# Обработка исключений
# Работа с файлами и каталогами

Занятие 1

# Обработка исключений

# Типы возможных аномалий

Программные ошибки (bugs). Так обычно называются ошибки, которые допускает программист.

Пользовательские ошибки (user errors). В отличие от программных ошибок, пользовательские ошибки обычно возникают из-за тех, кто запускает приложение, а не тех, кто его создает.

Исключения (exceptions). Исключениями, или исключительными ситуациями, обычно называются аномалии, которые могут возникать во время выполнения и которые трудно, а порой и вообще невозможно, предусмотреть во время программирования приложения.

# Генерация общего исключения

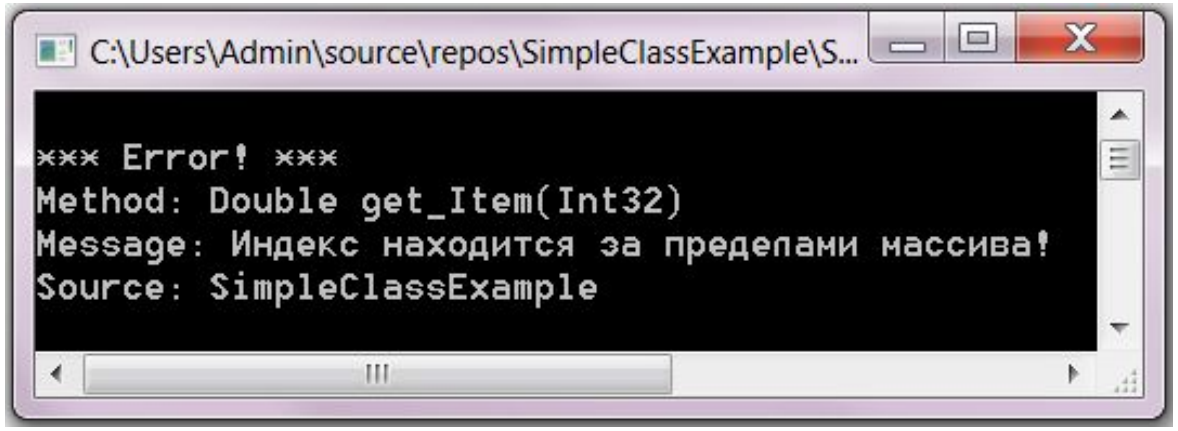
- Вспомним индексатор в классе Vector

```
public double this[int index]
{
    get
    {
        if (index >= 0 && index < 3)
            return vector[index];
        return 0;
    }
    set
    {
        if (index >= 0 && index < 3)
            vector[index] = value;
    }
}
```

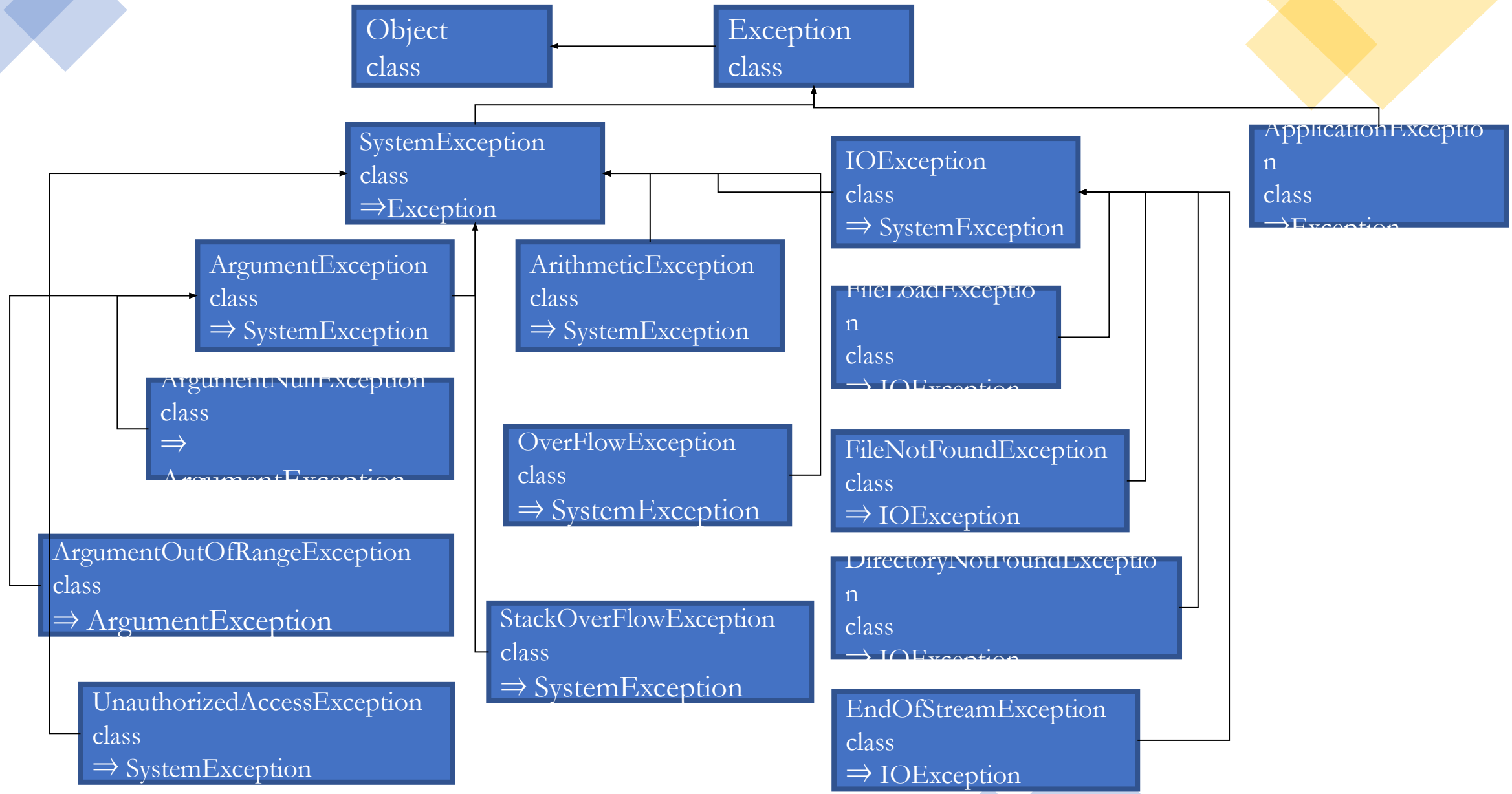
```
public double this[int index]
{
    get
    {
        if (index >= 0 && index < 3)
            return vector[index];
        throw new Exception(
       "Индекс находится за" +
       "пределами массива!");
    }
    set
    {
        if (index >= 0 && index < 3)
            vector[index] = value;
        else
            throw new Exception(
                "Индекс находится за" +
            "пределами массива!");
    }
}
```

