

# Структуры данных Контейнерные классы Работа с файлами

---

# Абстрактные структуры данных

- *Массив*

конечная совокупность однотипных величин. Занимает непрерывную область памяти и предоставляет прямой (произвольный) доступ к элементам по индексу.

- *Линейный список*

- *Стек*

- *Очередь*

- *Бинарное дерево*

- *Хеш-таблица (ассоциативный массив, словарь)*

- *Граф*

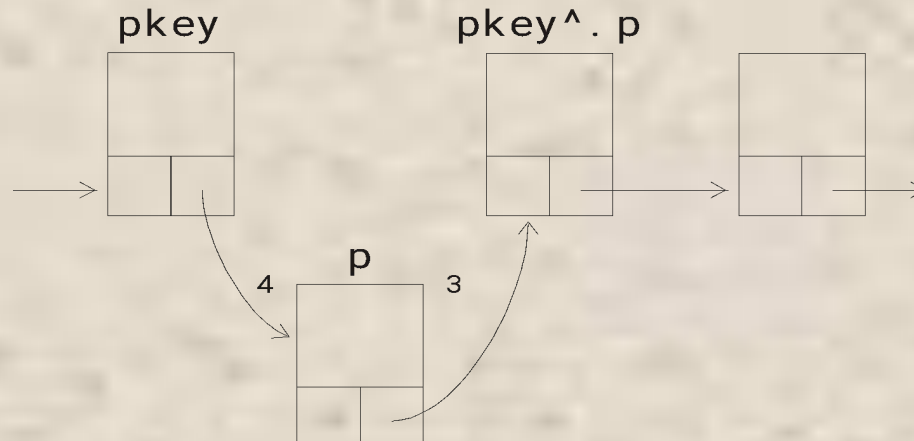
- *Множество*

# Линейный список

- В *списке* каждый элемент связан со следующим и, возможно, с предыдущим. Количество элементов в списке может изменяться в процессе работы программы.
- Каждый элемент списка содержит *ключ*, идентифицирующий этот элемент.
- Виды списков:

односвязные ← кольцевые  
двусвязные ←

- Преимущество перед массивом - простая вставка элементов:

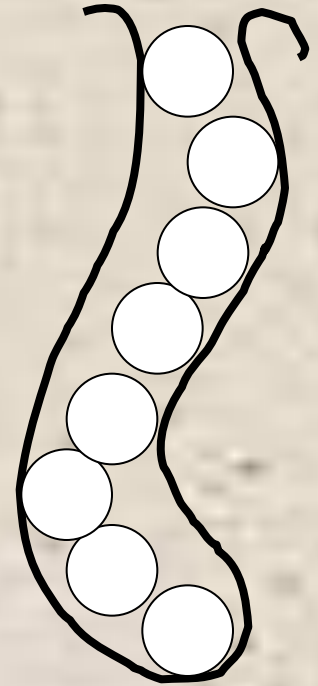


# Стек

*Стек* — частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца, называемого вершиной стека (стек реализует принцип обслуживания LIFO).

Другие операции со стеком не определены.

При выборке элемент исключается из стека.

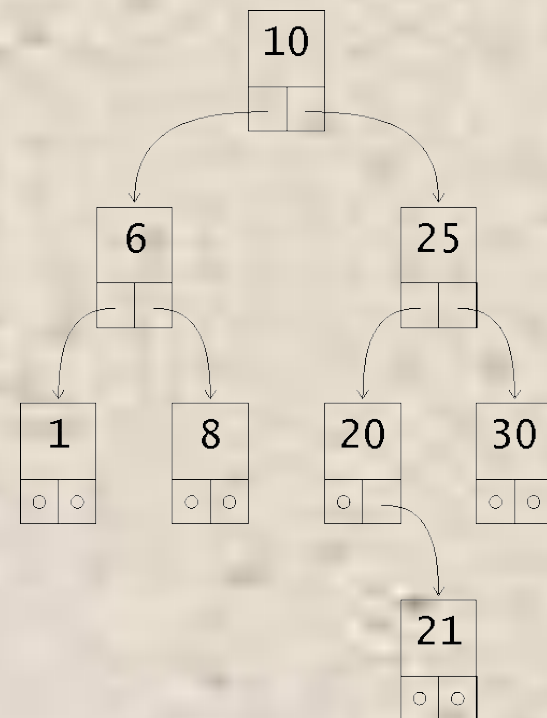


# Очередь

- *Очередь* — частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца (очередь реализует принцип обслуживания FIFO). Другие операции с очередью не определены.
- При выборке элемент исключается из очереди.

