



## Урок 4

# Объектно- ориентированное программирование. Часть 4

Списки. Обобщенные списки. Лямбда-выражения. Linq.

[Generic. Collection. Необобщенные коллекции](#)

[Обобщенные коллекции](#)

[Список обобщенных коллекций](#)

[Класс List<T>](#)

[Структура KeyValuePair<TKey, TValue>](#)

[Класс Dictionary<TKey, TValue>](#)

[Класс Queue<T>](#)

[Класс Stack<T>](#)

[Инициализация коллекции](#)

[Перебор элементов коллекции без привязки к ее реализации](#)

[Изменение порядка элементов массива на обратный](#)

[Первый способ – самостоятельно](#)

[Второй способ – с использованием метода расширения Reverse](#)

[Извлечение уникальных элементов из коллекции](#)

[Лямбда-выражения](#)

[Использование преимуществ контравариантности](#)

[Пример метода расширений](#)

[Еще примеры метода расширений](#)

[Linq](#)

[Простой Linq-запрос](#)

[Два where \(условия\)](#)

[Еще один пример с where](#)

[Демонстрация OrderBy и преобразования в список](#)

[Демонстрация использования Linq с массивом пользовательских данных](#)

[Основные Linq](#)

[Практика](#)

[«Астероиды» с коллекциями](#)

[Новшества C# 7](#)

[Локальные функции](#)

[Локальная переменная-ссылка](#)

[Ссылка как результат функции](#)

[Pattern matching](#)

[Деконструкторы](#)

[Кортежи](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

# Generic. Collection. Необобщенные коллекции

Необобщенные коллекции вошли в состав среды .NET Framework еще в версии 1.0. Они определяются в пространстве имен **System.Collections**. Необобщенные коллекции представляют собой структуры данных общего назначения, оперирующие ссылками на объекты. Они позволяют манипулировать объектом любого типа, хотя и не типизированным способом. В этом их преимущество и недостаток одновременно. Благодаря тому, что необобщенные коллекции оперируют ссылками на объекты, в них можно хранить разнотипные данные. Это удобно в тех случаях, когда требуется манипулировать совокупностью разнотипных объектов, или же когда типы хранящихся в коллекции объектов заранее не известны. Но если коллекция предназначена для хранения объекта конкретного типа, необобщенные коллекции не обеспечивают типовую безопасность, которую можно обнаружить в обобщенных коллекциях.

Пример:

```
using System;
using System.Collections;
class Program
{
    static void Main(string[] args)
    {
        ArrayList list = new ArrayList();
        list.Add(1);
        list.Add(3.14);
        list.Add("Строка");
        list.Add(new int[] { 1,2,3});
        foreach (object element in list)
            Console.WriteLine(element);
        Console.ReadKey();
    }
}
```

Вывод программы:

```
1
3,14
Строка
System.Int32[]
```

## Обобщенные коллекции

Обобщенные объекты .Net Framework легко отличить по угловым скобкам после их названия:

- ▲ { } System.Collections.Generic
  - ISet<T>
  - ▷ LinkedList<T>
  - ▷ LinkedList<T>.Enumerator
  - ▷ LinkedListNode<T>
  - ▷ Queue<T>
  - ▷ Queue<T>.Enumerator
  - ▷ SortedDictionary<TKey, TValue>
  - ▷ SortedDictionary<TKey, TValue>.Enumerator
  - ▷ SortedDictionary<TKey, TValue>.KeyCollection
  - ▷ SortedDictionary<TKey, TValue>.KeyCollection.Enumerator
  - ▷ SortedDictionary<TKey, TValue>.ValueCollection
  - ▷ SortedDictionary<TKey, TValue>.ValueCollection.Enumerator
  - ▷ SortedList<TKey, TValue>
  - ▷ SortedSet<T>
  - ▷ SortedSet<T>.Enumerator
  - ▷ Stack<T>
  - ▷ Stack<T>.Enumerator

В таблице описаны основные обобщенные интерфейсы, с которыми придется иметь дело при работе с обобщенными классами коллекций:

Назначение	Интерфейс System.Collections.Generic
Определяет общие характеристики (размер, перечисление и безопасность к потокам и др.) для всех типов обобщенных коллекций	ICollection<T>
Определяет способ сравнения объектов	IComparer<T>
Позволяет объекту обобщенной коллекции представлять свое содержимое посредством пар «ключ/значение»	IDictionary<TKey, TValue>
Возвращает интерфейс <b>IEnumerator&lt;T&gt;</b> для заданного объекта	IEnumerable<T>
Позволяет выполнять итерацию в стиле <b>foreach</b> по элементам коллекции	IEnumerator<T>
Обеспечивает поведение добавления, удаления и индексации элементов в последовательном списке объектов	IList<T>
Предоставляет базовый интерфейс для абстракции множеств	ISet<T>

## Список обобщенных коллекций

Назначение	Поддерживаемые основные интерфейсы	Обобщенный класс
Представляет обобщенную коллекцию ключей и значений	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	Dictionary<TKey, TValue>
Представляет двусвязный список	ICollection<T>, IEnumerable<T>	LinkedList<T>
Последовательный список элементов с динамически изменяемым размером	ICollection<T>, IEnumerable<T>, IList<T>	List<T>
Обобщенная реализация очереди – списка, работающего по алгоритму «первый вошел – первый вышел» (FIFO)	ICollection, IEnumerable<T>	Queue<T>
Обобщенная реализация словаря – отсортированного множества пар «ключ/значение»	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	SortedDictionary<TKey, TValue>
Представляет коллекцию объектов, поддерживаемых в отсортированном порядке без дублирования	ICollection<T>, IEnumerable<T>, ISet<T>	SortedSet<T>
Обобщенная реализация стека – списка, работающего по алгоритму «последний вошел – первый вышел» (LIFO)	ICollection, IEnumerable<T>	Stack<T>

### Класс List<T>

В классе **List<T>** реализуется обобщенный динамический массив. Он ничем принципиально не отличается от класса необобщенной коллекции **ArrayList**.