# Перехват исключений

```
static void Main(string[] args)
{
    Vector v1 = new Vector(1, 0, 1);
    try
    {
        Console.WriteLine(v1[3]);
    }
    catch (Exception exc)
    {
        Console.WriteLine("\n*** Error! ***"); // ошибка
        Console.WriteLine("Method: {0}", exc.TargetSite); // метод
        Console.WriteLine("Message: {0}", exc.Message); // сообщение
        Console.WriteLine("Source: {0}", exc.Source); // источник
        Console.ReadKey();
    }
}
```




- Оператор try указывает блок кода, предназначенный для обработки ошибок или очистки.
- Блок catch имеет доступ к объекту Exception, который содержит информацию об ошибке. Блок catch применяется либо для компенсации последствий ошибки, либо для *повторной генерации* исключения.


Object
class
Exception
class
SystemException
class
⇒Exception
IOException
class
⇒ SystemException
ApplicationExceptio
n
class
ArgumentException
class
⇒ SystemException
ArithmeticException
class
⇒ SystemException
FileLoadExceptio
n
class
ArgumentNullException
class
⇒
OverFlowException
class
⇒ SystemException
FileNotFoundException
class
⇒ IOException
ArgumentOutOfRangeException
class
⇒ ArgumentException
DirectoryNotFoundExceptio
n
class
StackOverFlowException
class
⇒ SystemException
UnauthorizedAccessException
class
⇒ SystemException
EndOfStreamException
class
⇒ IOException

# Пользовательски е исключения.

- Если не устраивают встроенные исключения можно создать свой тип исключений.
- Например, мы хотим ограничить модуль векторов: 0>V.Length>=10

Exception Unhandled

**SimpleClassExample.VectorIsOutOfRangeException:** 'Vector Error Message: Вектор не может иметь длину больше 10!'

View Details | Copy Details | Start Live Share session...

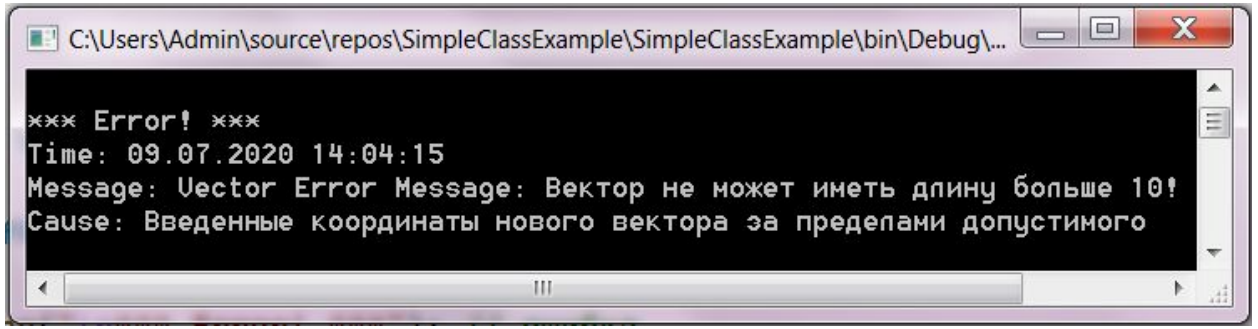
Exception Settings

```
class VectorIsOutOfRangeException : ApplicationException
{
    private string messageDetails = String.Empty;
    1 reference
    public DateTime ErrorTimeStamp { get; set; }
    1 reference
    public string CauseOfError { get; set; }
    0 references
    public VectorIsOutOfRangeException() { }
    0 references
    public VectorIsOutOfRangeException(string message,
        string cause, DateTime time)
    {
        messageDetails = message;
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
    // Переопределение свойства Exception.Message.
    0 references
    public override string Message
    {
        get
        {
            return string.Format("Vector Error Message: {0}
                                 messageDetails);
        }
    }
}
```

# Генерация и перехват исключения

```
public Vector(double p1, double p2, double p3)
{
    if (Math.Sqrt(p1 * p1 +
            p2 * p2 + p3 * p3)>10)
        throw new VectorIsOutOfRangeException(
            "Вектор не может иметь длину больше 10!",
            "Введенные координаты нового вектора за " +
            "пределами допустимого", DateTime.Now);
    vector = new double[3];
    vector[0] = p1;
    vector[1] = p2;
    vector[2] = p3;
}
```

```
try
{
    Vector v2 = new Vector(7, 6, 5);
}
catch (VectorIsOutOfRangeException ve)
{
    Console.WriteLine("\n*** Error! ***"); // ошибка
    Console.WriteLine("Time: {0}", ve.ErrorTimeStamp); // время
    Console.WriteLine("Message: {0}", ve.Message); // сообщение
    Console.WriteLine("Cause: {0}", ve.CauseOfError); //причина
    Console.ReadKey();
}
```



- Во многих случаях роль специального исключения состоит не в предоставлении дополнительной функциональности помимо той, что унаследована от базовых классов, а в обеспечении строго именованного типа, четко описывающего природу ошибки и тем самым позволяющего клиенту использовать разную логику обработки для разных типов исключений. Тогда, переопределять виртуальное свойство Message вовсе не требуется, поскольку можно также просто передавать поступающее сообщение конструктору родителя

```
class VectorIsOutOfRangeException : ApplicationException
{
    private string messageDetails = String.Empty;
    2 references
    public DateTime ErrorTimeStamp { get; set; }
    2 references
    public string CauseOfError { get; set; }
    0 references
    public VectorIsOutOfRangeException() { }
    // Передача сообщения конструктору родителя.
    1 reference
    public VectorIsOutOfRangeException(string message,
        string cause, DateTime time)
        : base(message)
    {
        CauseOfError = cause;
        ErrorTimeStamp = time;
    }
}
```

Exception Unhandled

**SimpleClassExample.VectorIsOutOfRangeException:** 'Вектор не может иметь длину больше 10!'

View Details | Copy Details | Start Live Share session...