# Бинарное дерево

- *Бинарное дерево* — динамическая структура данных, состоящая из узлов, каждый из которых содержит, помимо данных, не более двух ссылок на различные бинарные поддеревья.
- На каждый узел имеется ровно одна ссылка. Начальный узел называется *корнем* дерева.
- Узел, не имеющий поддеревьев, называется *листом*. Исходящие узлы называются *предками*, входящие — *потомками*.
- *Высота дерева* определяется количеством уровней, на которых располагаются его узлы.

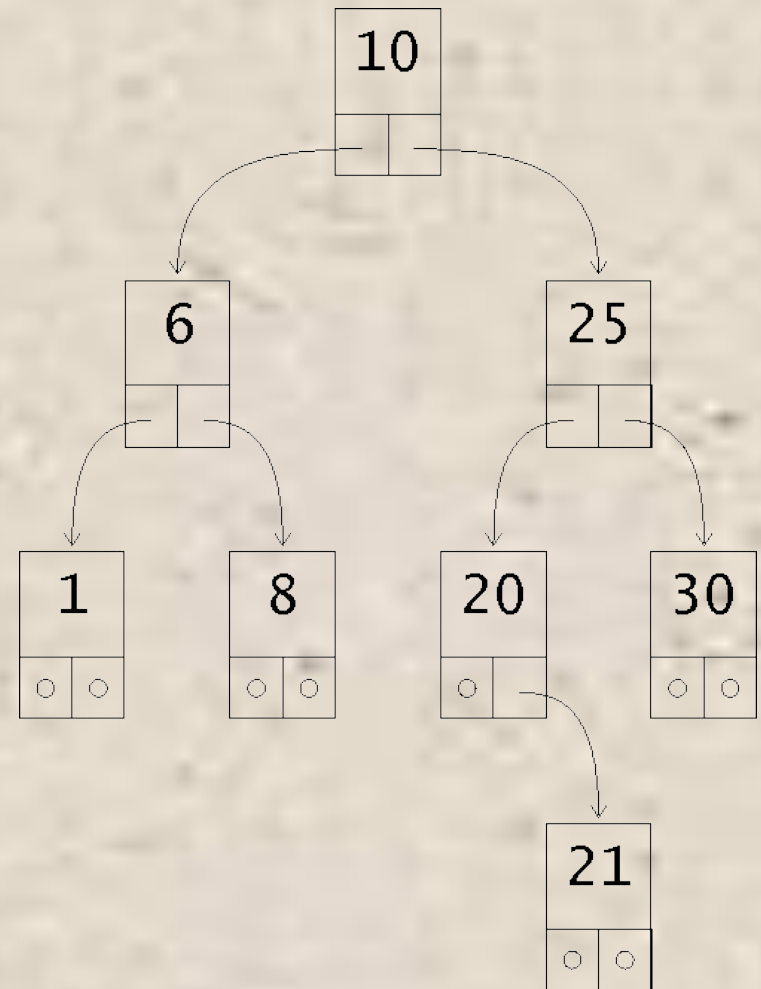


# Дерево поиска

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддеревья меньше ключа этого узла, а все ключи его правого поддеревья — больше, оно называется *деревом поиска*.

Одинаковые ключи не допускаются.

В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле.

