != 0)
  Console.WriteLine("Нажата клавиша <Alt>.");
  if ((ConsoleModifiers.Control & keypress.Modifiers) != 0)
  Console.WriteLine("Нажата клавиша <Control>.");
  if ((ConsoleModifiers.Shift & keypress.Modifiers) != 0)
  Console.WriteLine("Нажата клавиша <Shift>.");
} while (keypress.KeyChar != 'Q');
```

Результат:

```
Введите несколько
символов, a по
окончании - <Q>.
a Вы нажали клавишу: a
b Вы нажали клавишу: b
d Вы нажали клавишу: d
A Вы нажали клавишу: A
Нажата клавиша <Shift>.
B Вы нажали клавишу: B
Нажата клавиша <Shift>.
C Вы нажали клавишу: C
Нажата клавиша <Shift>.
Нажата клавиша
<Control>.
Q Вы нажали клавишу: Q
Нажата клавиша <Shift>.
```

Запись данных в поток вывода на консоль

```
   // Организовать вывод в потоки Console.Out и
Console.Error.
   int a = 10, b = 0;
   int results;
   Console.Out.WriteLine("Деление на ноль приведет "
   + "к исключительной ситуации.");
   try
   {
     result = a / b; // сгенерировать исключение при
попытке деления на ноль
   }
   catch (DivideByZeroException exc)
   {
     Console.Error.WriteLine(exc.Message);
   }
```

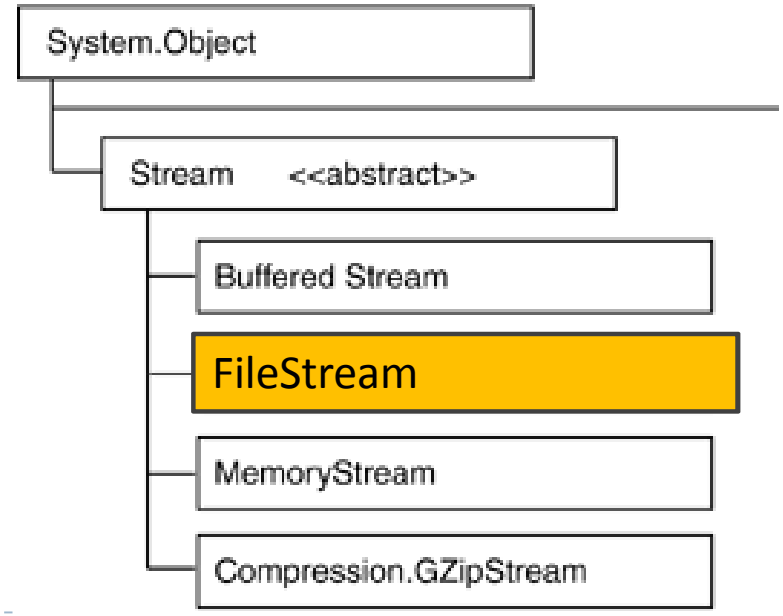
Результат:

```
Деление на ноль
приведет к
исключительной
ситуации.
Попытка деления на
ноль.
```

Класс FileStream

- Для создания байтового потока, привязанного к файлу, служит класс **FileStream**. Этот класс является производным от класса Stream и наследует всего его функции.
- Классы потоков, в том числе и FileStream, определены в пространстве имен System.IO. Поэтому в самом начале любой использующей их программы обычно вводится следующая строка кода.

```
using System.IO;
```



Класс FileStream

- Для формирования байтового потока, привязанного к файлу, создается объект класса FileStream:

```
FileStream(string путь, FileMode режим);
```

- где путь обозначает имя открываемого файла, включая полный путь к нему; а режим — порядок открытия файла.
- Как правило, этот конструктор открывает файл для доступа с целью чтения или записи. Исключением из этого правила служит открытие файла в режиме **FileMode.Append**, когда файл становится доступным только для записи.

Класс FileStream

- **Таблица 4. Значения из перечисления FileMode**

Значение	Описание
FileMode.Append	Добавляет выводимые данные в конец файла
FileMode.Create	Создает новый выходной файл. Существующий файл с таким же именем будет разрушен
FileMode.CreateNew	Создает новый выходной файл. Файл с таким же именем не должен существовать
FileMode.Open	Открывает существующий файл
FileMode.OpenOrCreate	Открывает файл, если он существует. В противном случае создает новый файл
FileMode.Truncate	Открывает существующий файл, но сокращает его длину до нуля

Класс FileStream

- Если попытка открыть файл оказывается неудачной, то генерируется исключение. Если же файл нельзя открыть из-за того что он не существует, генерируется исключение **FileNotFoundException**. А если файл нельзя открыть из-за какой-нибудь ошибки ввода-вывода, то генерируется исключение **IOException**.
- К числу других исключений, которые могут быть сгенерированы при открытии файла, относятся следующие:
  - **ArgumentNullException** (указано пустое имя файла),
  - **ArgumentException** (указано неверное имя файла),
  - **ArgumentOutOfRangeException** (указан неверный режим),
  - **SecurityException** (у пользователя нет прав доступа к файлу),
  - **PathTooLongException** (слишком длинное имя файла или путь к нему),
  - **NotSupportedException** (в имени файла указано устройство, которое не поддерживается),
  - **DirectoryNotFoundException** (указан неверный каталог).
- Исключения PathTooLongException, DirectoryNotFoundException и FileNotFoundException относятся к подклассам класса исключений IOException. Поэтому все они могут быть перехвачены, если перехватывается исключение IOException.

Класс FileStream

Один из способов открытия файла test.dat для ввода:

```
FileStream fin;
try {
    fin = new FileStream("test", FileMode.Open);
}
// перехватить все исключения, связанные с вводом-выводом
catch (IOException exc) {
    Console.WriteLine(exc.Message);
    // Обработать ошибку.
}
// перехватить любое другое исключение.
catch (Exception exc) {
    Console.WriteLine(exc.Message);
    // Обработать ошибку, если это возможно.
    // Еще раз сгенерировать необрабатываемые исключения.
}
```

Класс FileStream

- Если требуется ограничить доступ к файлу только для чтения или же только для записи, то в таком случае следует использовать такой конструктор:

```
FileStream(string путь, FileMode режим, FileAccess доступ)
```

- путь обозначает имя открываемого файла, включая и полный путь к нему, а режим — порядок открытия файла. В то же время доступ обозначает конкретный способ доступа к файлу. В последнем случае указывается одно из значений, определяемых в перечислении FileAccess и приведенных ниже.
  - **FileAccess.Read**
  - **FileAccess.Write**
  - **FileAccess.ReadWrite**

Класс FileStream

- Например, файл test. dat открывается только для чтения:

```
FileStream fin = new FileStream("test.dat", FileMode.Open,
FileAccess.Read);
```

- По завершении работы с файлом его следует закрыть, вызвав метод:

```
void Close();
```

- При закрытии файла высвобождаются системные ресурсы, распределенные для этого файла, что дает возможность использовать их для другого файла.
- Оператор using предоставляет еще один способ закрытия файла, который больше не нужен.

Чтение байтов из FileStream

- В классе **FileStream** определены два метода для чтения байтов из файла: ReadByte () и Read (). Так, для чтения одного байта из файла используется метод ReadByte ():

```
int ReadByte(); //из файла считывается один байт, который затем возвращается в виде целого значения
```

- Для чтения блока байтов из файла служит метод Read (), общая форма которого выглядит так.

