Приднестровский государственный университет им. Т.Г. Шевченко

физико-математический факультет

кафедра прикладной математики и информатики

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 7

по дисциплине:
«*Системы программирования*»

Тема:
«Массивы и коллекции»

РАЗРАБОТАЛИ:
ст. преподаватель кафедры ПМиИ
Великодный В.И.
ст. преподаватель кафедры ПМиИ
Калинкова Е.В.

г. Тирасполь

Цель работы:

Изучить принципы работы с массивами и коллекциями в C#. Закрепить полученные знания при решении задач с использованием массивов и обобщенных коллекций.

Теоретическая часть

Массивы

Общие принципы организации массивов

Массивом называют набор элементов одного типа, к которым можно обращаться через одно общее имя переменной. Каждый элемент массива имеет целочисленный номер – ***индекс***, определяющий позицию элемента в массиве. В языке C# каждый индекс изменяется в диапазоне от 0 до некоторого конечного значения. Число индексов характеризует ***размерность*** массива. Массивы могут быть ***одномерными*** и ***многомерными***. ***Размер*** массива – это количество всех его элементов.

Массивами в C# можно пользоваться практически так же, как и в других языках программирования. Однако в C# массивы имеют существенные отличия: они относятся ***к ссылочным типам данных***, более того – реализованы ***как объекты***. Фактически, имя массива является ссылкой на динамическую область памяти, называемую кучей, в которой последовательно размещается набор элементов определенного типа. Элементами массива могут быть величины как значимых, так и ссылочных типов (в том числе другие массивы). *Массив значимых типов* хранит значения, *массив ссылочных типов* – ссылки на элементы.

Объявление и инициализация одномерных массивов

Одномерный массив представляет собой список связанных переменных.

Для того чтобы воспользоваться массивом в программе, требуется двухэтапная процедура, поскольку в C# массивы реализованы в виде объектов. Во-первых, необходимо объявить переменную, которая может обращаться к массиву. И во-вторых, нужно создать экземпляр массива, используя оператор `new`. Так, для объявления одномерного массива обычно применяется следующая общая форма:

```
тип[] имя_массива = new тип[размер];
```

где *тип* объявляет конкретный тип элемента массива. Тип элемента определяет тип данных каждого элемента, составляющего массив. Квадратные скобки, которые сопровождают тип, указывают на то, что объявляется одномерный массив. А *размер* определяет число элементов массива.

Пример. В приведенной ниже строке кода создается массив из пяти элементов типа `int`.

```
int[] myArr = new int[5];
```

В переменной `myArr` хранится ссылка на область памяти, выделяемую для массива оператором `new`.

Приведенное выше объявление массива можно разделить на два отдельных оператора. Например:

```
int[] myArr;
myArr = new int[5];
```

В данном случае переменная `myArr` не ссылается на какой-то определенный физический объект, когда она создается в первом операторе. И лишь после выполнения второго оператора эта переменная ссылается на массив.

Доступ к отдельному элементу массива осуществляется по индексу. Индекс обозначает положение элемента в массиве. В языке C# индекс первого элемента всех массивов оказывается нулевым. В частности, массив `myArr` состоит из 5 элементов с индексами от 0 до 4. Для индексирования массива достаточно указать номер требуемого элемента в квадратных скобках. Так, первый элемент массива `myArr` обозначается как `myArr[0]`, а последний его элемент – как `myArr[4]`.

Пример. Программа, в которой заполняются все элементы массива `myArr`.

```
using System;
class Program
{
    static void Main()
    {
        // Объявляем массив
        int[] myArr = new int[5];
        // Инициализируем каждый элемент массива вручную
        myArr[0] = 17;
        myArr[1] = 25;
        myArr[2] = 14;
        myArr[3] = 3;
        myArr[4] = 8;
        // Выводим элементы массива
        for (int i = 0; i < 5; i++)
            Console.WriteLine(myArr[i]);
        Console.ReadLine();
   }
}
```

Следует иметь в виду, что если массив только объявляется, но явно не инициализируется, каждый его элемент будет установлен в значение, принятое по умолчанию для соответствующего типа данных (например, элементы массива типа `bool` будут устанавливаться в `false`, а элементы массива типа `int` – в 0).