В этом классе реализуются интерфейсы **ICollection**, **ICollection<T>**, **IList**, **IList<T>**, **IEnumerable** и **IEnumerable<T>**. У класса **List<T>** имеются следующие конструкторы:

```
public List()  
  
public List(IEnumerable<T> collection)  
  
public List(int capacity)
```

Первый конструктор создает пустую коллекцию класса **List** с выбираемой по умолчанию первоначальной емкостью. Второй – создает коллекцию типа **List** с количеством инициализируемых элементов, которое определяется параметром **collection** и равно первоначальной емкости массива. Третий конструктор создает коллекцию типа **List**, имеющую первоначальную емкость, задаваемую параметром **capacity**. В данном случае емкость обозначает размер базового массива, используемого

для хранения элементов коллекции. Емкость коллекции, создаваемой в виде динамического массива, может увеличиваться автоматически по мере добавления в нее элементов.

Среди методов коллекции **List<T>** можно выделить следующие:

- **void Add(T item)** – добавление нового элемента в список;
- **void AddRange(ICollection collection)** – добавление в список коллекции или массива;
- **int BinarySearch(T item)** – бинарный поиск элемента в списке. Если элемент найден, метод возвращает его индекс в коллекции. При этом список должен быть отсортирован;
- **int IndexOf(T item)** – возвращает индекс первого вхождения элемента в списке;
- **void Insert(int index, T item)** – вставляет элемент **item** в списке на позицию **index**;
- **bool Remove(T item)** – удаляет элемент **item** из списка. Если удаление прошло успешно, возвращает **true**;
- **void RemoveAt(int index)** – удаляет элемент по указанному индексу **index**;
- **void Sort()** – сортировка списка.

## Структура **KeyValuePair<TKey, TValue>**

В пространстве имен **System.Collections.Generic** определена структура **KeyValuePair<TKey, TValue>**. Она служит для хранения ключа и его значения, применяется в классах обобщенных коллекций, в которых хранятся пары «ключ-значение» (например, как 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>** применяется конструктор:

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

\* **key** обозначает ключ, а **value** – значение.

## Класс **Dictionary<TKey, TValue>**

Класс **Dictionary<TKey, TValue>** позволяет хранить пары «ключ-значение» в коллекции, как в словаре. Значения доступны в словаре по соответствующим ключам. В этом отношении **Dictionary<TKey, TValue>** аналогичен необобщенному классу **Hashtable**.

В классе **Dictionary<TKey, TValue>** реализуются интерфейсы **IDictionary**, **IDictionary<TKey, TValue>**, **ICollection**, **ICollection<KeyValuePair<TKey, TValue>>**, **IEnumerable**, **IEnumerable<KeyValuePair<TKey, TValue>>**, **ISerializable** и **IDeserializationCallback**. В двух последних интерфейсах поддерживается сериализация списка. Словари имеют динамический характер, расширяясь по мере необходимости.

В классе **Dictionary<TKey, TValue>** предоставляются многочисленные конструкторы. Наиболее часто используемые:

```
public Dictionary()
public Dictionary(IDictionary<TKey, TValue> dictionary)
public Dictionary(int capacity)
```

В первом конструкторе создается пустой словарь с выбираемой по умолчанию первоначальной емкостью. Во втором – создается словарь с указанным количеством элементов **dictionary**. А в третьем конструкторе с помощью параметра **capacity** указывается емкость коллекции, создаваемой в виде словаря. Если размер словаря заранее известен, то, указав емкость создаваемой коллекции, можно исключить изменение размера словаря во время выполнения (как правило, это требует дополнительных затрат вычислительных ресурсов).

```
using System;
using System.Collections.Generic;

namespace MyGame
{
    public class Class1
    {
        private void ExampleDictionary()
        {
            #region ExampleDictionary

            var dict = new Dictionary<char, string>();

            dict.Add('r', "Roman");
            dict.Add('i', "Iva");
            dict.Add('v', "Viktor");

            // Перебор коллекции
            foreach (KeyValuePair<char, string> user in dict)
            {
                Console.WriteLine($"{user.Key} - {user.Value}");
            }

            dict['i'] = "Roman"; // Изменяем элемент с ключом i
            dict['t'] = "Roman"; // Добавляем элемент с ключом t

            foreach (KeyValuePair<char, string> user in dict)
            {
                Console.WriteLine($"{user.Key} - {user.Value}");
            }

            dict.Remove('i'); // Удаляем элемент по ключу

            if (dict.ContainsKey('i')) // Проверяем, имеется ли элемент с ключом i
            {
                var tempUser = dict['i']; // Получаем элемент по ключу i
            }

            // Перебор ключей
            foreach (var user in dict.Keys)
            {
                Console.WriteLine($"{user}");
            }
        }
    }
}
```



```

        // Перебор по значениям
        foreach (var p in dict.Values)
        {
            Console.WriteLine(p);
        }

        #endregion

        #region C# 5

        Dictionary<int, string> dictionary = new Dictionary<int, string>
        {
            {1, "Roman" },
            {2, "Ivan" },
            {3, "Igor" },
            {4, "Vova" }
        };

        #endregion

        #region C# 6

        Dictionary<int, string> dictionary2 = new Dictionary<int, string>
        {
            [1] = "Roman",
            [2] = "Roman",
            [3] = "Roman",
            [4] = "Roman"
        };

        #endregion
    }
}

```

## Класс Queue<T>

Класс **Queue** представляет обычную очередь, работающую по алгоритму FIFO («первый вошел – первый вышел»). У класса **Queue** есть 3 основных метода:

- **Dequeue** – извлекает и возвращает первый элемент очереди;
- **Enqueue** – добавляет элемент в конец очереди;
- **Peek** – просто возвращает первый элемент из начала очереди без его удаления.

```
private void ExampleQueue()
{
    var arr = new Queue<int>(4);

    arr.Enqueue(1); // 1
    arr.Enqueue(1); // 1 1
    arr.Enqueue(5); // 1 1 5
    arr.Enqueue(2); // 1 1 5 2

    Console.WriteLine(arr.Peek()); // 1 1 5 2
    Console.WriteLine(arr.Dequeue()); // 1 5 2
}
```

## Класс Stack<T>

Класс **Stack** представляет коллекцию, которая использует алгоритм LIFO («последний вошел – первый вышел»). При такой организации каждый следующий добавленный элемент помещается поверх предыдущего. Извлечение из коллекции происходит в обратном порядке: извлекается тот элемент, который находится выше всех в стеке. В классе **Stack** можно выделить три основных метода, которые позволяют управлять элементами:

- **Push** – добавляет элемент в стек на первое место;
- **Pop** – извлекает и возвращает первый элемент из стека;
- **Peek** – просто возвращает первый элемент из стека без его удаления.

```
private void ExampleStack()
{
    var arr = new Stack<int>(4);

    arr.Push(1);
    arr.Push(1);
    arr.Push(5);
    arr.Push(2);

    Console.WriteLine(arr.Peek());
    Console.WriteLine(arr.Pop());
}
```

## Инициализация коллекции

Пример, как инициализировать коллекцию в момент объявления:

```
List<int> list = new List<int>() { 1, 2, 3, 4, 5 };
Dictionary<int, string> dict = new Dictionary<int, string>() { { 1, "One" }, { 2, "Two" } };
```

Начиная с C# 6.0 (Visual Studio 2015) доступен еще один способ инициализации словарей:

```
Dictionary<int, string> dict = new Dictionary<int, string>
{
    [1] = "One",
    [2] = "Two"
};
```

## Перебор элементов коллекции без привязки к ее реализации

Вместо того, чтобы писать циклы с обращением к элементам коллекции по индексу или ключу, воспользуйтесь конструкцией **foreach**. Она позволяет обратиться к объектам любого класса, реализующим интерфейс **IEnumerable** (или **IEnumerable<T>**):

```
using System;
using System.Collections.Generic;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = { 1, 2, 3, 4, 5 };
            foreach (int n in array)
            {
                Console.Write("{0} ", n);
            }
            Console.WriteLine();
            List<DateTime> times = new List<DateTime>(new[] { DateTime.Now,
DateTime.UtcNow });
            foreach (DateTime time in times)
            {
                Console.WriteLine(time);
            }
            Dictionary<int, string> numbers = new Dictionary<int, string>();
            numbers[1] = " One"; numbers[2] = " Two"; numbers[3] = " Three";
            foreach (KeyValuePair<int, string> pair in numbers)
            {
                Console.WriteLine($"{pair}");
            }
        }
    }
}
```

# Изменение порядка элементов массива на обратный

## Первый способ – самостоятельно

```
private static void Reverse<T>(T[] array)
{
    int left = 0, right = array.Length - 1;
    while (left < right)
    {
        T temp = array[left];
        array[left] = array[right];
        array[right] = temp;
        left++;
        right--;
    }
}
```

## Второй способ – с использованием метода расширения Reverse

```
int[] array = new int[5] { 1, 2, 3, 4, 5 };
IEnumerable<int> reversed = array.Reverse<int>();
```

Этот код будет корректно работать на любой коллекции с интерфейсом **IEnumerable<T>**. Но возвращает он не массив, а объект-итератор (привет делегатам!), который будет перебирать элементы оригинальной коллекции в обратном порядке.

Какой способ предпочесть – решаете вы, исходя из своих потребностей.

## Извлечение уникальных элементов из коллекции

**Задача.** Есть коллекция объектов, на основе которой вы хотите сгенерировать новую, содержащую по одной копии каждого объекта.

**Решение.** Чтобы сгенерировать коллекцию без повторяющихся элементов, вы должны отслеживать все элементы оригинальной коллекции и добавлять в новую лишь те, которые раньше не встречались. Рассмотрим пример:

```
private static ICollection<T> GetUniques<T>(ICollection<T> list)
{
    // Для отслеживания элементов используйте словарь
    Dictionary<T, bool> found = new Dictionary<T, bool> ();
    List<T> uniques = new List<T>();
    // Этот алгоритм сохраняет оригинальный порядок элементов
    foreach (T val in list)
    {
        if (!found.ContainsKey(val))
        {
            found[val] = true;
            uniques.Add(val);
        }
    }
    return uniques;
}
```

# Лямбда-выражения

C# поддерживает способность обрабатывать события «встроенным образом». С использованием анонимных методов назначается блок операторов кода непосредственно событию – вместо построения отдельного метода, подлежащего вызову делегатом. Лямбда-выражения – это всего лишь лаконичный способ записи анонимных методов, который в конечном итоге упрощает работу с типами делегатов .NET.

Чтобы подготовить фундамент для изучения лямбда-выражений, создадим новое консольное приложение **SimpleLambdaExpressions**. Затем займемся методом **FindAll ()** обобщенного типа **List<T>**. Этот метод может быть вызван, когда нужно извлечь подмножество элементов из коллекции, и он имеет следующий прототип:

```
// Метод класса System.Collections.Generic.List<T>.
public List<T> FindAll(Predicate<T> match)
```

Метод возвращает объект **List<T>**, представляющий подмножество данных. Единственный параметр **FindAll ()** – обобщенный делегат типа **System.Predicate<T>**. Он может указывать на любой метод, возвращающий **bool** и принимающий единственный параметр:

```
// Этот делегат используется методом FindAll() для извлечения подмножества
public delegate bool Predicate<T>(T obj);
```

Когда вызывается **FindAll ()**, каждый элемент в **List<T>** передается методу, указанному объектом **Predicate<T>**. Реализация этого метода будет производить вычисления, чтобы проверить соответствие элемента данных указанному критерию и вернуть в результате **true** или **false**. Если метод вернет **true**, текущий элемент будет добавлен в **List<T>**, представляющий искомое подмножество. Прежде чем посмотреть, как лямбда-выражения упрощают работу с **FindAll ()**, решим эту задачу в длинной нотации, используя объекты делегатов непосредственно.