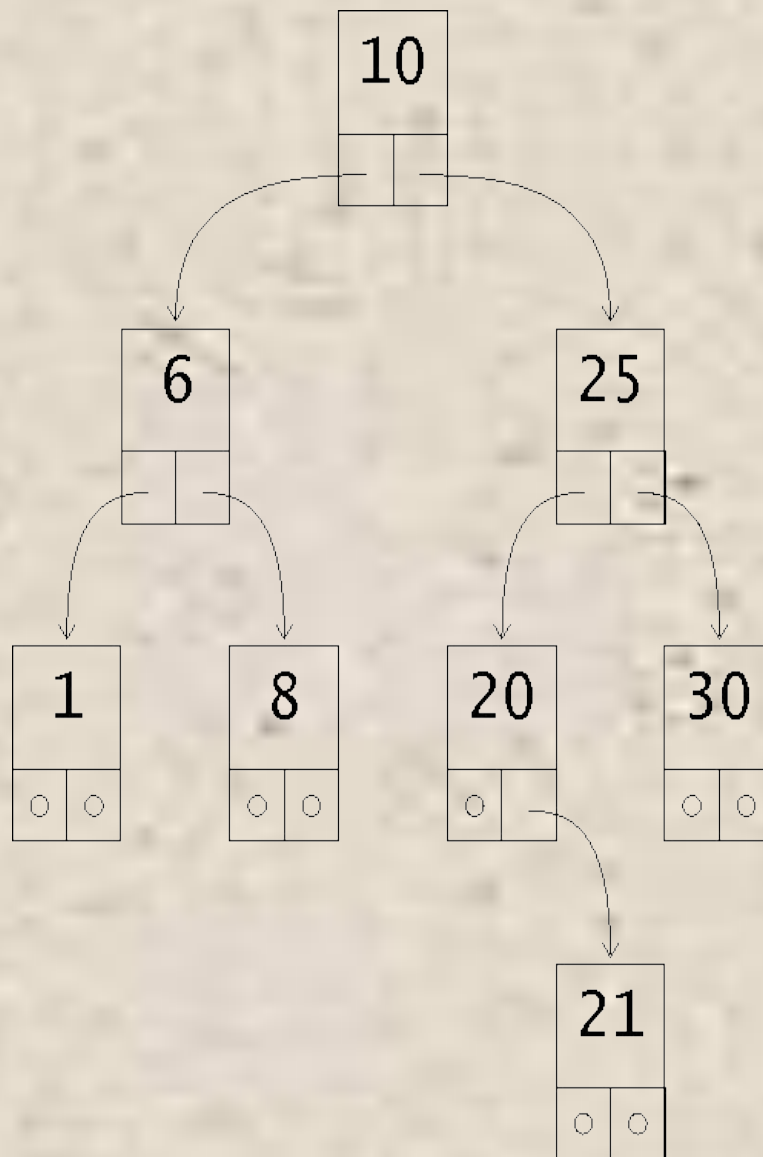




# Обход дерева

```
procedure print_tree( дерево );  
begin  
  print_tree( левое_поддерево )  
  посещение корня  
  print_tree( правое_поддерево )  
end;
```

1 6 8 10 20 21 25 30





# Хеш-таблица

- *Хеш-таблица (ассоциативный массив, словарь) — массив, доступ к элементам которого осуществляется не по номеру, а по ключу (т.е. это таблица, состоящая из пар «ключ-значение»)*
  - рус-англ-словарь:  
{кукла} -> doll

Ключ	Значение
boy	мальчик
girl	девочка
dog	собачка

Хеш-таблица эффективно реализует операцию поиска значения по ключу. Ключ преобразуется в число (*хэш-код*), которое используется для быстрого нахождения нужного значения в хеш-таблице.

# Граф, множество

- *Граф* — совокупность узлов и ребер, соединяющих различные узлы. Множество реальных практических задач можно описать в терминах графов, что делает их структурой данных, часто используемой при написании программ.
- *Множество* — неупорядоченная совокупность элементов. Для множеств определены операции:
  - проверки принадлежности элемента множеству
  - включения и исключения элемента
  - объединения, пересечения и вычитания множеств.
- Все эти структуры данных называются *абстрактными*, поскольку в них не задается реализация допустимых операций.

# Контейнеры

<http://msdn.microsoft.com/ru-ru/library/ybcx56wz.aspx?ppud=4>

- *Контейнер (коллекция)* - стандартный класс, реализующий абстрактную структуру данных.
- Для каждого типа коллекции определены методы работы с ее элементами, не зависящие от конкретного типа хранимых данных.
- Использование коллекций позволяет сократить сроки разработки программ и повысить их надежность.
- Каждый вид коллекции поддерживает свой набор операций над данными, и быстродействие этих операций может быть разным.
- Выбор вида коллекции зависит от того, что требуется делать с данными в программе и какие требования предъявляются к ее быстродействию.
- В библиотеке .NET определено множество стандартных контейнеров.
- Основные пространства имен, в которых они описаны — `System.Collections`, `System.Collections.Specialized` и `System.Collections.Generic`

# System.Collections

<b>ArrayList</b>	Массив, динамически изменяющий свой размер
<b>BitArray</b>	Компактный массив для хранения битовых значений
<b>Hashtable</b>	Хэш-таблица
<b>Queue</b>	Очередь
<b>SortedList</b>	Коллекция, отсортированная по ключам. Доступ к элементам — по ключу или по индексу
<b>Stack</b>	Стек