Помимо заполнения массива элемент за элементом (как показано в предыдущем примере), можно также заполнять его с использованием специального синтаксиса инициализации массивов. Для этого необходимо перечислить включаемые в массив элементы в фигурных скобках { }. Такой синтаксис удобен при создании массива известного размера, когда нужно быстро задать его начальные значения:

```
// Инициализация массива с использованием ключевого слова new
int[] myArr = new int[] { 10, 20, 30, 40, 50 };

// Инициализация массива без использования ключевого слова new
string[] info = { "Фамилия", "Имя", "Отчество" };

// При инициализации массива его размер можно указывать явным образом
char[] symbol = new char[4] { 'X', 'Y', 'Z', 'M' };
```

Несмотря на свою избыточность, форма инициализации массива с оператором `new` оказывается полезной в том случае, если новый массив присваивается уже существующей ссылке на массив. Например:

```
int[] myArr;
myArr = new int[] { 10, 20, 30, 40, 50 };
```

В данном случае переменная `myArr` объявляется в первом операторе и инициализируется во втором.

Ошибкой будет инициализация массива следующим образом:

```
int[] myArr;
myArr = { 10, 20, 30, 40, 50 }; // Ошибка
```

Объявление и инициализация двумерных массивов

Двумерный массив является простейшей формой многомерного массива. Местоположение любого элемента в двумерном массиве обозначается двумя индексами. Такой массив можно представить в виде таблицы, на строки которой указывает один индекс, а на столбцы – другой. Синтаксис объявления двумерного массива:

```
тип[,] имя_массива = new тип[размер1, размер2];
```

Например, следующее объявление создает массив из четырех строк и пяти столбцов.

```
int[,] matr = new int[4, 5];
```

Для доступа к элементу двумерного массива следует указать оба индекса, разделив их запятой, например, `matr[2, 3]` (рис. 1).

matr[0,0]	matr[0,1]	matr[0,2]	matr[0,3]	matr[0,4]
matr[1,0]	matr[1,1]	matr[1,2]	matr[1,3]	matr[1,4]
matr[2,0]	matr[2,1]	matr[2,2]	matr[2,3]	matr[2,4]
matr[3,0]	matr[3,1]	matr[3,2]	matr[3,3]	matr[3,4]

Рис. 1. *Двумерный массив*

Массив можно инициализировать при объявлении. Например:

```
int[,] arr = new int[,] {{1, 2}, {3, 4}, {5, 6}};
```

или

```
int[,] arr = new int[3,2] {{1, 2}, {3, 4}, {5, 6}};
```

или

```
int[,] arr = {{1, 2}, {3, 4}, {5, 6}};
```

1	2
3	4
5	6

Рис. 2. *Массив `arr` после инициализации*

Массивы трех и более измерений

В C# допускаются массивы трех и более измерений. Общая форма объявления многомерного массива.

```
тип[,...,] имя_массива = new тип[размер1, размер2, ... размерN];
```

Например, в приведенном ниже объявлении создается трехмерный целочисленный массив размерами 4×10×3.

```
int[,,] multidim = new int[4, 10, 3];
```

А в следующем операторе элементу массива `multidim` с координатами местоположения (2,4,1) присваивается значение 100.

```
multidim[2, 4, 1] = 100;
```

Ниже приведен пример программы, в которой сначала организуется трехмерный массив, содержащий матрицу значений 3×3×3, а затем значения элементов этого массива суммируются по одной из диагоналей матрицы.

```
using System;
class ThreeDMatrix
{
    static void Main()
    {
        int[,,] m = new int[3, 3, 3];
        int sum = 0, n = 1;
        for(int x = 0; x < 3; x++)
            for(int y = 0; y < 3; y++)
                for(int z = 0; z < 3; z++)
                    m[x, y, z] = n++;
```

```
        sum = m[0, 0, 0] + m[1, 1, 1] + m[2, 2, 2];
        Console.WriteLine("Сумма значений по первой диагонали: " + sum);
    }
}
```

Ступенчатые массивы

В приведенных выше примерах применения двумерного массива, по существу, создавался так называемый прямоугольный массив. Двумерный массив можно представить в виде таблицы, в которой длина каждой строки остается неизменной по всему массиву. Но в C# можно также создавать специальный тип двумерного массива, называемый ступенчатым массивом. *Ступенчатый массив* представляет собой массив массивов, в котором длина каждого массива может быть разной. Следовательно, ступенчатый массив может быть использован для составления таблицы из строк разной длины.