Добавим в класс **Program** метод **TraditionalDelegateSyntax()**, который взаимодействует с **System.Predicate<T>** для обнаружения четных чисел в списке целочисленных значений **List<T>**:

```
using System;
using System.Collections.Generic;
// Эндрю Троелсен. Язык программирования C#5.0
// Понятие лямбда-выражения
namespace Лямбда_выражения
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Fun with Lambdas *****\n");
            TraditionalDelegateSyntax();
            Console.ReadLine();
        }

        static void TraditionalDelegateSyntax()
        {
            // Создать список целых чисел
            List<int> list = new List<int>();
            list.AddRange(new int[] { 20, 1, 4, 8, 9, 4, 4 });
            // Вызов FindAll() с использованием традиционного синтаксиса делегатов
            // Создаем обобщенный экземпляр обобщенного делегата, используя встроенный
            // делегат Predicate
            Predicate<int> predicate = new Predicate<int>(IsEvenNumber);
            // Создаем список целых чисел, используя метод FindAll, в который передаем
            // делегат
            List<int> evenNumbers = list.FindAll(predicate);
            Console.WriteLine("Здесь только четные числа:");
            foreach (int evenNumber in evenNumbers)
            {
                Console.Write("{0}\t", evenNumber);
            }
            Console.WriteLine();
        }

        // Цель для делегата Predicate<>.
        static bool IsEvenNumber(int i)
        {
            // Это четное число?
            return (i % 2) == 0;
        }
    }
}
```

Хотя этот традиционный подход к работе с делегатами функционирует ожидаемым образом, метод **IsEvenNumber ()** вызывается только при очень ограниченных условиях. В частности, когда вызывается **FindAll ()**, который взваливает на нас все заботы по определению метода. Если бы вместо этого использовался анонимный метод, код стал бы существенно яснее.

Рассмотрим новый метод в классе **Program**:

```
using System;
using System.Collections.Generic;
// Эндрю Троелсен. Язык программирования C#5.0
// Понятие лямбда-выражения
namespace Лямбда_выражения_00200
{
    class Program
    {
        static void Main(string[] args)
        {
        }
        static void AnonymousMethodSyntax()
        {
            // Создать список целых чисел
            List<int> list = new List<int>();
            list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
            // Теперь использовать анонимный метод
            List<int> evenNumbers = list.FindAll(delegate (int i)
                                                { return (i % 2) == 0; });
            // Вывод четных чисел
            Console.WriteLine("Here are your even numbers:");
            foreach (int evenNumber in evenNumbers)
            {
                Console.Write("{0}\t", evenNumber);
            }
            Console.WriteLine();
        }
    }
}
```

Для дальнейшего упрощения вызова **FindAll ()** можно применять лямбда-выражения. Используя этот новый синтаксис, вообще не приходится иметь дело с лежащим в основе объектом делегата. Рассмотрим новый метод в классе **Program**:

```
static void LambdaExpressionSyntax()
{
    // Создать список целых чисел
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
    // Теперь использовать лямбда-выражение C#
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
    // Вывод четных чисел
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

Обратите внимание на странный оператор кода, передаваемый методу **FindAll ()**, который в действительности и является лямбда-выражением. В этой модификации примера вообще нет никаких

следов делегата **Predicate<T>** (как и ключевого слова **delegate**). Все, что указано вместо них – это лямбда-выражение: `i => (i % 2) == 0`

Лямбда-выражения могут применяться везде, где используется анонимный метод или строго типизированный делегат (обычно в более лаконичном виде). «За кулисами» компилятор C# транслирует лямбда-выражение в стандартный анонимный метод, использующий тип делегата **Predicate<T>**.

## Использование преимуществ контравариантности

**Задача.** В предыдущих версиях .NET (до 4) возникали ситуации, в которых делегаты вели себя неожиданным образом. Например, делегат с параметром-типом базового класса должен легко присваиваться делегатам с производными параметрами-типами – поскольку любой делегат, вызываемый из базового класса, должен позволять вызывать себя из производного. Ситуацию иллюстрирует код, приведенный ниже:

```
using System;
namespace ConsoleApplication1
{
    class Program
    {
        class Shape
        {
            public void Draw() { Console.WriteLine("Drawing shape"); }
        };
        class Rectangle : Shape
        {
            public void Expand() { /*...*/ }
        };
        // А делегат и метод определены так:
        delegate void ShapeAction<T>(T shape);
        static void DrawShape(Shape shape)
        {
            if (shape != null)
            {
                shape.Draw();
            }
        }
        static void Main(string[] args)
        {
        }
    }
}
```



**Решение.** В версии .NET 4 контравариантность делегатов решила проблему, и вы можете присваивать менее специфичные делегаты более специфичным. Теперь параметр-тип **T** определен с модификатором **in**. Это означает, что делегат не возвращает **T**. В коде, приведенном далее, параметр-тип делегата модифицирован ключевым словом **in**:

```
using System;
namespace ConsoleApplication1
{
    class Shape
    {
        public void Draw() { Console.WriteLine("Drawing shape"); }
    };
    class Rectangle : Shape
    {
        public void Expand() { /*...*/ }
    };
    class Program
    {
        delegate void ShapeAction<in T>(T shape);
        static void DrawShape(Shape shape)
        {
            if (shape != null)
            {
                shape.Draw();
            }
        }
        static void Main(string[] args)
        {
            // Очевидно, что этот код должен работать
            ShapeAction<Shape> action = DrawShape;
            action(new Rectangle());
            /* Интуитивно понятно, что любой метод, удовлетворяющий делегату
            ShapeAction<Shape>, должен работать с объектом Rectangle, потому что Rectangle
            является производным от Shape. Всегда есть возможность присвоить менее
            специфичный метод более специфичному делегату. Но до появления версии .NET 4
            нельзя было присвоить менее специфичный делегат более специфичному делегату. Это
            очень важное различие. Теперь это можно, поскольку параметр-тип помечен
            модификатором "in" */
            // Следующие действия были возможны до появления .NET 4
            ShapeAction<Rectangle> rectAction1 = DrawShape;
            rectAction1(new Rectangle());
            // Take Advantage of Contravariance 293
            // А это было невозможно до появления .NET 4
            ShapeAction<Rectangle> rectAction2 = action;
            rectAction2(new Rectangle());
            Console.ReadKey();
        }
    }
}
```

## Пример метода расширений