# Параметризованные коллекции (классы-прототипы, generics)

- классы, имеющие типы данных в качестве параметров

## Класс-прототип (версия 2.0)

Dictionary<K,T>

**LinkedList<T>**

**List<T>**

Queue<T>

SortedDictionary<K,T>

Stack<T>

## Обычный класс

HashTable

—

ArrayList

Queue

SortedList

Stack

# Выбор класса коллекции

- Нужен ли последовательный список, элемент которого обычно удаляется сразу после извлечения его значения? (Queue, Stack)
- Нужен ли доступ к элементам в определенном порядке (FIFO, LIFO) или в произвольном порядке? (Queue, Stack, LinkedList)
- Необходимо ли иметь доступ к каждому элементу по индексу? (ArrayList, StringCollection, List, ...)
- Будет ли каждый элемент содержать только одно значение, сочетание из одного ключа и одного значения или сочетание из одного ключа и нескольких значений?
- Нужна ли возможность отсортировать элементы в порядке, отличном от порядка их поступления? (HashTable, SortedList,...)
- Необходимы ли быстрый поиск и извлечение данных? (Dictionary)
- Нужна ли коллекция только для хранения строк? (StringCollection, StringDictionary)



# Пример использования класса List

```
using System;  
using System.Collections.Generic;  
namespace ConsoleApplication1 {  
class Program {  
    static void Main() {  
        List<int> lint = new List<int>();  
        lint.Add( 5 ); lint.Add( 1 ); lint.Add( 3 );  
        lint.Sort();  
        int a = lint[2];  
        Console.WriteLine( a );  
        foreach ( int x in lint ) Console.Write( x + " ");  
    }  
}}
```