Ступенчатые массивы объявляются с помощью ряда квадратных скобок, в которых указывается их размерность. Например, для объявления двумерного ступенчатого массива служит следующая общая форма:

```
тип[][] имя_массива = new тип[размер][];
```

где размер обозначает число строк в массиве. Память для самих строк распределяется индивидуально, и поэтому длина строк может быть разной. Например, в приведенном ниже фрагменте кода объявляется ступенчатый массив `jagged`. Память сначала распределяется для его первого измерения автоматически, а затем для второго измерения вручную.

```
int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];
```

После создания ступенчатого массива доступ к его элементам осуществляется по индексу, указываемому в отдельных квадратных скобках. Например, в следующей строке кода элементу массива `jagged`, находящемуся на позиции с координатами (2,1), присваивается значение 10.

```
jagged[2][1] = 10;
```

Обратите внимание на синтаксические отличия в доступе к элементу ступенчатого и прямоугольного массива.

Ступенчатые массивы представляют собой массивы массивов, и поэтому они не обязательно должны состоять из одномерных массивов. Например, в приведенной ниже строке кода создается массив двумерных массивов.

```
int[][,] jagged = new int[3][,];
```

В следующей строке кода элементу массива `jagged[0]` присваивается ссылка на массив размерами 4×2.

```
jagged[0] = new int[4, 2];
```

А в приведенной ниже строке кода элементу массива `jagged[0][1,0]` присваивается значение переменной `i`.

```
jagged[0][1,0] = i;
```

Типовые задачи на обработку одномерных массивов

В задачах на обработку одномерных массивов наиболее часто встречаются следующие алгоритмы:

1. ***Поиск максимального (минимального) значения в массиве***

Пример. Программа поиска максимального элемента массива.

```
using System;
```

```
class Program
{
    static void Main()
    {
        int n, i, max;
        Console.WriteLine("Введите количество элементов");
        n = Convert.ToInt32(Console.ReadLine());
        int[] array = new int[n];
        Console.WriteLine("Введите элементы массива");
        for (i = 0; i < n; i++)
        {
            Console.Write("array[{0}]=", i);
            array[i] = Convert.ToInt32(Console.ReadLine());
        }
        max = array[0];
        for (i = 1; i < n; i++)
        {
            if (array[i] > max)
                max = array[i];
        }
        Console.WriteLine("Максимальный элемент массива: {0}", max);
        Console.ReadLine();
    }
}
```

Чтобы найти минимальный элемент, достаточно изменить условие в условном операторе на (`array[i] < max`). Конечно, вспомогательную переменную в этом случае лучше назвать `min`. Это необязательно, но неподходящее имя запутает программу и усложнит поиск ошибки.

2. *Поиск индекса максимального (минимального) элемента массива*

Пример. Программа поиска индекса максимального элемента массива.

```
using System;
class Program
{
    static void Main()
    {
        int n, i;
        Console.WriteLine("Введите количество элементов");
        n = Convert.ToInt32(Console.ReadLine());
        int[] array = new int[n];
        Console.WriteLine("Введите элементы массива");
        for (i = 0; i < n; i++)
        {
            Console.Write("array[{0}]=", i);
            array[i] = Convert.ToInt32(Console.ReadLine());
        }
        int max = array[0], imax = 0;
        for (i = 1; i < n; i++)
        {
            if (array[i] > max)
            {
                max = array[i]; imax = i;
            }
        }
        Console.WriteLine("Индекс максимального элемента массива: {0}", imax);
        Console.ReadLine();
    }
}
```

Примечание. Если в массиве есть несколько элементов, равных максимальному, то в результате работы программы будет выведен индекс первого максимального элемента.