```
int Read(byte[] array, int offset, int count);
```

- В методе Read () предпринимается попытка считать количество count байтов в массив array, начиная с элемента array[offset]. Он возвращает количество байтов, успешно считанных из файла.
- Если же возникает ошибка ввода-вывода, то генерируется исключение IOException.
- К числу других вероятных исключений, которые генерируются при этом, относится **NotSupportedException**. Это исключение генерируется в том случае, если чтение из файла не поддерживается в потоке

```
/* Отобразить содержимое текстового файла. Например, для просмотра содержимого файла
TEST.CS введите в командной строке следующее: ShowFile TEST.CS */
static void Main(string[] args) {
        FileStream fin; int i;
        if (args.Length != 1) {
           Console.WriteLine("Применение: ShowFile Файл");
           return;
        }
        try{ fin = new FileStream(args[0], FileMode.Open);  }
        catch (IOException exc) {
           Console.WriteLine("Не удается открыть файл"); Console.WriteLine(exc.Message);
            return; // Файл не открывается, завершить программу
        }
        // Читать байты до конца файла
        try{ do{
                i = fin.ReadByte();
                if (i != -1) Console.Write((char)i);
             } while (i != -1);
        }
        catch (IOException exc) {
           Console.WriteLine("Ошибка чтения файла"); Console.WriteLine(exc.Message);
        }
        finally { fin.Close(); }
    }
```

```
// Отобразить содержимое текстового файла. Краткий вариант
static void Main(string[] args) {
        FileStream fin = null; int i;
        if (args.Length != 1) {
            Console.WriteLine("Применение: ShowFile File");
            return;
        }
        // Использовать один блок try для открытия файла и чтения из него
        try {
            fin = new FileStream(args[0], FileMode.Open);
            // Читать байты до конца файла,
            do {
                i = fin.ReadByte();
                if (i != -1) Console.Write((char)i);
            } while (i != -1);
        }
        catch (IOException exc) {
            Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
        }
        finally {
            if (fin != null) fin.Close();
        }
    }
```

Запись в файл

- Для записи байта в файл служит метод **WriteByte ()**:

```
void WriteByte(byte value);
```

- Этот метод выполняет запись в файл байта, обозначаемого параметром **value**.
- Если базовый поток не открывается для вывода, то генерируется исключение **NotSupportedException**. А если поток закрыт, то генерируется исключение **ObjectDisposedException**.
- Для записи в файл целого массива байтов может быть вызван метод **Write()**:

```
void Write(byte[] array, int offset, int count);
```

- В методе Write () предпринимается попытка записать в файл количество count байтов из массива array, начиная с элемента array [offset].
  Он возвращает количество байтов, успешно записанных в файл.
- Если во время записи возникает ошибка, то генерируется исключение **IOException**. А если базовый поток не открывается для вывода, то генерируется исключение **NotSupportedException**.
  Кроме того, может быть сгенерирован ряд других исключений.

Запись в файл

- Если данные требуется записать на физическое устройство без предварительного накопления в буфере, то для этой цели можно вызвать метод:

```
void Flush();
```

- При неудачном исходе данной операции генерируется исключение **IOException**. Если же поток закрыт, то генерируется исключение **ObjectDisposedException**.
- По завершении вывода в файл следует закрыть его с помощью метода **Close ()**. Этим гарантируется, что любые выведенные данные, оставшиеся в дисковом буфере, будут записаны на диск. В этом случае отпадает необходимость вызывать метод Flush () перед закрытием файла.

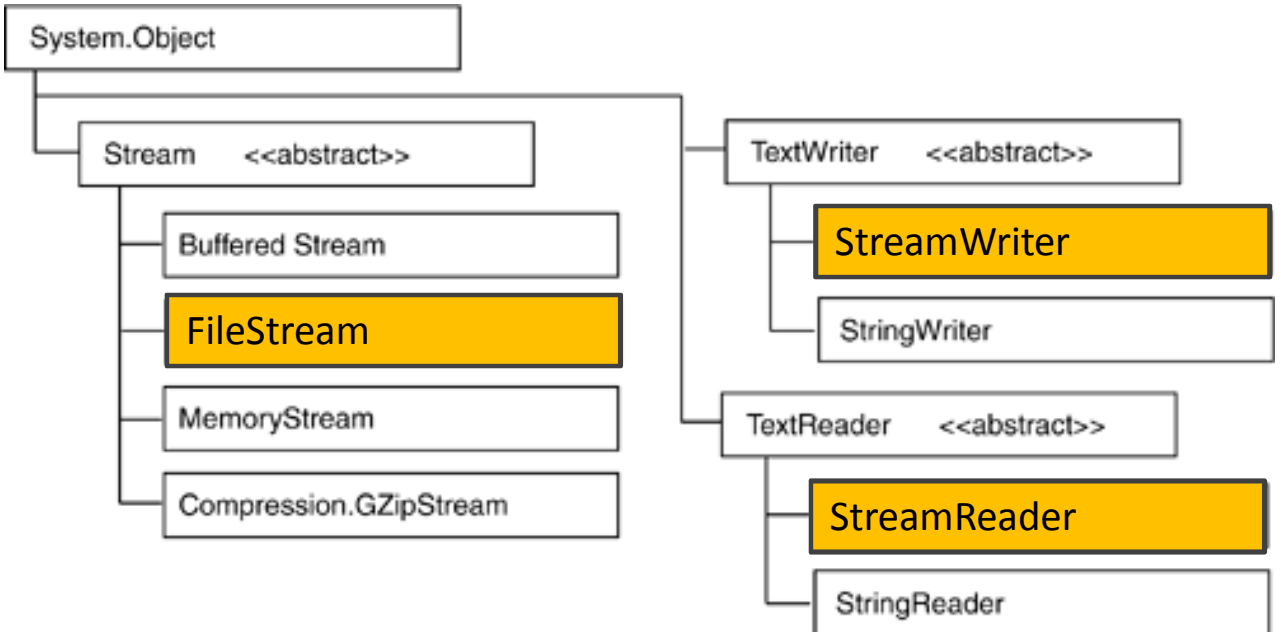
```
// Записать данные в файл.
using System;
using System.IO;
class WriteToFile {
    static void Main(string[] args){
        FileStream fout = null;
        try{
            // Открыть выходной файл.
            fout = new FileStream("test.txt", FileMode.CreateNew);
            // Записать весь английский алфавит в файл.
            for (char c = 'A'; c <= 'Z'; C++)
                fout.WriteByte((byte)c);
        }
        catch (IOException exc)
        {
            Console.WriteLine("Ошибка ввода-вывода: \n" + exc.Message);
        }
        finally
        {
            if (fout != null) fout.Close();
        }
    }
}
```

Результат (test.txt):
ABCDEFGHIJKLMNOPQRSTUVWXYZ