Exception Settings

# Обработка многочисленных исключений

```
public static Vector operator /(Vector v, double d)
{
    if (d == 0) throw new DivideByZeroException(
        "Деление вектора на ноль не может быть выполнено!");
    return new Vector(v[0] / d, v[1] / d, v[2] / d);
}
```

```
Vector v1 = new Vector(2, 1, 2);
for (double d = 1.0d; d >= 0; d -= 0.25d)
{
    Console.WriteLine("v1/{0} = {1}", d, v1 / d);
}
```

```
try
{
    for (double d = 1.0d; d >= 0; d -= 0.25d)
    {
        Console.WriteLine("v1/{0} = {1}", d, v1 / d);
    }
}
catch (VectorIsOutOfRangeException ve)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", ve.Message);
}
catch (DivideByZeroException de)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", de.Message);
}
```

```
//Этот код компилироваться не будет!
Vector v1 = new Vector(1, 1, 1);
try
{
    for (double d = 1.0d; d >= 0; d -= 0.25d)
    {
        Console.WriteLine("v1/{0} = {1}", d, v1 / d);
    }
}
catch (Exception e)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", e.Message);
}
catch (VectorIsOutOfRangeException ve)
{
    Con
    Con
}
catch (DivideByZeroException de)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", de.Message);
}
```

class SimpleClassExample.VectorIsOutOfRangeException

A previous catch clause already catches all exceptions of this or of a super type ('Exception')

```
//Теперь все нормально
Vector v1 = new Vector(1, 1, 1);
try
{
    for (double d = 1.0d; d >= 0; d -= 0.25d)
    {
        Console.WriteLine("v1/{0} = {1}", d, v1 / d);
    }
}
catch (VectorIsOutOfRangeException ve)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", ve.Message);
}
catch (DivideByZeroException de)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", de.Message);
}
catch (Exception e)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", e.Message);
}
```

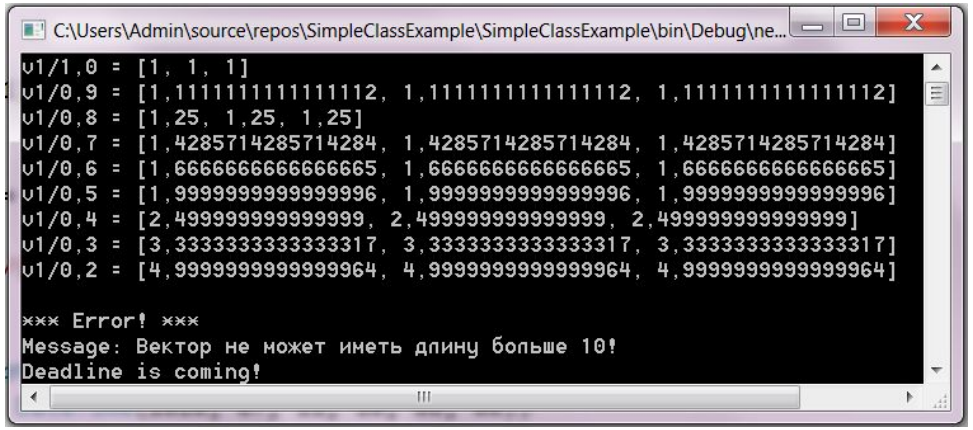
# Общие операторы catch

```
Vector v1 = new Vector(1, 1, 1);
try
{
    for (double d = 1.0d; d >= 0; d -= 0.25d)
    {
        Console.WriteLine("v1/{0} = {1}", d, v1 / d);
    }
}
catch
{
    Console.WriteLine("Что-то плохое случилось!");
}
```

# Фильтры исключений (C# 6)

```
try
{
    for (double d = 1.0d; d >= 0; d -= 0.1d)
    {
        Console.WriteLine("v1/{0:0.0} = {1}", d, v1 / d);
    }
}
catch (VectorIsOutOfRangeException ve)
when (ve.ErrorTimeStamp < new DateTime(2020, 07, 11, 10, 00, 00))
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", ve.Message);
    Console.WriteLine("Deadline is coming!");
}
catch (VectorIsOutOfRangeException ve)
when (ve.ErrorTimeStamp > new DateTime(2020, 07, 11, 10, 00, 00))
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", ve.Message);
    Console.WriteLine("Deadline has passed...");
}
```

Если генерируется исключение **VectorIsOutOfRangeException,** тогда вычисляется булевское выражение, находящееся после ключевого слова **when.** Если результатом оказывается **false,** то данный блок **catch** игнорируется и просматриваются любые последующие конструкции **catch.**



# Передача исключений

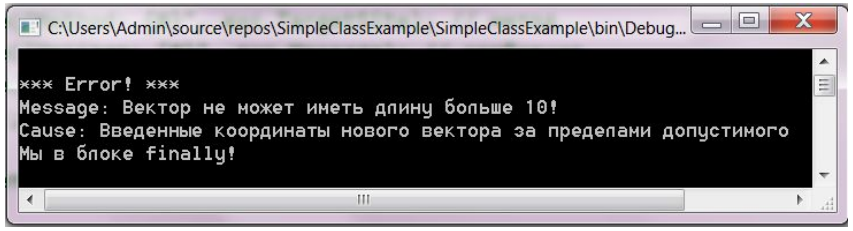
```
try
{
    // Логика, которая может привести
    // к созданию вектора, длиной > 10...
}
catch (VectorIsOutOfRangeException ve)
{
    // Выполнение любой частичной обработки данной ошибки
    //и передача дальнейшей ответственности.
    throw;
}
```

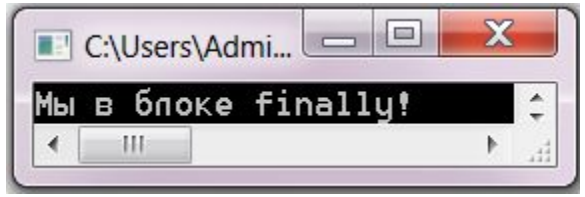
- Исключение можно перехватывать и генерировать повторно

# Блок finally

```
// Исключение генерируется и поймано
try
{
    Vector v2 = new Vector(7, 6, 5);
}
catch (VectorIsOutOfRangeException ve)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", ve.Message);
    Console.WriteLine("Cause: {0}", ve.CauseOfError);
}
finally
{
    Console.WriteLine("Мы в блоке finally!");
}
```

```
// Исключение не генерируется
try
{
    Vector v2 = new Vector(1, 2, 3);
}
catch (VectorIsOutOfRangeException ve)
{
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Message: {0}", ve.Message);
    Console.WriteLine("Cause: {0}", ve.CauseOfError);
}
finally
{
    Console.WriteLine("Мы в блоке finally!");
}
```




- Блок finally выполняется всегда — независимо от того, возникало ли исключение, и полностью ли был выполнен блок try. Блоки finally обычно используются для размещения кода очистки