При решении данной задачи можно обойтись одной дополнительной переменной. Дело в том, что по номеру элемента можно легко найти его значение в массиве.

```
using System;
class Program
{
    static void Main()
    {
        int n, i;
        Console.WriteLine("Введите количество элементов");
        n = Convert.ToInt32(Console.ReadLine());
        int[] array = new int[n];
        Console.WriteLine("Введите элементы массива");
        for (i = 0; i < n; i++)
        {
            Console.Write("array[{0}]=", i);
            array[i] = Convert.ToInt32(Console.ReadLine());
        }
        int imax = 0;
        for (i = 1; i < n; i++)
        {
            if (array[i] > array[imax])
                imax = i;
        }
        Console.WriteLine("Индекс максимального элемента массива: {0}", imax);
        Console.ReadLine();
    }
}
```

3. Подсчет суммы значений элементов массива, удовлетворяющих заданному условию

Пример. Программа подсчета суммы четных элементов массива.

```
using System;
class Program
{
    static void Main()
    {
        int n, i, S = 0;
        Console.WriteLine("Введите количество элементов");
        n = Convert.ToInt32(Console.ReadLine());
        int[] array = new int[n];
        Console.WriteLine("Введите элементы массива");
        for (i = 0; i < n; i++)
        {
            array[i] = Convert.ToInt32(Console.ReadLine());
        }
        for (i = 0; i < n; i++)
        {
            if (array[i] % 2 == 0)
                S += array[i];
        }
        Console.WriteLine("Сумма четных элементов массива: {0}", S);
        Console.ReadLine();
    }
}
```

4. *Подсчет элементов массива, удовлетворяющих заданному условию*

Пример. Программа подсчета количества отрицательных элементов массива.

```
using System;
class Program
{
    static void Main()
    {
        int n, i, k = 0;
        Console.WriteLine("Введите количество элементов");
        n = Convert.ToInt32(Console.ReadLine());
        int[] array = new int[n];
        Console.WriteLine("Введите элементы массива");
        for (i = 0; i < n; i++)
        {
            array[i] = Convert.ToInt32(Console.ReadLine());
        }
        for (i = 0; i < n; i++)
        {
            if (array[i] < 0)
                k++;
        }
        Console.WriteLine("Количество отрицательных элементов массива: {0}", k);
        Console.ReadLine();
    }
}
```

5. *Поиск значений*

Пример. Программа поиска элемента *x* в массиве из *n* элементов. Значение элемента *x* вводится с клавиатуры.

Решение.

Для решения задачи разумно применить следующий метод – последовательный просмотр массива и сравнение значения очередного рассматриваемого элемента с данным. Если значение очередного элемента совпадает с *x*, то запоминаем его индекс в переменной *k*.

```
for (int i = 0; i < array.Length; i++)
{
    if (array[i] == x)
        k = i;
}
```

Этот способ решения поставленной задачи, безусловно, приводит к цели, но обладает рядом существенных недостатков:

- если значение *x* встречается в массиве несколько раз, то найдено будет последнее из них;
- после того, как нужное значение уже найдено, массив просматривается до конца, то есть всегда выполняется *n* сравнений.

Разумно прекратить просмотр сразу после обнаружения заданного элемента или до конца массива. Таким образом, условие окончания цикла может выглядеть так:

```
for (int i = 0, k = -1; i < array.Length; i++)
{
    if (array[i] == x)
    {
        k = i; break;
    }
}
```

или так:

```
i=0;
while (i < array.Length && array[i] != x)
{
    i++;
}
```

Примечание. Если в массиве есть несколько элементов, совпадающих с элементом *x*, в результате работы программы будет найден первый из них, то есть элемент с наименьшим индексом.

```
using System;
class Program
{
    static void Main()
    {
        int n, i, x;
        Console.WriteLine("Введите количество элементов");
        n = Convert.ToInt32(Console.ReadLine());
        int[] array = new int[n];
        Console.WriteLine("Введите элементы массива");
        for (i = 0; i < n; i++)
        {
            Console.Write("array[{0}]=", i);
            array[i] = Convert.ToInt32(Console.ReadLine());
        }
        Console.WriteLine("Введите искомый элемент");
        x = Convert.ToInt32(Console.ReadLine());
        i = 0;
        while (i < n && array[i] != x)
        {
            i++;
        }
        if (i < n)
            Console.WriteLine("Индекс искомого элемента: {0}", i);
        else
            Console.WriteLine("Данный элемент в массиве отсутствует");
        Console.ReadLine();
    }
}
```

Типовые задачи на обработку двумерных массивов

В задачах на обработку двумерных массивов наиболее часто встречаются следующие алгоритмы:

- обработка всего массива;
- обработка отдельно по строкам и столбцам;
- обработка относительно диагоналей.

Для обработки двумерных массивов могут применяться методы решения задач, рассмотренные в пункте «Типовые задачи на обработку одномерных массивов». В отличие от одномерных массивов, для перебора всех элементов двумерного массива надо использовать двойной цикл.