```
  /* Копировать файл. Например, для копирования файла FIRST.DAT в файл
SECOND.DAT введите: CopyFile FIRST.DAT SECOND.DAT */
  static void Main(string[] args){
          int i;
          FileStream fin = null;
          FileStream fout = null;
          if (args.Length != 2){
              Console.WriteLine("Применение: CopyFile Откуда Куда");
              return;
          }
          try{
              // Открыть файлы.
              fin = new FileStream(args[0], FileMode.Open);
              fout = new FileStream(args[1], FileMode.Create);
              // Скопировать файл,
              do{
                  i = fin.ReadByte();
                  if (i != -1) fout.WriteByte((byte)i);
              } while (i != -1);
          }
          catch (IOException exc){
              Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
          }
          finally{
              if (fin != null) fin.Close();
              if (fout != null) fout.Close();
          }
      }
```

Символьный ввод-вывод в файл

- Для выполнения операций символьного ввода-вывода в файлы объект класса **FileStream** заключается в оболочку класса **StreamReader** или **StreamWriter**. В этих классах выполняется автоматическое преобразование байтового потока в символьный и наоборот.
- Класс **StreamWriter** является производным от класса **TextWriter**, а класс **StreamReader** — производным от класса **TextReader**. Следовательно, в классах StreamReader и StreamWriter доступны методы и свойства, определенные в их базовых классах.



Символьный ввод-вывод в файл

- Для создания символьного потока вывода достаточно заключить объект класса **Stream**, например **FileStream**, в оболочку класса **StreamWriter**:

```
StreamWriter(Stream поток)
```

где поток обозначает имя открытого потока.

- Этот конструктор генерирует исключение **ArgumentException**, если поток не открыт для вывода, а также исключение **ArgumentNullException**, если поток оказывается пустым.
- После создания объекта класс StreamWriter выполняет автоматическое преобразование символов в байты.

```
// Программа ввода с клавиатуры и вывода на диск (StreamWriter)
string str; FileStream fout;
// Открыть сначала поток файлового ввода-вывода
try {
  fout = new FileStream("test.txt", FileMode.Create);
}
catch (IOException exc) {
  Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
  return;
}
// Заключить поток файлового ввода-вывода в оболочку класса StreamWriter
StreamWriter fstr_out = new StreamWriter(fout);
try {
  Console.WriteLine("Введите текст, а по окончании – 'стоп'.");
  do {
    Console.Write(": ");
    str = Console.ReadLine();
    if (str != "стоп") {
      str = str + "\r\n"; // добавить новую строку
      fstr_out.Write(str);
    }
  } while (str != "стоп");
}
catch (IOException exc) {
  Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
}
finally{  fstr_out.Close(); }
```

Применение класса StreamWriter

- В некоторых случаях файл удобнее открывать средствами самого класса StreamWriter:

```
StreamWriter(string путь);
StreamWriter(string путь, bool append);
```

где путь — это имя открываемого файла, включая полный путь к нему.

- Если во второй форме этого конструктора значение параметра append равно true, то выводимые данные присоединяются в конец существующего файла. В противном случае эти данные перезаписывают содержимое указанного файла. Но независимо от формы конструктора файл создается, если он не существует.
- При появлении ошибок ввода- вывода в обоих случаях генерируется исключение IOException.

```
// Открыть файл средствами класса StreamWriter.
string str;
        StreamWriter fstr_out = null;
        try {
            // Открыть файл, заключенный в оболочку класса StreamWriter.
            fstr_out = new StreamWriter("test.txt");
            Console.WriteLine("Введите текст, а по окончании – 'стоп'.");
            do
            {
                Console.Write(": ");
                str = Console.ReadLine();
                if (str != "стоп")
                {
                    str = str + "\r\n"; // добавить новую строку
                    fstr_out.Write(str);
                }
            } while (str != "стоп");
        }
        catch (IOException exc){
            Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
        }
        finally
        {
            if (fstr_out != null) fstr_out.Close();
        }
```

Применение класса StreamReader

- Для создания символьного потока ввода достаточно заключить байтовый поток в оболочку класса **StreamReader**:

```
StreamReader(Stream поток);
```

где поток обозначает имя открытого потока.

- Этот конструктор генерирует исключение **ArgumentNullException**, если поток оказывается пустым, а также исключение **ArgumentException**, если поток не открыт для ввода.
- После своего создания объект класса StreamReader выполняет автоматическое преобразование байтов в символы. По завершении ввода из потока типа StreamReader его нужно закрыть.

```
// программа ввода с диска и вывода на экран (StreamReader)
FileStream fin;
        string s;
        try {
            fin = new FileStream("test.txt", FileMode.Open);
        }
        catch (IOException exc){
            Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
            return;
        }
        StreamReader fstr_in = new StreamReader(fin);
        try {
            while ((s = fstr_in.ReadLine()) != null)
            {
                Console.WriteLine(s);
            }
        }
        catch (IOException exc)
        {
            Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
        }
        finally
        {
            fstr_in.Close();
        }
    }
```

Применение класса StreamReader

- Свойство **EndOfStream** можно использовать для отслеживания конца файла:

```
while(!fstr_in.EndOfStream) {
  s = fstr_in.ReadLine();
  Console.WriteLine(s);
}
```

- Иногда файл проще открыть, используя непосредственно класс StreamReader, аналогично классу StreamWriter:

```
StreamReader(string путь);
```

где путь — это имя открываемого файла, включая полный путь к нему.

- Указываемый файл должен существовать. В противном случае генерируется исключение **FileNotFoundException**. Если путь оказывается пустым, то генерируется исключение **ArgumentNullException**. А если путь содержит пустую строку, то генерируется исключение **ArgumentException**. Кроме того, могут быть сгенерированы исключения **IOException** и **DirectoryNotFoundException**.

Переадресация стандартных потоков

- Стандартные потоки, например Console.In, могут быть **переадресованы**. И чаще всего они переадресовываются в файл
- Переадресация стандартных потоков достигается двумя способами.
- **Первый способ:** при выполнении программы из командной строки с помощью операторов < и >, переадресовывающих потоки Console.In и Console.Out соответственно.
- Допустим, что имеется следующая программа.