```
using System;
using ExtensionMethods;
// Пример метода расширений (слово this перед параметром)
// https://msdn.microsoft.com/ru-ru/library/bb383977.aspx
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        // Создали метод, который добавляет (расширяет список методов) метод WordCount в
        // список методов типа String
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace MyProgram
{
    class Program
    {
        static void Main()
        {
            string s = "Hello Extension Methods";
            // Теперь в классе s появился новый метод
            int i = s.WordCount();
        }
    }
}
```

## Еще примеры метода расширений

```
using System;
using System.Collections.Generic;
using System.Linq;

public static class ExampleExtensions
{
    /// <summary>
    /// Добавление элементов в коллекцию
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="self"></param>
    /// <param name="coll"></param>
    /// <returns></returns>
    public static T AddTo<T>(this T self, ICollection<T> coll)
    {
        coll.Add(self);
        return self;
    }

    /// <summary>
    /// Переводит строку в логический тип данных
    /// </summary>
    /// <param name="self"></param>
}
```

```

    /// <returns> Если не может конвертировать, то вернет ложь </returns>
    public static bool TryBool(this string self)
    {
        return Boolean.TryParse(self, out var res) && res;
    }

    /// <summary>
    /// Проверяет, существует ли элемент среди множества
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="self"></param>
    /// <param name="elem"></param>
    /// <returns></returns>
    public static bool IsOneOf<T>(this T self, params T[] elem)
    {
        return elem.Contains(self);
    }

    /// <summary>
    /// Возвращает отформатированную строку
    /// </summary>
    /// <param name="format"></param>
    /// <param name="args"></param>
    /// <returns></returns>
    public static string Fun(this string format, params object[] args)
    {
        return String.Format(format, args);
    }
}

public class Test
{
    private void AddToList()
    {
        var list = new List<int>();
        var list2 = new List<int>();

        list.Add(1);

        1.AddTo(list).AddTo(list2);
    }

    private void ExampleTryBool(string value)
    {
        bool res = value.TryBool();
    }

    private void ExampleOne()
    {
        var arr = new[] { 1, 5, 6, 3 };

        var t = 7;

        foreach (var i in arr)
        {
            if (i == 7)
            {
            }
        }
    }
}

```

```
    }  
  
    if (t.IsOneOf(5, 8, 6, 3))  
    {  
  
    }  
}  
}
```

## Linq

Linq использует следующие программные конструкции:

- Неявная типизация локальных переменных;
- Синтаксис инициализации объектов и коллекций;
- Лямбда-выражения;
- Расширяющие методы;
- Анонимные типы.

В основу Linq положено понятие запроса, в котором определяется информация, получаемая из источника данных. Например, запрос списка рассылки почтовых сообщений заказчикам может потребовать предоставления адресов всех заказчиков, проживающих в конкретном городе; запрос базы данных товарных запасов – список товаров, запасы которых исчерпались на складе; а запрос журнала, регистрирующего интенсивность использования Интернета, – список наиболее часто посещаемых веб-сайтов. И хотя все эти запросы отличаются в деталях, их можно выразить, используя одни и те же синтаксические элементы Linq. Как только запрос будет сформирован, его можно выполнить. Это делается, в частности, в цикле **foreach**. В результате выполнения запроса выводятся его результаты. Поэтому использование запроса может быть разделено на две главные стадии. На первой запрос формируется, а на второй – выполняется. Таким образом, при формировании запроса определяется, что именно следует извлечь из источника данных. А при выполнении запроса выводятся конкретные результаты.

Для обращения к источнику данных по запросу, сформированному средствами Linq, в этом источнике должен быть реализован интерфейс **IEnumerable**. Он имеет две формы: обобщенную и необобщенную. Как правило, работать с источником данных легче, если в нем реализуется обобщенная форма **IEnumerable<T>**, где **T** обозначает обобщенный тип перечисляемых данных. Здесь и далее предполагается, что в источнике данных реализуется форма интерфейса **IEnumerable<T>**. Этот интерфейс объявляется в пространстве имен **System.Collections.Generic**. Класс, в котором реализуется форма интерфейса **IEnumerable<T>**, поддерживает перечисление – это означает, что его содержимое может быть получено по очереди или в определенном порядке. Форма интерфейса **IEnumerable<T>** поддерживается всеми массивами в C#. Поэтому на примере массивов можно наглядно продемонстрировать основные принципы работы Linq. Применение Linq не ограничивается массивами. Есть два способа выполнения запроса Linq: отложенное и немедленное. При отложенном выполнении Linq-выражение не выполняется, пока не будет произведена итерация или перебор по выборке.

## Простой Linq-запрос

```
// Сформировать простой запрос Linq
using System;
// Обязательно подключить
using System.Linq;
class SimpQuery
{
    static void Main()
    {
        // Все массивы в C# неявным образом преобразуются в форму интерфейса
        IEnumerable<T>
        // Благодаря этому любой массив в C# может служить в качестве источника данных,
        // извлекаемых
        // по запросу Linq
        int[] nums = { 1, -2, 3, 0, -4, 5 };
        // Создадим запрос, получающий только положительные числа
        // posNums - переменная запроса
        // Ей присваивается результат выполнения запроса
        var posNums = from n           // n -переменная диапазона (как в foreach)
                      in nums         // Источник данных
                      where n > 0     // Предикат (условие) - фильтр данных
                      select n;       // Какие данные получаем? В сложных запросах
// здесь можно указать, например, фамилию адресата вместо всего адреса
        Console.WriteLine("Положительные числа: ");
        // Выполняем запрос и выводим положительные числа на экран
        foreach (int i in posNums) Console.Write(i + " ");
        Console.WriteLine();
    }
}
```

## Два where (условия)