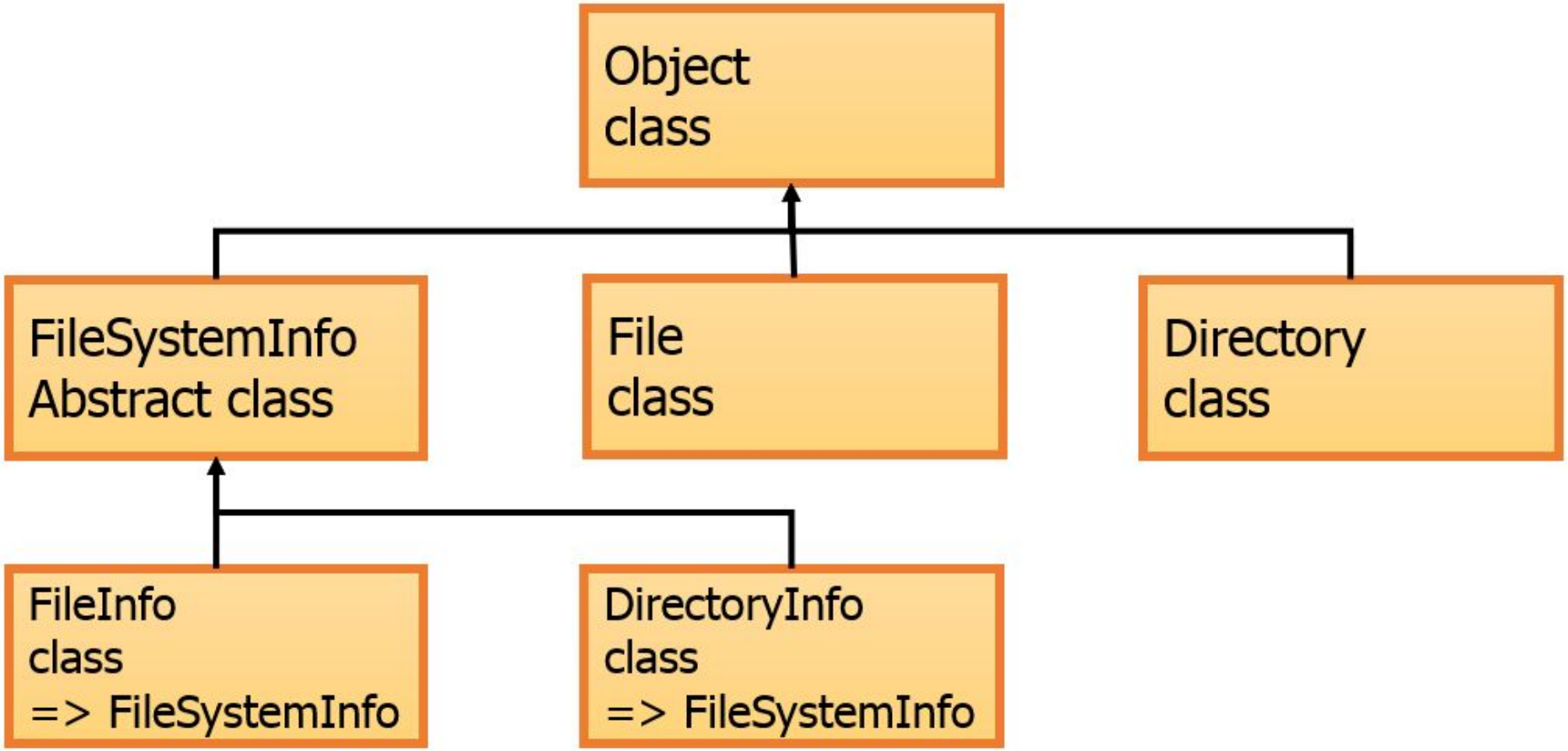
# Операции с файлами и каталогами

# Замечание!

- Про ***потоки данных***, которые обычно применяются для чтения и записи напрямую в файлы и сетевые подключения и могут соединяться в цепочки либо помещаться внутрь декорированных потоков с целью добавления функциональности сжатия или шифрования мы поговорим на отдельном занятии, сейчас просто насколько слов о работе с файлами и каталогами.

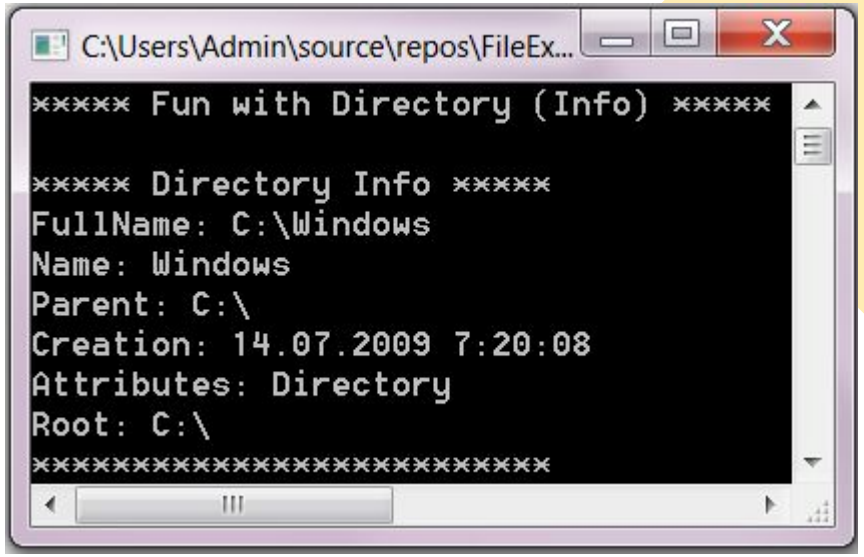
# System.IO

- Пространство имен **System.IO** предоставляет набор типов для разнообразной работы с файлами и папками.
- Для большинства средств можно выбирать из двух классов: один предлагает статические методы, а другой — методы экземпляра.
  - Статические классы
    - **File**
    - **Directory**
  - Классы с методами экземпляра (сконструированного с указанием имени файла или каталога)
    - **FileInfo**
    - **DirectoryInfo**
- Вдобавок имеется статический класс по имени **Path.** Он ничего не делает с файлами или каталогами, а предоставляет методы строкового манипулирования для имен файлов и путей к каталогам. Класс **Path** также помогает при работе с временными файлами.