Пример. Программа поиска индексов максимального элемента двумерного массива.

```
using System;
class Program
{
    static void Main()
    {
        int m, n;
```

```
        Console.WriteLine("Введите количество строк");
        m = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Введите количество столбцов");
        n = Convert.ToInt32(Console.ReadLine());
        int[,] matr = new int[m, n];
        int i, j, max, imax, jmax;
        for (i = 0; i < m; i++)
        {
            Console.WriteLine("Введите элементы {0}-й строки", i + 1);
            for (j = 0; j < n; j++)
                matr[i, j] = Convert.ToInt32(Console.ReadLine());
        }
        max = matr[0, 0]; imax = 0; jmax = 0;
        for (i = 0; i < m; i++)
        {
            for (j = 0; j < n; j++)
                if (matr[i, j] > max)
                {
                    max = matr[i, j]; imax = i; jmax = j;
                }
        }
        Console.WriteLine("Максимальный элемент matr[{0},{1}]={2}",imax,jmax,max);
        Console.ReadLine();
    }
}
```

Пример. Двумерный целочисленный массив размером $m \times n$ заполнить с клавиатуры.

1. Вывести массив на экран в виде таблицы.
2. Подсчитать сумму элементов каждой строки.
3. Определить максимальные значения для каждого столбца.

```
using System;
class Program
{
    static void Main()
    {
        int m, n;
        Console.WriteLine("Введите количество строк");
        m = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Введите количество столбцов");
        n = Convert.ToInt32(Console.ReadLine());
        int[,] matr = new int[m, n];
        int i, j, S, max;
        for (i = 0; i < m; i++)
        {
            Console.WriteLine("Введите элементы {0}-й строки", i + 1);
            for (j = 0; j < n; j++)
                matr[i, j] = Convert.ToInt32(Console.ReadLine());
        }
        Console.WriteLine("Введенный массив:");
        for (i = 0; i < m; i++)
        {
            for (j = 0; j < n; j++)
                Console.Write("{0,6}", matr[i, j]);
            Console.WriteLine();
        }
        for (i = 0; i < m; i++)
        {
            S = 0;
```

```
            for (j = 0; j < n; j++)
                S += matr[i, j];
            Console.WriteLine("Сумма элементов {0}-й строки: {1}", i+1, S);
        }
        for (j = 0; j < n; j++)
        {
            max = matr[0, j];
            for (i = 1; i < m; i++)
                if (matr[i, j] > max)
                    max = matr[i, j];
            Console.WriteLine("Максимальный элемент {0}-го столбца: {1}",j+1,max);
        }
        Console.ReadLine();
    }
}
```

Пример. Найти сумму элементов квадратной матрицы, лежащих выше главной диагонали (рис. 3).

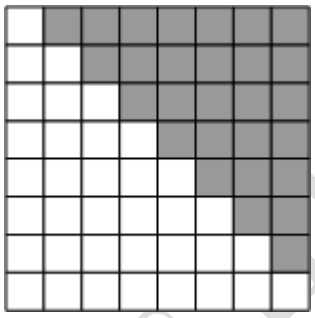


Рис. 3. *Рисунок к условию задачи из примера*

Алгоритм решения данной задачи построен следующим образом: с помощью двух циклов (первый по строкам, второй по столбцам) просматривается каждый элемент матрицы, но суммирование происходит только в том случае, если этот элемент находится выше главной диагонали, то есть выполняется свойство $i < j$.

```
using System;
class Program
{
    static void Main()
    {
        const int n = 5;
        int i, j, S = 0;
        int[,] matr = new int[n, n];
        Random rand = new Random();
        Console.WriteLine("Заполненный массив:");
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < n; j++)
            {
                matr[i, j] = rand.Next(0, 100);
                Console.Write("{0,6}", matr[i, j]);
            }
            Console.WriteLine();
        }
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (i < j)
                    S += matr[i, j];
        Console.WriteLine("Сумма элементов выше главной диагонали: {0}", S);
        Console.ReadLine();
```

```
    }
}
```

Возможен еще один вариант решения данной задачи. В нем проверка условия $i < j$ не выполняется, но, тем не менее, в нем также суммируются элементы матрицы, находящиеся выше главной диагонали. В первой строке заданной матрицы необходимо сложить все элементы, начиная со второго. Во второй – все, начиная с третьего, в i-й строке процесс начнется с (i+1)-го элемента и так далее.

```
for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
        S += matr[i, j];
Console.WriteLine("Сумма элементов выше главной диагонали: {0}", S);
```

Класс System.Array

Все массивы в C# являются производными от базового класса `System.Array`. Класс `System.Array` наследует ряд интерфейсов: `ICloneable, IList, ICollection, Enumerable`, и обязан реализовать все их методы и свойства. Помимо наследования свойств и методов класса `Object` и вышеперечисленных интерфейсов, класс `Array` имеет довольно большое число собственных методов и свойств, которые упрощают работу программиста.