```
using System;
using System.Linq;
class TwoWheres
{
    static void Main()
    {
        int[] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
        // Создаем запрос на выборку положительных чисел меньше 10
        var posNums = from n in nums
                      where n > 0
                      where n < 10
                      select n;

        Console.WriteLine("Положительные числа меньше 10: ");
        foreach (int i in posNums) Console.Write(i + " ");
        Console.WriteLine();
    }
}
```

## Еще один пример с where

```
// Еще один where
using System;
using System.Linq;
class WhereDemo2 {
    static void Main() {
        string[] strs = { ".com", ".net", "hsNameA.com", "hsNameB.net",
                           "test", ".network", "hsNameC.net", "hsNameD.com" };
        // Create a query that obtains Internet addresses that end with .net
        // Создадим запрос, который получает все Интернет-адреса, заканчивающиеся на
        // .net
        var netAddrs = from addr in strs
                       where addr.Length > 4
                          && addr.EndsWith(".net", StringComparison.Ordinal)
        // Он возвращает логическое значение true, если вызывающая его строка
        // оканчивается
        // последовательностью символов, указываемой в качестве аргумента этого метода
        // Сортировка результатов запроса с помощью оператора orderby
        select addr;
        // Выполним запрос и выведем результаты
        foreach (var str in netAddrs) Console.WriteLine(str);
    }
}
```

## Демонстрация OrderBy и преобразования в список

```
// Демонстрация OrderBy
using System;
using System.Collections.Generic;
using System.Linq;
class OrderbyDemo
{
    static void Main()
    {
        int[] nums = { 10, -19, 4, 7, 2, -5, 0 };
        // Запрос, который получает значения в отсортированном порядке
        var posNums = from n in nums
                     where n > 0 orderby n ascending
                     select n;
        Console.Write("Значения по возрастанию: ");
        // Преобразовать в список (для примера такой возможности)
        List<int> a = posNums.ToList<int>();
        // Execute the query and display the results
        foreach (int i in posNums) Console.Write(i + " ");
        nums[1] = 10;
        Console.WriteLine("\n" + posNums.Sum());
        Console.WriteLine();
    }
}
```

## Демонстрация использования Linq с массивом пользовательских данных

```
using System;
using System.Linq;
class EmailAddress
{
    public string Name { get; set; }
    public string Address { get; set; }
    public EmailAddress(string n, string a)
    {
        Name = n;
        Address = a;
    }
}
class SelectDemo2
{
    static void Main()
    {
        EmailAddress[] addr = {
            new EmailAddress("Herb", "Herb@HerbSchildt.com"),
            new EmailAddress("Tom", "Tom@HerbSchildt.com"),
            new EmailAddress("Sara", "Sara@HerbSchildt.com")
        };
        // Create a query that selects e-mail addresses
        var eAddr = from entry in addr
                    select entry.Address;
        Console.WriteLine("The e-mail addresses are");
        // Execute the query and display the results
        foreach (string s in eAddr) Console.WriteLine("  " + s);
    }
}
```

## Основные Linq

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Lesson
{
    public struct User
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public int Age
        {
            get => _age;
            set => _age = value;
        }

        private int _age;

        public User(string firstName, string lastName) : this()
        {
            FirstName = firstName;
            LastName = lastName;
        }
    }
}
```

```

    }
}

internal class ExampleLinq
{
    private readonly User[] _users;
    private readonly int[] _numbers;

    public ExampleLinq()
    {
        _users = new[]
        {
            new User("Roman", "Muratov") {Age = 18},
            new User("Ivan", "Petrov") {Age = 22},
            new User("Igor", "Ivanov") {Age = 25},
            new User("Lera", "Muratova") {Age = 17},
            new User("Sveta", "Petrova") {Age = 27},
            new User("Lena", "Ivanova") {Age = 33},
            new User("Lera", "Muratova") {Age = 17},
            new User("Sveta", "Petrova") {Age = 27},
            new User("Lena", "Ivanova") {Age = 33}
        };
        _numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
    }

    public void Filtration()
    {
        IEnumerable<int> evens = from i in _numbers
                                where i % 2 == 0 && i > 10
                                select i;

        IEnumerable<int> evens1 = _numbers.Where(i => i % 2 == 0 && i > 10);

        foreach (int i in evens)
            Console.WriteLine(i);
    }

    public void SelectingComplexObjects()
    {
        var selectedUsers = from user in _users
                            where user.Age > 25
                            select user;

        var selectedUsers1 = _users.Where(u => u.Age > 25).ToList();

        foreach (User user in selectedUsers)
            Console.WriteLine("{0} - {1}", user.FirstName, user.Age);
    }

    public void Projection()
    {
        var names = _users.Select(u => u.FirstName).ToList();

        foreach (string user in names)
            Console.WriteLine(user);
    }

    public void ExampleLet()