Object
class
FileSystemInfo
Abstract class
File
class
Directory
class
FileInfo
class
=> FileSystemInfo
DirectoryInfo
class
=> FileSystemInfo

# Пример работы с классом DirectoryInfo

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Directory (Info) *****\n");
    ShowWindowsDirectoryInfo();
    Console.ReadLine();
}
1 reference
static void ShowWindowsDirectoryInfo()
{
    // Вывести информацию о каталоге.
    DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
    Console.WriteLine("***** Directory Info *****");
    Console.WriteLine("FullName: {0}", dir.FullName); // полное имя
    Console.WriteLine("Name: {0}", dir.Name); // имя каталога
    Console.WriteLine("Parent: {0}", dir.Parent); // родительский каталог
    Console.WriteLine("Creation: {0}", dir.CreationTime);//время создания
    Console.WriteLine("Attributes: {0}", dir.Attributes); // атрибуты
    Console.WriteLine("Root: {0}", dir.Root); // корневой каталог
    Console.WriteLine("************************* \n");
}
```



# Метод GetFiles()

```
***** Fun with Directory (Info)

Found 32 *.jpg files

*********************
File name: img13.jpg
File size: 1231580
Creation: 14.07.2009 1:48:53
 Attributes: Archive
 *********************
*********************
File name: img14.jpg
File size: 1526083
Creation: 14.07.2009 1:48:53
 Attributes: Archive
 *********************
*********************
File name: img15.jpg
File size: 1492201
Creation: 14.07.2009 1:48:53
 Attributes: Archive
 *********************
*********************
File name: img16.jpg
```

```
static void DisplayImageFiles()
{
    DirectoryInfo dir = new DirectoryInfo(@"C:\Windows\Web\Wallpaper");
    // Получить все файлы с расширением *.jpg.
    FileInfo[] imageFiles = dir.GetFiles("*.jpg",
        SearchOption.AllDirectories);
    // Сколько файлов найдено?
    Console.WriteLine("Found {0} *.jpg files\n", imageFiles.Length);
    // Вывести информацию о каждом файле.
    foreach (FileInfo f in imageFiles)
    {
        Console.WriteLine("********************");
        Console.WriteLine("File name: {0}", f.Name); // имя
        Console.WriteLine("File size: {0}", f.Length); // размер
        Console.WriteLine("Creation: {0}", f.CreationTime); //время созд.
        Console.WriteLine(" Attributes: {0}", f.Attributes); // атрибуты
        Console.WriteLine(" ********************");
    }
}
```

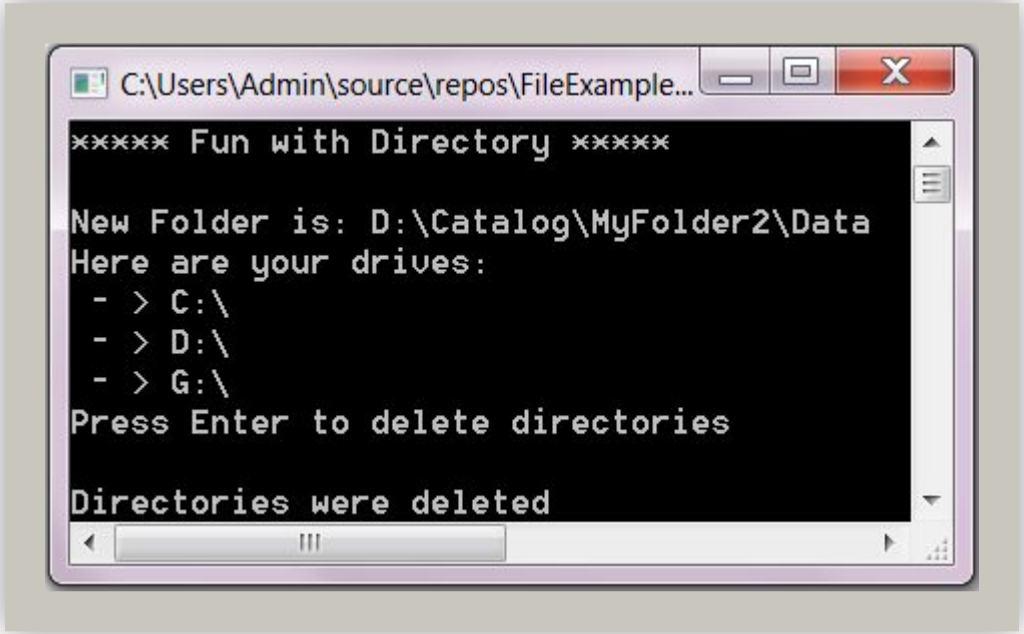
# Метод CreateSubdirectory()

```
static void AppDirectory()
{
    DirectoryInfo dir = new DirectoryInfo(@"D:\Catalog");
    // Создать \MyFolder в начальном каталоге.
    dir.CreateSubdirectory("MyFolder");
    // Получить возвращенный объект DirectoryInfo.
    DirectoryInfo myDataFolder = dir.CreateSubdirectory
        (@"MyFolder2\Data");
    // Напечатать путь ..\MyFolder2\Data.
    Console.WriteLine("New Folder is: {0}", myDataFolder);
}
```

# Пример работы с классом Directory

```
static void WorkWithDirectoryType()
{
    // Перечислить все дисковые устройства данного компьютера.
    string[] drives = Directory.GetLogicalDrives();
    Console.WriteLine("Here are your drives:");
    foreach (string s in drives)
        Console.WriteLine(" - > {0} ", s);
    // Удалить то, что было ранее создано.
    Console.WriteLine("Press Enter to delete directories");
    Console.ReadLine();
    try
    {
        Directory.Delete(@"D:\Catalog\MyFolder");
        // Второй параметр указывает,
        // нужно ли удалять все подкаталоги.
        Directory.Delete(@"D:\Catalog\MyFolder2", true);
        Console.WriteLine("Directories were deleted");
    }
    catch (IOException e)
    {
        Console.WriteLine(e.Message);
    }