Таблица 1. Некоторые члены класса System.Array

Член класса	Вид	Назначение
`Length`	свойство	Возвращает количество элементов данного массива (по всем размерностям)
`Rank`	свойство	Возвращает размерность массива
`GetLength()`	экземплярный метод	Возвращает количество элементов для заданного в качестве аргумента измерения массива. Нумерация измерений начинается с нуля.
`GetLowerBound(), GetUpperBound()`	экземплярные методы	Возвращает нижнюю и верхнюю границу по указанному измерению. Для массивов нижняя граница всегда равна нулю.
`GetValue(), SetValue()`	экземплярные методы	Возвращает или устанавливает значение элемента массива с указанными индексами.
`IndexOf, LastIndexOf`	статические методы	Определяют индексы первого и последнего вхождения элемента в массиве, равного данному значению. Возвращает -1, если такового вхождения не обнаружено
`BinarySearch()`	статический метод	Возвращает индекс первого вхождения элемента в отсортированном массиве, используя алгоритм двоичного поиска.
`Clear()`	статический метод	Происходит обнуление значений массива, т.е. каждому элементу присваивается значение по умолчанию
`Clone()`	статический метод	Создает неглубокую копию массива. Если элементы массива относятся к типу значений, то все они копируются. Если массив содержит элементы ссылочных типов, то сами эти элементы не копируются, а копируются лишь ссылки на них.
`Copy()`	статический метод	Позволяет копировать весь массив или его часть в другой массив.
`CopyTo()`	экземплярный метод	Копируются все элементы одномерного массива в другой одномерный массив, начиная с заданного индекса.
`Reverse()`	статический метод	Выполняет обращение одномерного массива, переставляя элементы в обратном порядке.
`Sort()`	статический метод	Осуществляет сортировку одномерного массива.

Для вызова статического метода в программном коде нужно придерживаться следующего примера – `Array.Sort(имя_массива)`, где `Array` – класс, в котором содержится метод `Sort`, который в свою очередь будет сортировать ваш массив с именем `имя_массива`.

Обращение к свойству или вызов экземплярного метода производится через обращение к экземпляру класса, например,

```
имя_массива.свойство
```

или

```
имя_массива.экземплярный_метод(аргумент).
```

Цикл foreach

Цикл `foreach` служит для циклического обращения к элементам коллекции, представляющей собой группу объектов. Ниже приведена общая форма оператора цикла `foreach`:

```
foreach (тип имя_переменной_цикла in коллекция)
   оператор;
```

Здесь *тип имя_переменной_цикла* обозначает тип и имя переменной управления циклом, которая получает значение следующего элемента коллекции на каждом шаге выполнения цикла `foreach`. А *коллекция* обозначает циклически опрашиваемую коллекцию, которая здесь и далее представляет собой массив. Следовательно, тип переменной цикла должен соответствовать типу элемента массива.

В любой точке блока `foreach` можно разорвать цикл с помощью ключевого слова `break` или перейти к следующей итерации в цикле с помощью ключевого слова `continue`.

Пример.

```
int[] array = new int[] { 4, 8, 6, 9, 5 };
foreach (int x in array)
{
    Console.WriteLine(x);
}
```

Подобные действия мы можем сделать и с помощью цикла `for`:

```
int[] array = new int[] { 4, 8, 6, 9, 5 };
for (int i = 0; i < array.Length; i++)
{
    Console.WriteLine(array[i]);
}
```

Пример.

```
int[,] myTwoArr = new int[,] { { 4, 8, 9 }, { 6, 9, 5 }, { 3, 7, 1 } };
int sum = 0;
// Вычисление суммы элементов двумерного массива
foreach (int elem in myTwoArr)
    sum += elem;
Console.WriteLine("Сумма элементов двумерного массива: {0}", sum);
```