```



```

    {
        var people = from u in _users
                      let age = u.Age <= 18 ? u.Age + (18 - u.Age) : u.Age
                      select new User(u.FirstName, u.LastName)
                      {
                          Age = age
                      };

        foreach (var user in people)
            Console.WriteLine("{0} - {1}", user.FirstName, user.Age);
    }

    public void SamplingFromSeveralSources()
    {
        var people = from user in _users
                      from number in _numbers
                      select new { Name = user.LastName, Number = number };

        foreach (var p in people)
            Console.WriteLine("{0} - {1}", p.Name, p.Number);
    }

    public void Sorting()
    {
        var sortedUsers = from u in _users
                           // ascending (сортировка по возрастанию) и descending (сортировка по убыванию)
                           orderby u.Age
                           select u;

        // ThenByDescending() (для сортировки по убыванию)
        var result = _users.OrderBy(u => u.FirstName).ThenBy(u =>
u.Age).ThenBy(u => u.FirstName.Length);

        foreach (User u in sortedUsers)
            Console.WriteLine("{0} - {1}", u.FirstName, u.Age);
    }

    public void WorkingWithSets()
    {
        string[] peopleFromAstrakhan = { "Igor", "Roman", "Ivan" };
        string[] peopleFromMoscow = { "Roman", "Vitalik", "Denis" };

        // Разность множеств
        var result = peopleFromAstrakhan.Except(peopleFromMoscow);
        // Пересечение множеств
        var result1 = peopleFromAstrakhan.Intersect(peopleFromMoscow);
        // Объединение множеств
        var result2 = peopleFromAstrakhan.Union(peopleFromMoscow);
        // Удаление дубликатов
        var result3 =
peopleFromAstrakhan.Concat(peopleFromMoscow).Distinct();
    }

    public void ExampleAverage()
    {
        int min1 = _numbers.Min();
        // Минимальный возраст
        int min2 = _users.Min(n => n.Age);
    }

```

```

        int max1 = _numbers.Max();
// Максимальный возраст
        int max2 = _users.Max(n => n.Age);

        double avr1 = _numbers.Average();
// Средний возраст
        double avr2 = _users.Average(n => n.Age);
    }

    public void ExampleSkipAndTake()
    {
// Три первых элемента
        var result = _numbers.Take(3);
// Все элементы, кроме первых трех
        var result1 = _numbers.Skip(3);

        foreach (var t in _users.TakeWhile(x =>
x.FirstName.StartsWith("I")))
            Console.WriteLine(t);

        foreach (var t in _users.SkipWhile(x =>
x.FirstName.StartsWith("I")))
            Console.WriteLine(t);
    }

    public void ExampleGrouping()
    {
        var groups = from user in _users
                      group user by user.LastName;

// foreach (IGrouping<string, User> g in groups)
// {
//     Console.WriteLine(g.Key);
//     foreach (var t in g)
//         Console.WriteLine(t.FirstName);
//     Console.WriteLine();
// }

        var groups1 = _users.GroupBy(p => p.LastName)
                           .Select(g => new { LastName = g.Key, Count = g.Count()
});

        var groups2 = _users.GroupBy(p => p.LastName)
                           .Select(g => new
        {
            LastName = g.Key,
            Count = g.Count(),
            Name = g.Select(p => p)
        });

        foreach (var g in groups2)
        {
            Console.WriteLine(g.LastName);
            Console.WriteLine(g.Count);
            foreach (var t in g.Name)
                Console.WriteLine(t.FirstName);
            Console.WriteLine();
        }
    }
}

```

```

    }

    public void ExampleAllAndAny()
    {
        bool result = _users.All(u => u.Age > 20);
        Console.WriteLine(result
            ? "У всех пользователей возраст больше 20"
            : "Есть пользователи с возрастом меньше 20");

        bool result1 = _users.Any(u => u.Age < 20);
        Console.WriteLine(result1
            ? "Есть пользователи с возрастом меньше 20"
            : "У всех пользователей возраст больше 20");
    }
}

```

## Практика

### «Астероиды» с коллекциями

Изменим программу так, чтобы можно было стрелять очередями.

Добавим в файл **Game.cs** пространство имен:

```
using System.Collections.Generic;
```

Вместо одной пули создадим коллекцию пуль:

```
private static List<Bullet> _bullets=new List<Bullet>();
```

При нажатии на выстрел пуля будет добавляться в коллекцию:

```
if (e.KeyCode == Keys.ControlKey) _bullets.Add(new Bullet(new Point(_ship.Rect.X
+ 10, _ship.Rect.Y + 4), new Point(4, 0), new Size(4, 1)));
```

В методе **Draw** класса **Game** теперь нужно выводить все пули:

```
foreach (Bullet b in _bullets) b.Draw();
```

В методе **Update**:

```
foreach (Bullet b in _bullets) b.Update();
```

В методе **Load** удалить строку:

```
_bullet = new Bullet(new Point(0, 200), new Point(5, 0), new Size(4, 1));
```

Существенно изменится столкновение, поэтому приведем код **Update** класса **Game** полностью:

```
public static void Update()
{
    foreach (BaseObject obj in _objs) obj.Update();
    foreach (Bullet b in _bullets) b.Update();
    for (var i = 0; i < _asteroids.Length; i++)
    {
        if (_asteroids[i] == null) continue;
        _asteroids[i].Update();
        for (int j = 0; j < _bullets.Count; j++)
            if (_asteroids[i] != null && _bullets[j].Collision(_asteroids[i]))
            {
                System.Media.SystemSounds.Hand.Play();
                _asteroids[i] = null;
                _bullets.RemoveAt(j);
                j--;
            }
        if (_asteroids[i] == null || !_ship.Collision(_asteroids[i])) continue;
        _ship.EnergyLow(Rnd.Next(1, 10));
        System.Media.SystemSounds.Asterisk.Play();
        if (_ship.Energy <= 0) _ship.Die();
    }
}
```

## Новшества C# 7

В C# 7 добавлены out-переменные, которые позволяют объявить переменные сразу в вызове метода:

```
private void GetAge()
{
    Console.WriteLine(@"Введите возраст");
    var age = Console.ReadLine();
    if (!Int32.TryParse(age, out var ageInt)) Console.WriteLine(@"Возраст введен
не корректно");
}
```

## Локальные функции

Локальные функции – те, что определены внутри других методов:

```
private void GetAge()
{
    Console.WriteLine(@"Введите возраст");
    var ageInput = Console.ReadLine();

    bool IsAdult(string value) // Локальная функция
    {
        if (!Int32.TryParse(ageInput, out var age)) throw new
Exception(@"Возраст введен некорректно");
        if (age > 18)
        {
            return true;
        }
        return false;
    }

    if (!IsAdult(ageInput))
    {
        Console.WriteLine(@"Регистрация разрешена только совершеннолетним");
    }
}
```