}
```



# Работа с классом FileInfo

| Метод | Назначение |
|---|---|
| `ReadAllBytes()` | Открывает указанный файл, возвращает двоичные данные в виде массива байт и затем закрывает файл |
| `ReadAllLines()` | Открывает указанный файл, возвращает символьные данные в виде массива строк, затем закрывает файл |
| `ReadAllText()` | Открывает указанный файл, возвращает символьные данные в виде System.String(), затем закрывает файл |
| `WriteAllBytes()` | Открывает указанный файл, записывает в него байтовый массив и закрывает файл |
| `WriteAllLines()` | Открывает указанный файл, записывает в него массив строк и закрывает файл |
| `WriteAllText()` | Открывает указанный файл, записывает в него данные из указанной строки и закрывает файл |

# Работа с типом File

| Метод | Назначение |
|---|---|
| `ReadAllBytes()` | Открывает указанный файл, возвращает двоичные данные в виде массива байт и затем закрывает файл |
| `ReadAllLines()` | Открывает указанный файл, возвращает символьные данные в виде массива строк, затем закрывает файл |
| `ReadAllText()` | Открывает указанный файл, возвращает символьные данные в виде System.String(), затем закрывает файл |
| `WriteAllBytes()` | Открывает указанный файл, записывает в него байтовый массив и закрывает файл |
| `WriteAllLines()` | Открывает указанный файл, записывает в него массив строк и закрывает файл |
| `WriteAllText()` | Открывает указанный файл, записывает в него данные из указанной строки и закрывает файл |

- **File** — это статический класс, все методы которого принимают имя файла. Имя файла может или указываться относительно текущего каталога, или быть полностью определенным, включая каталог. На слайде перечислены некоторые методы класса **File** (все они являются **public** и **static)**

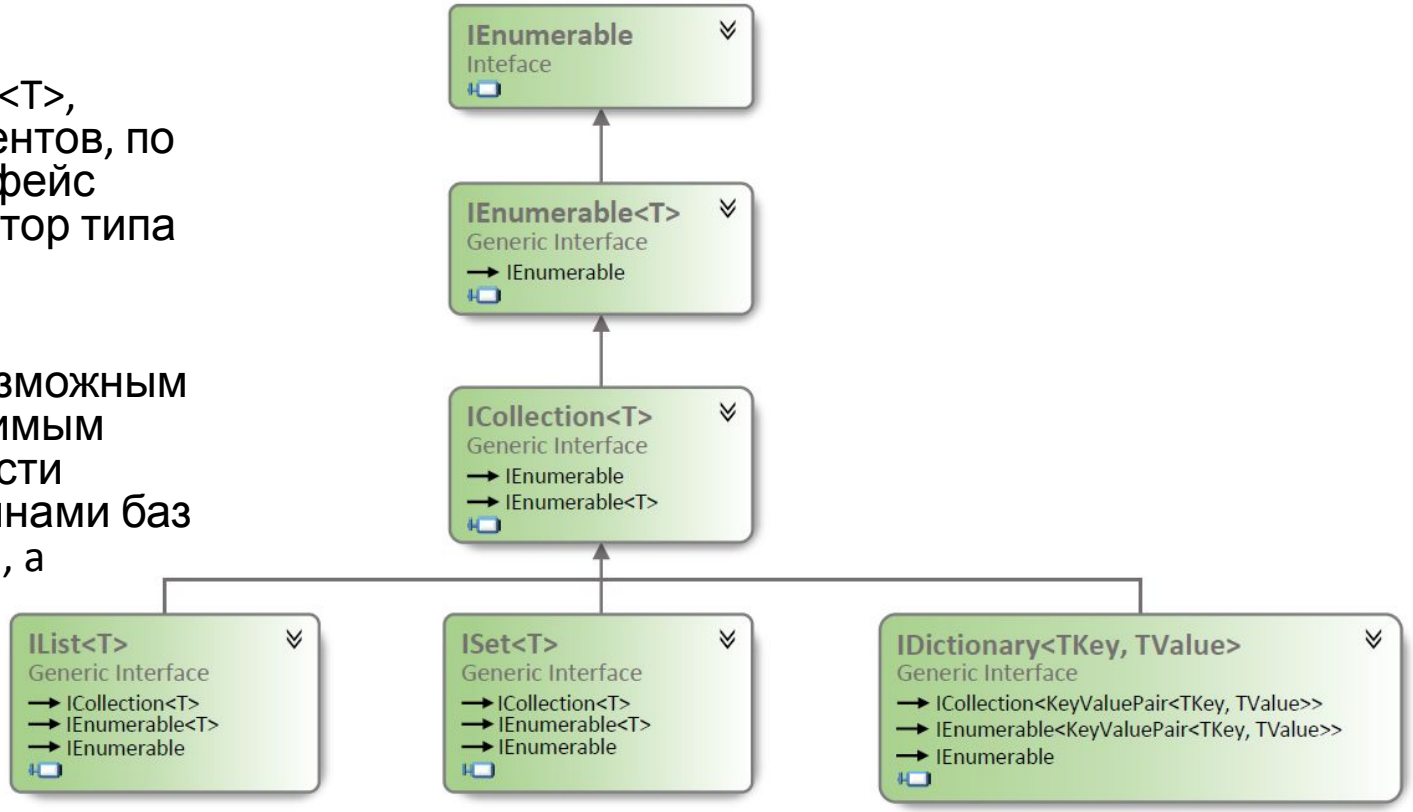
# Несколько слов об обобщенных коллекциях

# О чем эта часть.

- В .NET доступно много обобщенных коллекций и со временем их количество растет. В данной части презентации приведены наиболее важные интерфейсы и классы обобщенных коллекций, о существовании которых необходимо знать. И которые вы будете использоватью В пространствах имен System.Collections, System.Collections.Specialized и System.ComponentModel определены дополнительные необобщенные коллекции, но здесь они не рассматриваются.

# Интерфейсы

- Почти все интерфейсы, которые нужно знать, находятся в пространстве имен System.Collections.Generic. Справа показано, как были связаны основные интерфейсы до выхода версии .NET 4.5; здесь также присутствует необобщенный IEnumerable в качестве корневого интерфейса. Чтобы излишне не усложнять диаграмму, в нее не были включены интерфейсы, предназначенные только для чтения, которые появились в версии .NET 4.5.
- Наиболее фундаментальным интерфейсом обобщенной коллекции является IEnumerable<T>, представляющий последовательность элементов, по которой можно осуществлять проход. Интерфейс IEnumerable<T> позволяет запрашивать итератор типа IEnumerator<T>.
- Отделение поддерживающей проход последовательности от итератора делает возможным выполнение множества итераторов независимым образом на одной и той же последовательности одновременно. Если вы хотите думать терминами баз данных, то IEnumerable<T> является таблицей, а IEnumerator<T> — курсором.



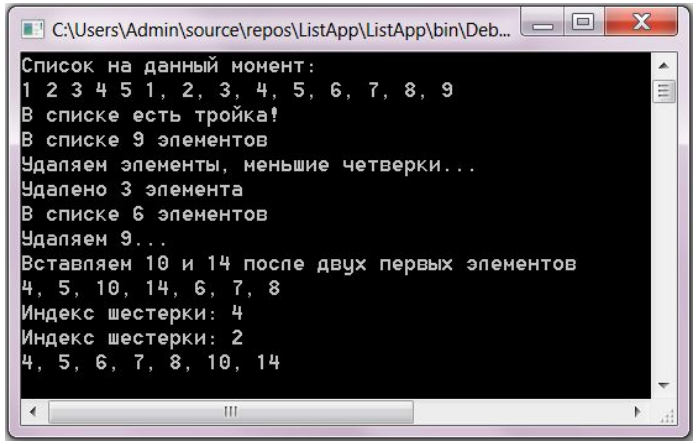
# Интерфейсы обобщенных коллекций

| Интерфейс | Описание |
|---|---|
| **ICollection<T>** | Определяет основополагающие свойства обобщенных коллекций |
| **IComparer<T>** | Определяет обобщенный метод Compare () для сравнения объектов, хранящихся в коллекции |
| **IDictionary<Tkey, TValue>** | Определяет обобщенную коллекцию, состоящую из пар "ключ-значение" |
| **IEnumerable<T>** | Определяет обобщенный метод GetEnumerator (), предоставляющий перечислитель для любого класса коллекции |
| **Enumerator<T>** | Предоставляет методы, позволяющие получать содержимое коллекции по очереди |
| **IEqualityComparer<T>** | Сравнивает два объекта на предмет равенства |
| **IList<T>** | **Определяет обобщенную коллекцию, доступ к которой можно получить с помощью индексатора** |

# Класс List<T>