```
using System;
class Test {
  static void Main() {
    Console.WriteLine("Это тест.");
  }
}
```

- Если выполнить эту программу из командной строки

```
Test > log
```

то символьная строка "Это тест." будет выведена в файл log. Аналогичным образом переадресуется ввод.

Переадресация стандартных потоков

- Для второго способа используются следующие методы:

```
static void SetIn(TextReader новый_поток_ввода);
static void SetOut(TextWriter новый_поток_вывода);
static void SetError(TextWriter новый_поток_сообщений_об_ошибках);
```

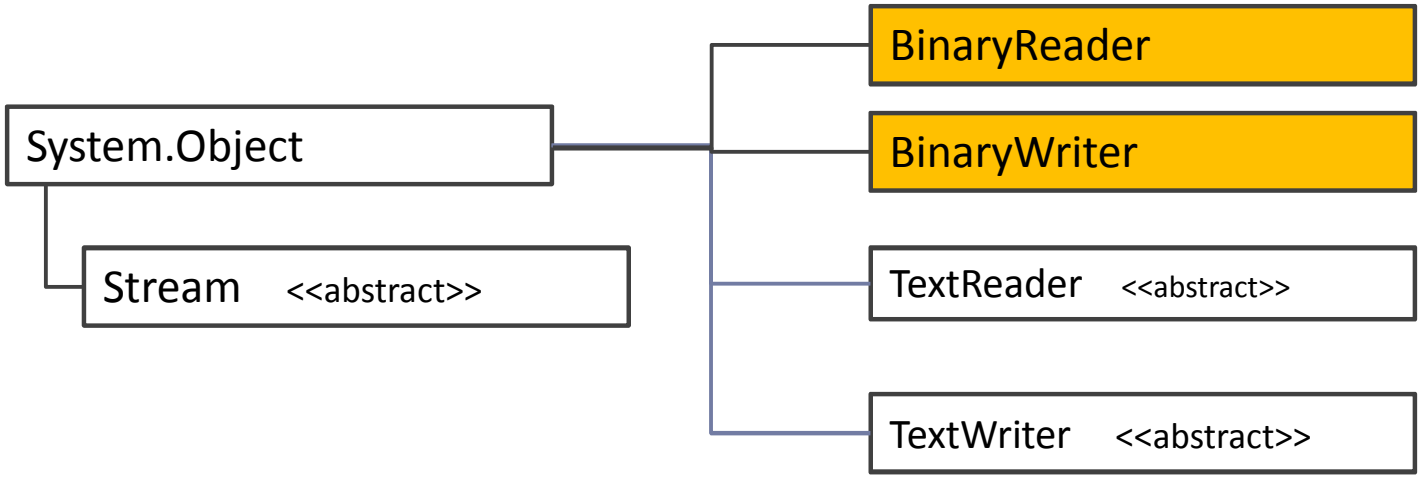
```
// Переадресовать поток Console.Out
static void Main() {
        StreamWriter log_out = null;
        try {
            log_out = new StreamWriter("logfile.txt");
            // Переадресовать стандартный вывод в файл logfile.txt.
            Console.SetOut(log_out);
            Console.WriteLine("Это начало файла журнала" +
                              + "регистрации.");
            for (int i = 0; i < 10; i++) Console.WriteLine(i);
            Console.WriteLine("Это конец файла журнала регистрации.");
        }
        catch (IOException exc) {
            Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
        }
        finally{ if (log_out != null) log_out.Close(); }
    }
```

Результат:

```
Это начало
файла журнала
регистрации.
0
1
2
3
4
5
6
7
8
9
Это конец файла
журнала
регистрации.
```

Классы потоков: двоичные потоки

- **Двоичные потоки** могут служить для непосредственного ввода и вывода двоичных данных — **BinaryReader** и **BinaryWriter**.



Чтение и запись двоичных данных BinaryWriter

- Для чтения и записи двоичных значений встроенных в C# типов данных служат классы потоков **BinaryReader** и **BinaryWriter**.
- Класс **BinaryWriter** служит оболочкой, в которую заключается байтовый поток, управляющий выводом двоичных данных:

```
BinaryWriter(Stream output);
```

- где output обозначает поток, в который выводятся записываемые данные.
  Для записи в выходной файл в качестве параметра output может быть указан объект, создаваемый средствами класса FileStream.
- Если же параметр output оказывается пустым, то генерируется исключение **ArgumentNullException**. А если поток, определяемый параметром output, не был открыт для записи данных, то генерируется исключение **ArgumentException**.
- По завершении вывода в поток типа BinaryWriter его нужно закрыть. При этом закрывается и базовый поток.

- **Таблица 5. Наиболее часто используемые методы, определенные в классе BinaryWriter**

Метод	Описание
`void Write(sbyte value)`	Записывает значение типа `sbyte` со знаком
`void Write(byte value)`	Записывает значение типа `byte` без знака
`void Write(byte[] buffer)`	Записывает массив значений типа `byte`
`void Write(short value)`	Записывает целочисленное значение типа `short` (короткое целое)
`void Write(ushort value)`	Записывает целочисленное значение типа `ushort` (короткое целое без знака)
`void Write(int value)`	Записывает целочисленное значение типа `int`
`void Write(uint value)`	Записывает целочисленное значение типа `uint` (целое без знака)
`void Write(long value)`	Записывает целочисленное значение типа `long` (длинное целое)
`void Write(ulong value)`	Записывает целочисленное значение типа `ulong` (длинное целое без знака)
`void Write(float value)`	Записывает значение типа `float` (с плавающей точкой одинарной точности)
`void Write(double value)`	Записывает значение типа `double` (с плавающей точкой двойной точности)
`void Write(decimal value)`	Записывает значение типа `decimal` (с двумя десятичными разрядами после запятой)
`void Write(char ch)`	Записывает символ
`void Write(char[] buffer)`	Записывает массив символов
`void Write(string value)`	Записывает строковое значение типа `string`, представленное во внутреннем формате с указанием длины строки

Чтение и запись двоичных данных BinaryReader

- Класс **BinaryReader** служит оболочкой, в которую заключается байтовый поток, управляющий вводом двоичных данных:

```
BinaryReader(Stream input);
```

где input обозначает поток, из которого вводятся считываемые данные. Для чтения из входного файла в качестве параметра input может быть указан объект, создаваемый средствами класса **FileStream**.

- Если же поток, определяемый параметром input, не был открыт для чтения данных или оказался недоступным по иным причинам, то генерируется исключение **ArgumentException**.
- По завершении ввода из потока типа BinaryReader его нужно закрыть. При этом закрывается и базовый поток.
- Следует иметь в виду, что в методе ReadString() считывается символьная строка, хранящаяся во внутреннем формате с указанием ее длины.
- Все методы данного класса генерируют исключение **IOException**, если возникает ошибка ввода.

Таблица 6. Наиболее часто используемые методы, определенные в классе BinaryReader

Метод	Описание
`bool ReadBoolean()`	Считывает значение логического типа `bool`
`byte ReadByte()`	Считывает значение типа `byte`
`sbyte ReadSByte()`	Считывает значение типа `sbyte`
`byte[] ReadBytes(int count)`	Считывает количество *`count`* байтов и возвращает их в виде массива
`char ReadChar()`	Считывает значение типа `char`
`char[] ReadChars(int count)`	Считывает количество *`count`* символов и возвращает их в виде массива
`decimal ReadDecimal()`	Считывает значение типа `decimal`
`double ReadDouble()`	Считывает значение типа `double`
`float ReadSingle()`	Считывает значение типа `float`
`short ReadInt16()`	Считывает значение типа `short`
`int ReadInt32()`	Считывает значение типа `int`
`long ReadInt64()`	Считывает значение типа `long`
`ushort ReadUInt16()`	Считывает значение типа `ushort`
`uint ReadUInt32()`	Считывает значение типа `uint`
`ulong ReadUInt64()`	Считывает значение типа `ulong`
`string ReadString()`	Считывает значение типа `string`, представленное во внутреннем двоичном формате с указанием длины строки. Этот метод следует использовать для считывания строки, которая была записана средствами класса `BinaryWriter`

Чтение и запись двоичных данных BinaryReader

- В классе BinaryReader определены также три приведенных ниже варианта метода **Read()**.
- При неудачном исходе операции чтения эти методы генерируют исключение **IOException**.
- Кроме того, в классе BinaryReader определен стандартный метод **Close ()**.

Метод	Описание
int Read ()	Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца файла возвращает значение -1
int Read (byte [ ] buffer, int offset, int count)	Делает попытку прочитать количество count байтов в массив buffer, начиная с элемента buffer[offset], и возвращает количество успешно считанных байтов
int Read(char [ ] buffer, int offset, int count)	Делает попытку прочитать количество count символов в массив buffer, начиная с элемента buffer [offset], и возвращает количество успешно считанных символов

```
// Записать двоичные данные, а затем считать их обратно.
BinaryWriter dataOut;
BinaryReader dataIn;
int i = 10; double d = 1023.56;
bool b = true; string str = "Это тест";
// Открыть файл для вывода
try  {
  dataOut = new BinaryWriter(new FileStream("testdata", FileMode.Create));
}
catch (IOException exc) {
  Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
  return;
}
// Записать данные в файл
try {
  Console.WriteLine("Запись " + i); dataOut.Write(i);
  Console.WriteLine("Запись " + d); dataOut.Write(d);
  Console.WriteLine("Запись " + b); dataOut.Write(b);
  Console.WriteLine("Запись " + 12.2 * 7.4);
  dataOut.Write(2.2 * 7.4);
  Console.WriteLine("Запись " + str); dataOut.Write(str);
 }
 catch (IOException exc) {
   Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
 }
 //см. далее…
```

```
   finally {
     dataOut.Close(); Console.WriteLine();
     //А теперь прочитать данные из файла,
     try {
       dataIn = new BinaryReader(new FileStream("testdata",
FileMode.Open));
     }
     catch (IOException exc) {
       Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
       return;
     }
     try {
       i = dataIn.ReadInt32(); Console.WriteLine("Чтение " + i);
       d = dataIn.ReadDouble(); Console.WriteLine("Чтение " + d);
       b = dataIn.ReadBoolean(); Console.WriteLine("Чтение " + b);
       d = dataIn.ReadDouble(); Console.WriteLine("Чтение " + d);
       str = dataIn.ReadString();
       Console.WriteLine("Чтение " + str);
     }
     catch (IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
     }
     finally {
       dataIn.Close();
     }
   }
```

```
Результат:
Запись 10
Запись
1023.56
Запись True
Запись 90.28
Запись
Это тест

Чтение 10
Чтение
1023.56
Чтение True
Чтение 90.28
Чтение Это
тест
```

```
/* Использовать классы BinaryReader и BinaryWriter для
реализации простой программы учета товарных запасов. */
 BinaryWriter dataOut; BinaryReader dataIn;
 string item; // наименование предмета
 int onhand; // имеющееся в наличии количество
 double cost; // цена
 try {
   dataOut = new
   BinaryWriter(new FileStream("inventory.dat", FileMode.Create));
 }
 catch (IOException exc) {
   Console.WriteLine("Не удается открыть файл товарных запасов для вывода");
   Console.WriteLine("Причина: " + exc.Message);
   return;
 }
 // Записать данные о товарных запасах в файл,
 try {
   dataOut.Write("Молотки"); dataOut.Write(10); dataOut.Write(3.95);
   dataOut.Write("Отвертки"); dataOut.Write(18); dataOut.Write(1.50);
   dataOut.Write("Плоскогубцы"); dataOut.Write(5); dataOut.Write(4.95);
   dataOut.Write("Пилы"); dataOut.Write(8); dataOut.Write(8.95);
 }
 catch (IOException exc) {
   Console.WriteLine("Ошибка записи в файл товарных запасов");
   Console.WriteLine("Причина: " + exc.Message);
 }
 finally { dataOut.Close();  }
 //см. далее...
```

```
//А теперь открыть файл товарных запасов для чтения,
try {
    dataIn = new BinaryReader(new FileStream("inventory.dat", FileMode.Open));
}
catch (IOException exc) {
    Console.WriteLine("Не удается открыть файл товарных запасов для ввода");
    Console.WriteLine("Причина: " + exc.Message);
    return;
}
// Найти предмет, введенный пользователем.
Console.Write("Введите наименование для поиска: ");
string what = Console.ReadLine(); Console.WriteLine();
try {
    for (; ;) {
        // Читать данные о предмете хранения
        item = dataIn.ReadString(); onhand = dataIn.ReadInt32();
        cost = dataIn.ReadDouble();
        // Проверить, совпадает ли он с запрашиваемым предметом.
        // Если совпадает, то отобразить сведения о нем.
        if (item.Equals(what, StringComparison.OrdinallgnoreCase)) {
            Console.WriteLine(item + ": " + onhand + " штук в наличии. "+
            "Цена: {0:C} за штуку", cost);
            Console.WriteLine("Общая стоимость по наименованию <{0}>: {1:C}.",
            item, cost * onhand);
               break;
}    }     } //см. далее…
```

```
 catch (EndOfStreamException) { Console.WriteLine("Предмет не найден."); }
  catch (IOException exc) {
     Console.WriteLine("Ошибка чтения из файла товарных запасов");
     Console.WriteLine("Причина: " + exc.Message);
  }
  finally {  dataIn.Close(); }
}
```

Результат:

```
Введите наименование для поиска: Отвертки
Отвертки: 18 штук в наличии. Цена: $1.50 за штуку.
Общая стоимость по наименованию <Отвертки>: $27.00.
```

Файлы с произвольным доступом

- Доступ к содержимому файла может быть и произвольным. Для этого служит, в частности, метод **Seek ()**, определенный в классе FileStream. Этот метод позволяет установить указатель положения в файле, или так называемый указатель файла, на любое место в файле. Общая форма:

```
long Seek(long offset, SeekOrigin origin);
```

где offset обозначает новое положение указателя файла в байтах относительно заданного начала отсчета (origin). В качестве origin может быть указано одно из приведенных ниже значений, определяемых в перечислении SeekOrigin.

Значение	Описание
SeekOrigin.Begin	Поиск от начала файла
SeekOrigin.Current	Поиск от текущего положения
SeekOrigin.End	Поиск от конца файла

- Следующая операция чтения или записи после вызова метода Seek () будет выполняться, начиная с нового положения в файле, возвращаемого этим методом.
- Если во время поиска в файле возникает ошибка, то генерируется исключение **IOException**. Если же запрос положения в файле не поддерживается базовым потоком, то генерируется исключение **NotSupportedException**.

```
// Продемонстрировать произвольный доступ к файлу.
FileStream f = null; char ch;
try {
    f = new FileStream("random.dat", FileMode.Create);
    // Записать английский алфавит в файл.
    for (int i = 0; i < 26; i++) f.WriteByte((byte)('A' + i));
    //А теперь считать отдельные буквы английского алфавита
    f.Seek(0, SeekOrigin.Begin); // найти нулевой байт
    ch = (char)f.ReadByte();
    Console.WriteLine("Первая буква: " + ch);
    f.Seek(1, SeekOrigin.Begin); ch = (char)f.ReadByte();
    Console.WriteLine("Вторая буква: " + ch);
    f.Seek(4, SeekOrigin.Begin); ch = (char)f.ReadByte();
    Console.WriteLine("Пятая буква: " + ch);
    Console.WriteLine();
    //А теперь прочитать буквы английского алфавита через одну.
    Console.WriteLine("Буквы алфавита через одну: ");

    for (int i = 0; i < 26; i += 2) {
        f.Seek(i, SeekOrigin.Begin); // найти i-й символ
        ch = (char)f.ReadByte();
        Console.Write(ch + " ");
}   }

catch (IOException exc) {
    Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
}

finally { if (f != null) f.Close(); }
```

Результат:

```
Первая буква: A
Вторая буква: B
Пятая буква: E
Буквы алфавита,
через одну:
ACEGIKMOQSUWY
```

Файлы с произвольным доступом

- Несмотря на то что метод Seek () имеет немало преимуществ при использовании с файлами, существует и другой способ установки текущего положения в файле с помощью свойства **Position**.
- Свойство Position доступно как для чтения, так и для записи. Поэтому с его помощью можно получить или же установить текущее положение в файле.
- В качестве примера ниже приведен фрагмент кода из предыдущей программы записи и чтения из файла с произвольным доступом random.dat, измененный с целью продемонстрировать применение свойства Position:

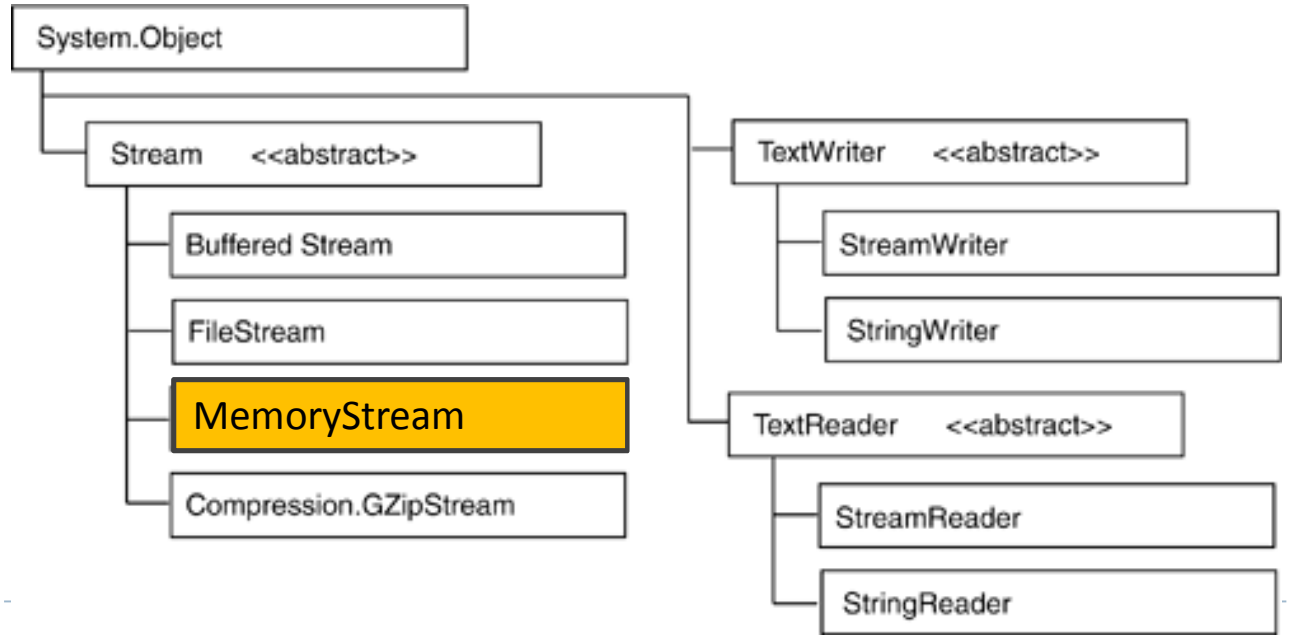
```
Console.WriteLine("Буквы алфавита через одну: ");
for(int i=0; i<26; i+=2) {
  f.Position = i; // найти i-й символ посредством свойства Position
  ch = (char) f.ReadByte ();
  Console.Write(ch + " ") ;
}
```

Применение класса MemoryStream

- Класс **MemoryStream** представляет собой реализацию класса Stream, в которой массив байтов используется для ввода и вывода:

```
MemoryStream(byte[ ] buffer);
```

где buffer обозначает массив байтов, используемый в качестве источника или адресата в запросах ввода-вывода. Используя этот конструктор, следует иметь в виду, что массив buffer должен быть достаточно большим для хранения направляемых в него данных.