## Локальная переменная-ссылка

В версии C# 7.0 была добавлена возможность возвращать ссылку на объект с помощью ключевого слова **ref** и определять локальную переменную, которая будет хранить ссылку:

```
private void Example()
{
    byte x = 1;
    ref byte xRef = ref x;
    Console.WriteLine(x); // 1
    xRef = 255;
    Console.WriteLine(x); // 255
}
```

## Ссылка как результат функции

В версии C# 7.0 была добавлена возможность возвращать ссылку на результат выполнения метода с помощью ключевого слова **ref**:

```
private void Main()
{
    int[] numbers = { 1, 1, 2, 3, 5, 8, 13 };
    ref int numberRef = ref Find(5, numbers); // Ищем число 5 в массиве
    numberRef = 11; // Заменяем 5 на 11
    Console.WriteLine(numbers[4]); // 11

    Console.Read();
}

private ref int Find(int elem, int[] collection)
{
    for (var i = 0; i < collection.Length; i++)
    {
        if (collection[i] == elem)
        {
            return ref collection[i]; // Возвращаем ссылку на адрес, а не само
значение
        }
    }
    throw new IndexOutOfRangeException("число не найдено");
}
```

## Pattern matching

Pattern matching позволяет сократить объем кода. Вместо приведения объекта к конкретному типу (операторы `is` и `as`) на новой строке, теперь это можно сделать непосредственно в условии. Pattern matching доступны с версии C# 7.0.

```
class Animals
{
    public virtual string GetName()
    {
        return $"I Class {nameof(Animals)}";
    }
}

class Cat : Animals
{
    public override string GetName()
    {
        return $"I Class {nameof(Cat)}";
    }

    public int Age { get; set; }
}

internal class Programm
{
    private static void Main()
    {

```

```

        Animals animals = new Animals
        {
            Age = 3 // ?? В классе Animals нет свойства Age
        };
        Example(animals);
    }

private static void Example(Animals animals)
{
    #region C# 6.0

    Cat cat = animals as Cat;
    if (cat != null && cat.Age > 5)
    {
        cat.GetName();
    }
    else
    {
        Console.WriteLine(@"Пришло не то животное");
    }

    #endregion

    #region C# 7.0

    if (animals is Cat cat1 && cat1.Age > 5)
    {
        cat1.GetName();
    }
    else
    {
        Console.WriteLine(@"Пришло не то животное");
    }

    #endregion

}

```

Кроме выражения **if**, **pattern matching** может применяться в конструкции **switch**:

```

private void Example(Animals animals)
{
    switch (animals)
    {
        case Cat cat:
            cat.GetName();
            break;
        case null:
            Console.WriteLine(@"Объект не задан");
            break;
        default:
            Console.WriteLine(@"Пришло не животное");
            break;
    }
}

```

**Примечание.** Для следующего функционала (деконструкторы и кортежи) понадобится загрузить **Nuget**-пакет (**System.ValueTuple**).

## Деконструкторы

Деконструкторы (не путать с деструкторами) позволяют выполнить декомпозицию объекта на отдельные части. Деконструкторы доступны, начиная с версии C# 7.0.

```
public struct User
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public int Age
    {
        get => _age;
        set => _age = value;
    }

    private int _age;

    public User(string firstName, string lastName) : this()
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public void Deconstruct(out string firstName, out string lastName, out
int age)
    {
        firstName = FirstName;
        lastName = LastName;
        age = Age;
    }
}

internal class Programm
{
    private static void Main()
    {
        User person = new User("Roman", "Muratov") { Age = 25 };

        (string firstName, string lastName, int age) = person;

        Console.WriteLine(firstName);    // Roman
        Console.WriteLine(lastName);    // Muratov
        Console.WriteLine(age);          // 25
    }
}
```

## Кортежи

Кортежи в C# – это типы, которые определяются с помощью упрощенного синтаксиса. Преимущества: более простой синтаксис, правила преобразований с учетом количества (кратности) и типов



элементов, а также единые правила для копий и назначений. При этом кортежи не поддерживают некоторые объектно-ориентированные идиомы, связанные с наследованием.

## Практическое задание

1. Добавить в программу коллекцию астероидов. Как только она заканчивается (все астероиды сбиты), формируется новая коллекция, в которой на один астероид больше.
2. Дана коллекция **List<T>**. Требуется подсчитать, сколько раз каждый элемент встречается в данной коллекции:
  - a. для целых чисел;
  - b. \* для обобщенной коллекции;
  - c. \*\* используя Linq.
3. \* Дан фрагмент программы:

```
Dictionary<string, int> dict = new Dictionary<string, int>()
{
    { "four", 4 },
    { "two", 2 },
    { "one", 1 },
    { "three", 3 },
};
var d = dict.OrderBy(delegate(KeyValuePair<string, int> pair) {
return pair.Value; });
foreach (var pair in d)
{
    Console.WriteLine("{0} - {1}", pair.Key, pair.Value);
}
```

- a. Свернуть обращение к **OrderBy** с использованием лямбда-выражения **=>**.
- b. \* Развернуть обращение к **OrderBy** с использованием делегата .

## Дополнительные материалы

1. [Ковариантность и контравариантность \(Википедия\)](#)
2. <https://andrewlock.net/deconstructors-for-non-tuple-types-in-c-7-0/>
3. <https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/nameof>
4. <https://docs.microsoft.com/ru-ru/dotnet/csharp/tuples>

## Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Татьяна Павловская. Программирование на языке высокого уровня – 2009 г.
2. Эндрю Троелсен. Язык программирования C# 5.0 и платформа .NET 4.5 –2013 г.
3. Герберт Шилдт. C# 4.0. Полное руководство – 2011 г.
4. Бен Ватсон. C# 4.0 на примерах – 2011 г.
5. [MSDN](#)