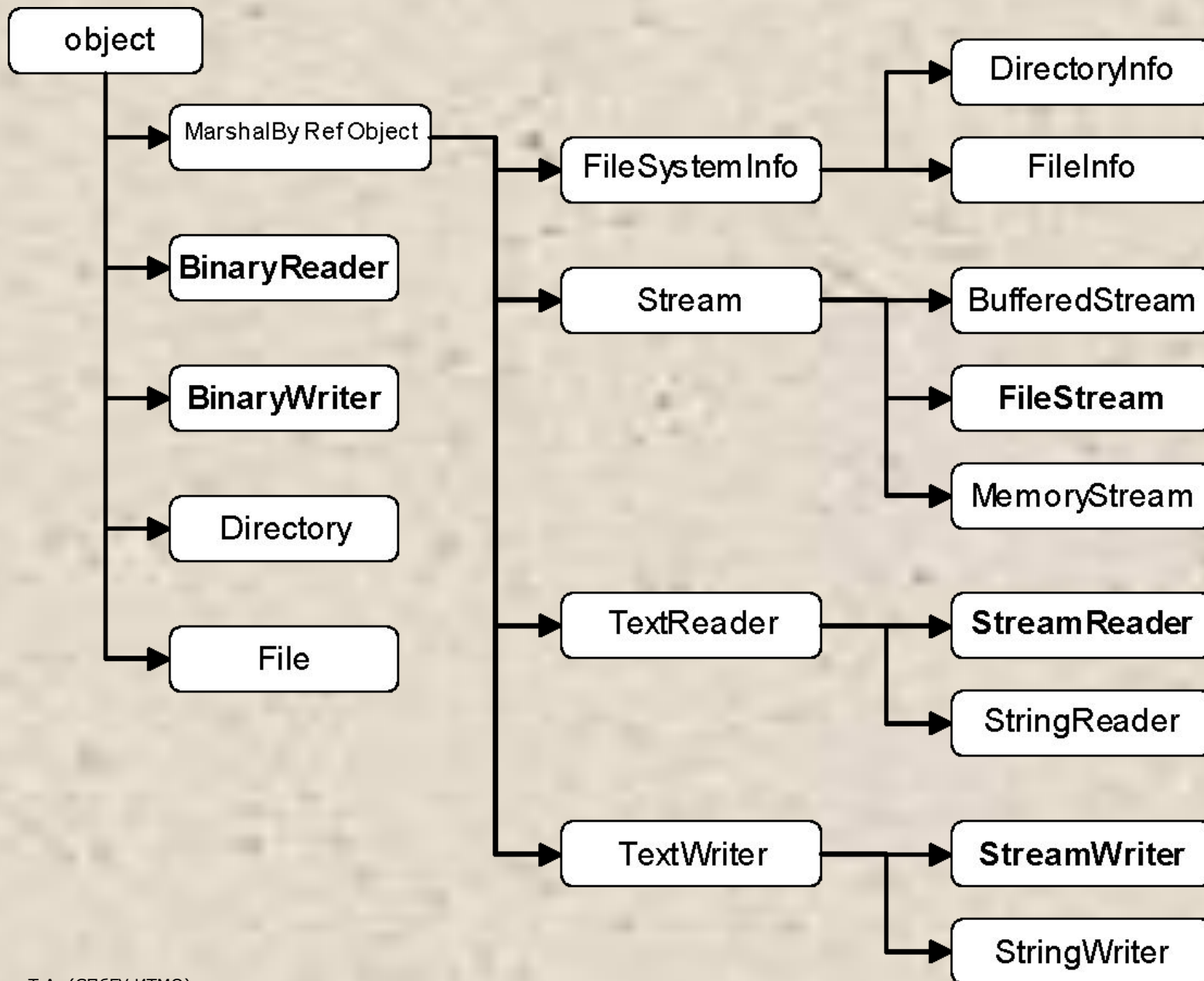
# Работа с файлами

---

# Общие принципы работы с файлами

- **Чтение** (*ввод*) — передача данных с внешнего устройства в оперативную память, обратный процесс — **запись** (*вывод*).
- Ввод-вывод в C# выполняется с помощью подсистемы ввода-вывода и классов библиотеки .NET. Обмен данными реализуется с помощью потоков.
- **Поток** (stream) — абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Потоки обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис.
- Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен.
- Обмен с потоком для повышения скорости передачи данных производится, как правило, через **буфер**. Буфер выделяется для каждого открытого файла.

# Классы .NET для работы с потоками



# Уровни обмена с внешними устройствами

Выполнять обмен с внешними устройствами можно на уровне:

- *двоичного представления данных*
  - (BinaryReader, BinaryWriter);
- *байтов*
  - (FileStream);
- *текста, то есть символов*
  - (StreamWriter, StreamReader).

# Доступ к файлам

- *Доступ к файлам может быть:*
  - **последовательным** - очередной элемент можно прочитать (записать) только после аналогичной операции с предыдущим элементом
  - *произвольным, или **прямым**, при котором выполняется чтение (запись) произвольного элемента по заданному адресу.*
- Текстовые файлы позволяют выполнять только последовательный доступ, в двоичных и байтовых потоках можно использовать оба метода.
- Прямой доступ в сочетании с отсутствием преобразований обеспечивает высокую скорость получения нужной информации.

# Пример чтения из текстового файла

```
static void Main() // весь файл -> в одну строку
{
    try
    {
        StreamReader f = new StreamReader( "text.txt" );
        string s = f.ReadToEnd();
        Console.WriteLine(s);
        f.Close();
    }
    catch( FileNotFoundException e )
    {
        Console.WriteLine( e.Message );
        Console.WriteLine( " Проверьте правильность имени файла!" );
        return;
    }
    catch
    {
        Console.WriteLine( " Неопознанное исключение!" );
        return;
    }
}
```

# Построчное чтение текстового файла

```
StreamReader f = new StreamReader( "text.txt" );  
    string s;  
    long i = 0;  
  
    while ( ( s = f.ReadLine() ) != null )  
        Console.WriteLine( "{0}: {1}", ++i, s );  
    f.Close();
```



# Чтение чисел из текстового файла – вар. 1

```
try {  
    List<int> list_int = new List<int>();  
    StreamReader file_in = new StreamReader( @"D:\FILES\1024" );  
    Regex regex = new Regex( "[^0-9-+]+" );  
    List<string> list_string = new List<string>(  
        regex.Split( file_in.ReadToEnd().TrimStart(' ') ) );  
    foreach (string temp in list_string)  
        list_int.Add( Convert.ToInt32(temp) );  
  
    foreach (int temp in list_int) Console.WriteLine(temp);  
    ...  
}  
catch (FileNotFoundException e)  
    { Console.WriteLine("Нет файла" + e.Message); return; }  
catch (FormatException e)  
    { Console.WriteLine(e.Message); return; }  
catch { ... }
```

# Чтение чисел из текстового файла – вар. 2

```
try {  
    StreamReader file_in = new StreamReader( @"D:\FILES\1024" );  
    char[] delim = new char[] { ' ' };  
    List<string> list_string = new List<string>(   
        file_in.ReadToEnd().Split( delim,   
            StringSplitOptions.RemoveEmptyEntries ) );  
    List<int> list_int = list_string.ConvertAll<int>(Convert.ToInt32);  
    foreach ( int temp in list_int ) Console.WriteLine( temp );  
    ...  
}  
catch (FileNotFoundException e)  
    { Console.WriteLine("Нет файла" + e.Message); return; }  
catch (FormatException e)  
    { Console.WriteLine(e.Message); return; }  
catch { ... }
```