В то же время цикл `for` более гибкий по сравнению с `foreach`. Если `foreach` последовательно извлекает элементы коллекции и только для чтения, то в цикле `for` мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы. Например:

```
int[] array = new int[] { 4, 8, 6, 9, 5 };
for (int i = 0; i < array.Length; i++)
{
    array[i] = array[i] * 2;
    Console.WriteLine(array[i]);
}
```

Коллекции

Введение в коллекции

В C# ***коллекция*** представляет собой совокупность объектов. В среде `.NET Framework` имеется немало интерфейсов и классов, в которых определяются и реализуются различные типы коллекций. Коллекции упрощают решение многих задач программирования благодаря тому, что предлагают готовые решения для создания целого ряда типичных, но порой трудоемких для разработки структур данных. Например, в среду `.NET Framework` встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.

Первоначально существовали только классы необобщенных коллекций. Но с внедрением обобщений в версии C# 2.0 среда `.NET Framework` была дополнена многими новыми обобщенными классами и интерфейсами. Благодаря введению обобщенных коллекций общее количество классов и интерфейсов удвоилось. Вместе с библиотекой распараллеливания задач (TPL) в версии 4.0 среды `.NET Framework` появился ряд новых классов коллекций, предназначенных для применения в тех случаях, когда доступ к коллекции осуществляется из нескольких потоков.

В среде `.NET Framework` поддерживаются пять типов коллекций: необобщенные, обобщенные, специальные, с поразрядной организацией и параллельные.

Необобщенные коллекции

Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". В отношении необобщенных коллекций важно иметь в виду следующее: они оперируют данными типа `object`. Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных. Очевидно, что такие коллекции не типизированы, поскольку в них хранятся ссылки на данные типа `object`. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен `System.Collections`.

Обобщенные коллекции

Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен `System.Collections.Generic`.

Специальные коллекции

Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен `System.Collections.Specialized`.

Поразрядная коллекция

В прикладном интерфейсе `Collections API` определена одна коллекция с поразрядной организацией – это `BitArray`. Коллекция типа `BitArray` поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например И, ИЛИ, исключающее ИЛИ, а следовательно, она существенно отличается своими возможностями от остальных типов коллекций. Коллекция типа `BitArray` объявляется в пространстве имен `System.Collections`.

Параллельные коллекции

Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен `System.Collections.Concurrent`.

В пространстве имен `System.Collections.ObjectModel` находится также ряд классов, поддерживающих создание пользователями собственных обобщенных коллекций.

Основополагающим для всех коллекций является понятие *перечислителя*, который поддерживается в необобщенных интерфейсах `IEnumerator` и `IEnumerable`, а также в обобщенных интерфейсах `IEnumerator<T>` и `IEnumerable<T>`. Перечислитель обеспечивает стандартный способ поочередного доступа к элементам коллекции. Следовательно, он перечисляет содержимое коллекции. В каждой коллекции должна быть реализована обобщенная или необобщенная форма интерфейса `IEnumerable`, поэтому элементы любого класса коллекции должны быть доступны посредством методов, определенных в интерфейсе `IEnumerator` или `IEnumerator<T>`. Для поочередного обращения к содержимому коллекции в цикле `foreach` используется перечислитель.

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. Таким образом, они позволяют манипулировать объектом любого типа, хотя и не типизированным способом. В этом состоит их преимущество и в то же время недостаток. 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 неизвестны. Но если коллекция предназначается для хранения объекта конкретного типа, то необобщенные коллекции не обеспечивают типовую безопасность, которую можно обнаружить в обобщенных коллекциях.

Классы необобщенных коллекций:

`ArrayList` – определяет динамический массив, расширяющийся и сокращающийся по мере необходимости

`Hashtable` – определяет хеш-таблицу для пар "ключ-значение"

`Queue` – определяет очередь, или список, действующий по принципу "первым пришел – первым обслужен" (FIFO – First In, First Out)

`SortedList` – определяет отсортированный список пар "ключ-значение"

`Stack` – определяет стек, или список, действующий по принципу "первым пришел – последним обслужен" (FILO – First In, Last Out)

Пример.