```
//самая простая инициализация списка
List<double> myList = new List<double>();
//инициализация списка с назначением свойства capacity
myList = new List<double>(15);
//инициализация списка с присваиванием значений элементам
myList = new List<double>() { 1.0d, 2.0d, 3.0d, 4.0d, 5.0d };
//инициализация списка с помощью массива
double[] darr = new double[] { 1.0d, 2.0d, 3.0d, 4.0d, 5.0d };
myList = new List<double>(darr);
//печать списка с помощью оператора foreach
Console.WriteLine("Список на данный момент:");
foreach (double d in myList)
    Console.Write("{0} ", d);
//Добавление элемента
myList.Add(6.0d);
//Добавление массива элементов
myList.AddRange(new double[] { 7.0d, 8.0d, 9.0d });
//печать элементов из списка (пользуемся статическим
//методом строки Join)
Console.WriteLine(string.Join(", ",myList));
//Проверка, все ли элементы удовлетворяют условию
if (myList.All(elem => elem > 3))
    Console.WriteLine("В списке все числа, больше тройки!");
```

```
//Проверка, есть ли элементы, удовлетворяющие условию
if (myList.Any(elem => elem < 0))
    Console.WriteLine("В списке есть отрицательные числа!");
//Проверка, есть ли в списке нужный элемент
if (myList.Contains(3.0d))
    Console.WriteLine("В списке есть тройка!");
//Свойство count  - количество элементов в списке
Console.WriteLine("В списке {0} элементов", myList.Count);
Console.WriteLine("Удаляем элементы, меньшие четверки...");
//Удаляем элементы, удовлетворяющие условию
Console.WriteLine("Удалено {0} элемента", myList.RemoveAll(elem => elem < 4));
Console.WriteLine("В списке {0} элементов", myList.Count);
Console.WriteLine("Удаляем 9...");
myList.Remove(9.0d);
Console.WriteLine("Вставляем 10 и 14 после двух первых элементов");
myList.InsertRange(2, new double[] { 10.0d, 14.0d });
Console.WriteLine(string.Join(", ", myList));
//Проводим поиск индекса элемента в неотсортированном списке
Console.WriteLine("Индекс шестерки: {0}", myList.IndexOf(6.0d));
//Сортируем список
myList.Sort();
//Проводим поиск индекса элемента в отсортированном списке
Console.WriteLine("Индекс шестерки: {0}", myList.BinarySearch(6.0d));
Console.WriteLine(string.Join(", ", myList));
```



# Структура KeyValuePair<TKey, TValue>

В пространстве имен `System.Collections.Generic` определена структура `KeyValuePair<TKey, TValue>` Она служит для хранения ключа и его значения и применяется в классах обобщенных коллекций, в которых хранятся пары "ключ-значение", как, например, в классе `Dictionary<TKey, TValue>`. В этой структуре определяются два следующих свойства.

```
public TKey Key { get; }
public TValue Value { get; }
```

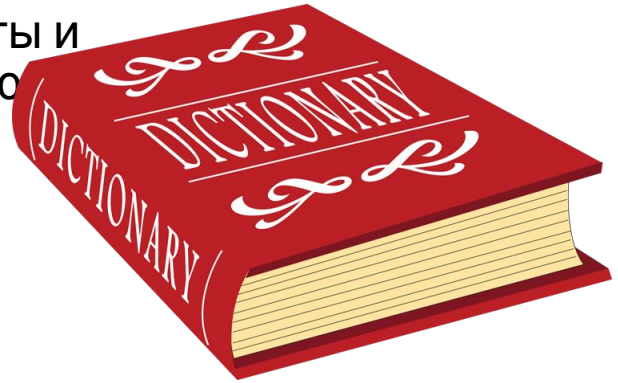
В этих свойствах хранятся ключ и значение соответствующего элемента коллекции. Для построения объекта типа KeyValuePair<TKey, TValue> служит конструктор:

```
public KeyValuePair(TKey key, TValue value)
```

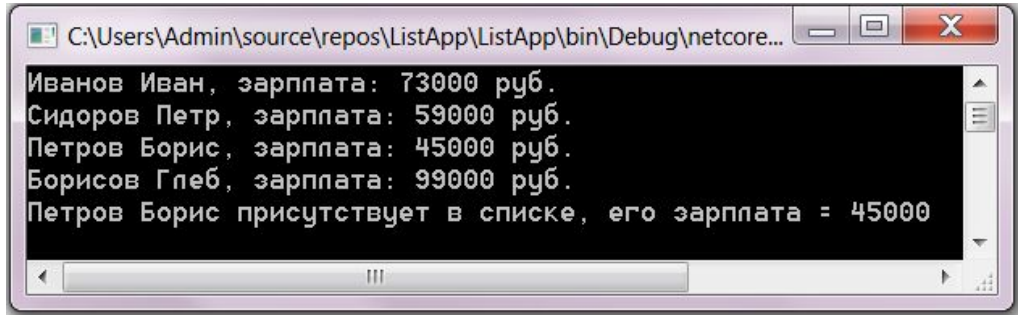
где key обозначает ключ, а value — значение.

# Класс Dictionary<TKey, TValue>

- Обобщенный класс **Dictionary** — одна из наиболее часто применяемых коллекций. Позволяет хранить пары "ключ-значение" в коллекции как в словаре. Значения доступны в словаре по соответствующим ключам.
- Для хранения ключей и значений он использует структуру данных в форме хеш-таблицы, а также характеризуется высокой скоростью работы и эффективностью

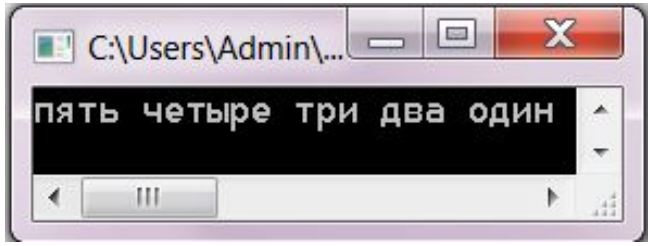


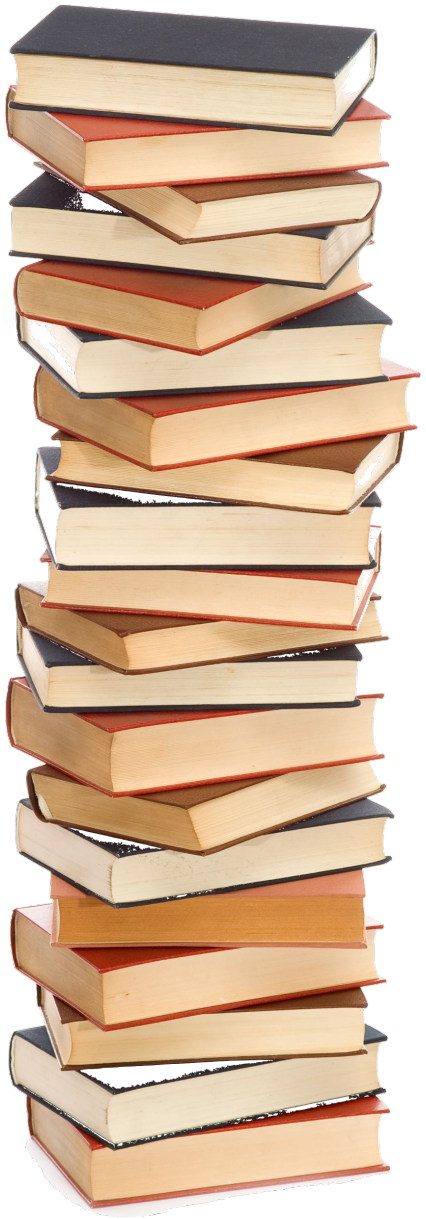

```
// Создать словарь для хранения имен и фамилий
// работников и их зарплаты.
Dictionary<string, double> currency =
new Dictionary<string, double>();
// Добавить элементы в коллекцию,
currency.Add("Иванов Иван", 73000);
currency.Add("Сидоров Петр", 59000);
currency.Add("Петров Борис", 45000);
currency.Add("Борисов Глеб", 99000);
// Получить коллекцию ключей, т.е. фамилий и имен.
ICollection<string> c = currency.Keys;
// Использовать ключи для получения значений, т.е. зарплаты,
foreach (string str in c)
    Console.WriteLine("{0}, зарплата: {1} руб.", str, currency[str]);
if (currency.ContainsKey("Петров Борис"))
    Console.WriteLine($"Петров Борис присутствует в списке, " +
        $"его зарплата = {currency["Петров Борис"]}");
else Console.WriteLine("Петров Борис отсутствует в списке");
```



# Класс Stack<T>

```
Stack<string> st = new Stack<string>();
st.Push("один");
st.Push("два");
st.Push("три");
st.Push("четыре");
st.Push("пять");
while (st.Count > 0)
{
    string str = st.Pop();
    Console.Write(str + " ");
}
Console.WriteLine();
```



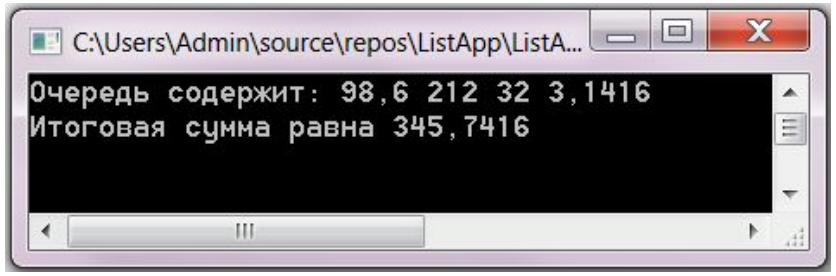



- Класс Stack<T> является обобщенным эквивалентом класса необобщенной коллекции Stack. В нем поддерживается стек в виде списка, действующего по принципу "первым пришел — последним обслужен".

# Класс Queue<T>

- В классе Queue<T> поддерживается очередь в виде списка, действующего по принципу "первым пришел — первым обслужен".

```
Queue<double> q = new Queue<double>();
q.Enqueue(98.6);
q.Enqueue(212.0);
q.Enqueue(32.0);
q.Enqueue(3.1416);
double sum = 0.0;
Console.Write("Очередь содержит: ");
while (q.Count > 0)
{
    double val = q.Dequeue();
    Console.Write(val + " ");
    sum += val;
}
Console.WriteLine("\nИтоговая сумма равна " + sum);
```



# Домашнее задание. Часть 1.

- Создать класс Matrix, описывающий квадратную матрицу 2x2.
- Добавить вычисляемое свойство – определитель.
- Реализовать двумерный индексатор.
- Реализовать метод нахождения обратной матрицы.
- Переопределить ToString().
- Переопределить в классе операции «+, ++, -, --, *, /, <, > (по определителю), ==, !=».
- Написать функции Parse (берет строку надлежащего формата и переводит ее в матрицу, если формат неверный, выбрасывает исключение) и TryParse (берет строку надлежащего формата и и пытается перевести ее в матрицу, если формат неверный, то возвращает false, в out параметр записывает null, если формат верный возвращает true, в out параметр записывает считанную матрицу). Правило записи матрицы в строку определите сами, хорошо бы она соотносилась с ToString().
  - Сигнатура Parse: Matrix Parse(string str);
  - Сигнатура TryParse: bool TryParse(string str, out Matrix matr);

# Домашнее задание. Часть 2.

- Создать класс ListOfMatrix, работающий со списком (List) матриц. Список можно реализовать как List<Matrix>. Список должен быть закрытым полем. Определить в классе индексатор, метод сортировки (можно вызывать встроенные Array.Sort или List.Sort), определения первого и последнего элементов, добавление элемента, нахождение минимального и максимального элементов (по определителю), преобразования к массиву матриц (ToArray()), вывод элементов в массив строк.
- Создать статический класс MatrixInOut, содержащий функции считывания списка матриц из файла и записи матриц в файл. Перехватывать исключения при неверной работе с файлами, выходе за пределы массива, деления на 0 (при нахождении обратной).
- Создать консольное приложение для работы с написанными ранее классами. Предусмотреть возможность ввода списка матриц с консоли, из файла. Вывод списка на консоль, вывод информации по списку. Вывод всех матриц, определитель которых меньше введенного числа. Вывод отсортированного списка.