```
// Продемонстрировать применение класса MemoryStream.
static void Main() {
        byte[] storage = new byte[255];
// Создать запоминающий поток.
        MemoryStream memstrm = new MemoryStream(storage);
// Заключить объект memstrm в оболочки классов
//чтения и записи данных в потоки.
StreamWriter memwtr = new StreamWriter(memstrm);
StreamReader memrdr = new StreamReader(memstrm);
try {
  // Записать данные в память, используя объект memwtr.
  for (int i = 0; i < 10; i++)
    memwtr.WriteLine("byte [" + i + "]:" + i);
  // Поставить в конце точку,
  memwtr.WriteLine(".");
  memwtr.Flush();
  Console.WriteLine("Чтение прямо из массива storage: ");
  // Отобразить содержимое массива storage непосредственно.
  foreach (char ch in storage) {
    if (ch == '.') break;
    Console.Write(ch);
  }
  //см. далее..
```

Результат:

```
Чтение прямо из массива storage:
byte [0]: 0
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

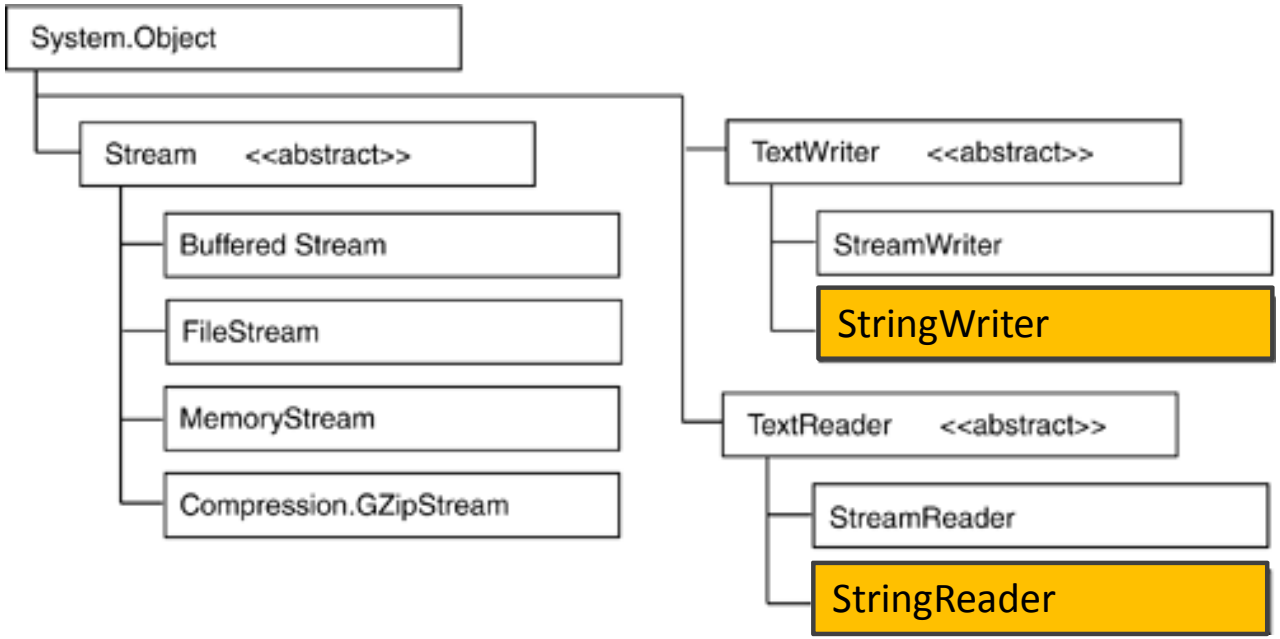
```
//…продолжение
Console.WriteLine("Чтение из потока с помощью объекта memrdr: ");
// Читать из объекта memstrm средствами ввода данных из потока
// установить указатель файла в исходное положение
memstrm.Seek(0, SeekOrigin.Begin); string str = memrdr.ReadLine();
        while (str != null) {
            str = memrdr.ReadLine();
            if (str[0] == '.') break;
           Console.WriteLine(str);
        }
    }
    catch (IOException exc) {
        Console.WriteLine("Ошибка ввода-вывода\n"
           + exc.Message);
    }
    finally {
   // Освободить ресурсы считывающего и
   //записывающего потоков
        memwtr.Close(); memrdr.Close();
    }
}
```

Результат:

```
Чтение из потока с
помощью объекта
memrdr:
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

Применение классов StringReader и StringWriter

- Класс **StringReader** наследует от класса TextReader, а класс **StringWriter** — от класса TextWriter. Следовательно, они представляют собой потоки, имеющие доступ к методам, определенным в этих двух базовых классах, что позволяет, например, вызывать метод ReadLine() для объекта класса StringReader, а метод WriteLine() — для объекта класса StringWriter.



Применение классов StringReader и StringWriter

- Конструктор класса StringReader:

```
StringReader(string s);
```

где s обозначает символьную строку, из которой производится чтение.

- Конструктор класса StringWriter:

```
StringWriter();
```

- Этот конструктор создает записывающий поток, который помещает выводимые данные в строку. Для получения содержимого этой строки достаточно вызвать метод ToString() .

```
//Продемонстрировать применение классов StringReader и StringWriter.
 StringWriter strwtr = null;
 StringReader strrdr = null;
 try{
     // Создать объект класса StringWriter.
     strwtr = new StringWriter();
     // Вывести данные в записывающий поток типа StringWriter.
     for (int i = 0; i < 10; i++)
          strwtr.WriteLine("Значение i равно: " + i);
     // Создать объект класса StringReader.
     strrdr = new StringReader(strwtr.ToString());
     //А теперь ввести данные из считывающего потока
     //типа StringReader.
     string str = strrdr.ReadLine();
     while (str != null){
         str = strrdr.ReadLine(); Console.WriteLine(str);
     }
 }
 catch (IOException exc) {
     Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
 }
 finally{
     // Освободить ресурсы считывающего и записывающего потоков,
     if (strrdr != null) strrdr.Close();
     if (strwtr != null) strwtr.Close();
 }
```

```
Результат:
Значение i равно: 1
Значение i равно: 2
Значение i равно: 3
Значение i равно: 4
Значение i равно: 5
Значение i равно: 6
Значение i равно: 7
Значение i равно: 8
Значение i равно: 9
```

Класс File

- В классе **File** имеются методы для копирования и перемещения, шифрования и расшифровывания, удаления файлов, а также для получения и задания информации о файлах, включая сведения об их существовании, времени создания, последнего доступа и различные атрибуты файлов (только для чтения, скрытых и пр.).
- Кроме того, в классе File имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа **FileStream** на него.
- Ряд методов для работы с файлами определен также в классе **FileInfo**. Этот класс отличается от класса File одним, очень важным преимуществом: для операций над файлами он предоставляет методы экземпляра и свойства, а не статические методы. Поэтому для выполнения нескольких операций над одним и тем же файлом лучше воспользоваться классом FileInfo.

Копирование файлов с помощью метода Copy ()

- Метода Copy () определен в классе File:

```
static void Copy (string имя_исходного_файла, string
имя_целевого_файла);
```

```
static void Copy (string имя_исходного_файла, string имя_целевого_файла,
boolean overwrite);
```

- Метод Copy () копирует файл, на который указывает имя_исходного_файла, в файл, на который указывает имя_целевого_файла.
- В первой форме данный метод копирует файл только в том случае, если файл, на который указывает имя_целевого_файла, еще не существует.
- Во второй форме копия заменяет и перезаписывает целевой файл, если он существует и если параметр overwrite принимает логическое значение true.
- Но в обоих случаях может быть сгенерировано несколько видов исключений, включая **IOException** и **FileNotFoundException**.

```
/* Скопировать файл, используя метод File.Copy().
чтобы скопировать файл FIRST.DAT в файл SECOND.DAT, введите:
CopyFile FIRST.DAT SECOND.DAT */
using System;
using System.IO;
class CopyFile
{
    static void Main(string[] args)
    {
        if (args.Length != 2)
        {
            Console.WriteLine("Применение: CopyFile Откуда Куда");
            return;
        }
        // Копировать файлы,
        try
        {
            File.Copy(args[0], args[1]);
        }
        catch (IOException exc)
        {
            Console.WriteLine("Ошибка копирования файла\n" + exc.Message);
        }
    }
}
```

Применение методов Exists() и GetLastAccessTime()

- Метод **Exists()** определяет, существует ли файл, а метод **GetLastAccessTime()** возвращает дату и время последнего доступа к файлу:

```
static bool Exists(string путь);
static DateTime GetLastAccessTime(string путь);
```

- В обоих методах путь обозначает файл, сведения о котором требуется получить. Метод Exists () возвращает логическое значение true, если файл существует и доступен для вызывающего процесса. А метод GetLastAccessTime () возвращает структуру DateTime, содержащую дату и время последнего доступа к файлу.
- Метод ToString () автоматически приводит дату и время к удобочитаемому виду.
- С указанием недействительных аргументов или прав доступа при вызове обоих рассматриваемых здесь методов может быть связан целый ряд исключений, но в действительности генерируется только исключение IOException.

```
// Применить методы Exists () и GetLastAccessTime() .
if (File.Exists("test.txt"))
            Console.WriteLine("Файл существует. В последний раз он“
              + "был доступен " + File.GetLastAccessTime("test.txt"));
        else
            Console.WriteLine("Файл не существует");
```

Кроме того, время создания файла можно выяснить,
вызвав метод **GetCreationTime()**,
а время последней записи в файл, вызвав метод **GetLastWriteTime()**.
Имеются также варианты этих методов для представления данных о файле в формате всеобщего скоординированного времени (UTC).

Преобразование числовых строк в их внутреннее представление

Числовые значения автоматически преобразуются методом **WriteLine()** в удобную для чтения текстовую форму.
В то же время аналогичный метод ввода для чтения и преобразования строк с числовыми значениями в двоичный формат их внутреннего представления не предоставляется , однако можно воспользоваться методом **Parse()**, определенным для всех встроенных числовых типов данных.

Эти структуры определены в пространстве имен System:

Имя структуры в .NET	Имя типа данных в C#
Decimal	decimal
Double	double
Single	float
Intl6	short
Int32	int
Int64	long
UIntl6	ushort
UInt32	uint
UInt64	ulong
Byte	byte
Sbyte	sbyte

Перегрузки метода Parse

Следует иметь в виду, что каждый метод возвращает двоичное значение, соответствующее преобразуемой строке:

Структура	Метод преобразования
Decimal	static decimal Parse(string s)
Double	static double Parse(string s)
Single	static float Parse(string s)
Int64	static long Parse(string s)
Int32	static int Parse(string s)
Intl6	static short Parse(string s)
UInt64	static ulong Parse(string s)
UInt32	static uint Parse(string s)
UIntl6	static ushort Parse(string s)
Byte	static byte Parse(string s)
Sbyte	static sbyte Parse (string s)

Приведенные выше варианты метода Parse () генерируют исключение **FormatException**, если строка s не содержит допустимое число, определяемое вызывающим типом данных. А если она содержит пустое значение, то генерируется исключение **ArgumentNullException**. Когда же значение в строке s превышает допустимый диапазон чисел для вызывающего типа данных, то генерируется исключение **OverflowException**.

```
// Эта программа усредняет ряд чисел, вводимых пользователем.
static void Main() {
    string str; int n; double sum = 0.0; double avg, t;
    Console.Write("Сколько чисел вы собираетесь ввести: ");
    str = Console.ReadLine();
    try  { n = Int32.Parse(str); }
    catch (FormatException exc) {
        Console.WriteLine(exc.Message); return; }
    catch (OverflowException exc) {
        Console.WriteLine(exc.Message); return; }
    Console.WriteLine("Введите " + n + " чисел.");
    for (int i = 0; i < n; i++) {
        Console.Write("> "); str = Console.ReadLine();
        try { t = Double.Parse(str); }
        catch (FormatException exc) {
            Console.WriteLine(exc.Message);
            t = 0.0;
        }
        catch (OverflowException exc) {
            Console.WriteLine(exc.Message);
            t = 0;
        }
        sum += t;
    }
    avg = sum / n;
    Console.WriteLine("Среднее равно " + avg);
}
```

Результат:

```
Сколько чисел вы
собираетесь ввести: 5
Введите 5 чисел.
> 1,1
> 2,2
> 3,3
> 4,4
> 5,5
Среднее равно 3,3
```

Метод Parse

- Следует особо подчеркнуть, что для каждого преобразуемого значения необходимо выбирать подходящий метод синтаксического анализа. Так, если попытаться преобразовать строку, содержащую значение с плавающей точкой, методом **Int32.Parse()**, то искомый результат, т.е. числовое значение с плавающей точкой, получить не удастся.
- Для того чтобы избежать генерирования исключений при преобразовании числовых строк, можно воспользоваться методом **TryParse ()**, определенным для всех числовых структур. В качестве примера ниже приведен один из вариантов метода TryParse (), определяемых в структуре Int32:

```
static bool TryParse(string s, out int результат);
```

где **s** обозначает числовую строку, передаваемую данному методу, который возвращает соответствующий результат после преобразования с учетом выбираемой по умолчанию местной специфики представления чисел. При неудачном исходе преобразования, например, когда параметр **s** не содержит числовую строку в надлежащей форме, метод TryParse () возвращает логическое значение false. В противном случае он возвращает логическое значение true.