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // необобщенная коллекция ArrayList
        ArrayList myList = new ArrayList() { 1, 2, "sun", 'c', 2.5 };
        object x = 45.8;
        myList.Add(x);          // добавление в конец списка объекта типа double
        myList.Add("day");      // добавление в конец списка объекта типа string
        myList.RemoveAt(0);     // удаление первого элемента
        foreach (object ob in myList)
        {
            Console.WriteLine(ob);
        }
        Console.WriteLine("Общее число элементов коллекции: " + myList.Count);
        Console.ReadLine();
    }
}
```

Обобщенные коллекции

Благодаря внедрению обобщений прикладной интерфейс `Collections API` значительно расширился, в результате чего количество классов коллекций и интерфейсов удвоилось. Обобщенные коллекции объявляются в пространстве имен `System.Collections.Generic`. Как правило, классы обобщенных коллекций являются не более чем обобщенными эквивалентами классов необобщенных коллекций, хотя это соответствие не является взаимно однозначным. Например, в классе обобщенной коллекции `LinkedList` реализуется двунаправленный список, тогда как в необобщенном эквиваленте его не существует. В некоторых случаях одни и те же функции существуют параллельно в классах обобщенных и необобщенных коллекций, хотя и под разными именами. Так, обобщенный вариант класса `ArrayList` называется `List`, а обобщенный вариант класса `HashTable` – `Dictionary`. Кроме того, конкретное содержимое различных интерфейсов и классов реорганизуется с минимальными изменениями для переноса некоторых функций из одного интерфейса в другой. Но в целом, имея ясное представление о необобщенных коллекциях, можно без особого труда научиться применять и обобщенные коллекции.

Как правило, обобщенные коллекции действуют по тому же принципу, что и необобщенные, за исключением того, что обобщенные коллекции типизированы. Это означает, что в обобщенной коллекции можно хранить только те элементы, которые совместимы по типу с ее аргументом. Так, если требуется коллекция для хранения несвязанных друг с другом разнотипных данных, то для этой цели следует использовать классы необобщенных коллекций. А во всех остальных случаях, когда в коллекции должны храниться объекты только одного типа, выбор рекомендуется останавливать на классах обобщенных коллекций.

Классы обобщенных коллекций:

`List<T>` – создает последовательный список. Обеспечивает такие же функциональные возможности, как и необобщенный класс `ArrayList`

`LinkedList<T>` – сохраняет элементы в двунаправленном списке

`Dictionary<Tkey, TValue>` – сохраняет пары "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс `Hashtable`

`Queue<T>` – создает очередь. Обеспечивает такие же функциональные возможности, как и необобщенный класс `Queue`

`Stack<T>` – создает стек. Обеспечивает такие же функциональные возможности, как и необобщенный класс `Stack`

`HashSet<T>` – сохраняет ряд уникальных значений, используя хеш-таблицу

`SortedList<TKey, TValue>` – создает отсортированный список из пар "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс `SortedList`

`SortedDictionary<TKey, TValue>` – создает отсортированный список из пар "ключ-значение". Похож на класс `SortedList<TKey, TValue>`, основные отличия состоят лишь в использовании памяти и в скорости вставки и удаления

`SortedSet<T>` – создает отсортированное множество

Пример.

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

class Program
{
    public static void Main()
    {
        // обобщенная коллекция List
        List<string> countries = new List<string>() { "Россия", "США", "Великобритания" };
        countries.Add("Франция");   // добавление элемента в конец
```

```
        countries.RemoveAt(1);      // удаление второго элемента
        foreach (string s in countries)
        {
            Console.WriteLine(s);
        }
        Console.ReadLine();
    }
}
```

Списки

Список List<T>

Класс `List<T>` представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

- `Add()` – добавляет новый элемент в список
- `AddRange()` – добавляет в список коллекцию или массив
- `BinarySearch()` – бинарный поиск элемента в отсортированном списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции.
- `IndexOf()` – возвращает индекс первого вхождения элемента в списке
- `Insert()` – вставляет элемент в список на указанную позицию
- `InsertRange()` – вставляет в список на указанную позицию коллекцию или массив
- `Remove()` – удаляет элемент из списка, и если удаление прошло успешно, то возвращает true
- `RemoveAt()` – удаляет элемент по указанному индексу
- `Contains()` – возвращает значение true, если элемент найден в списке, в противном случае – значение false
- `Sort()` – сортирует список
- `Reverse()` – реверсирует список

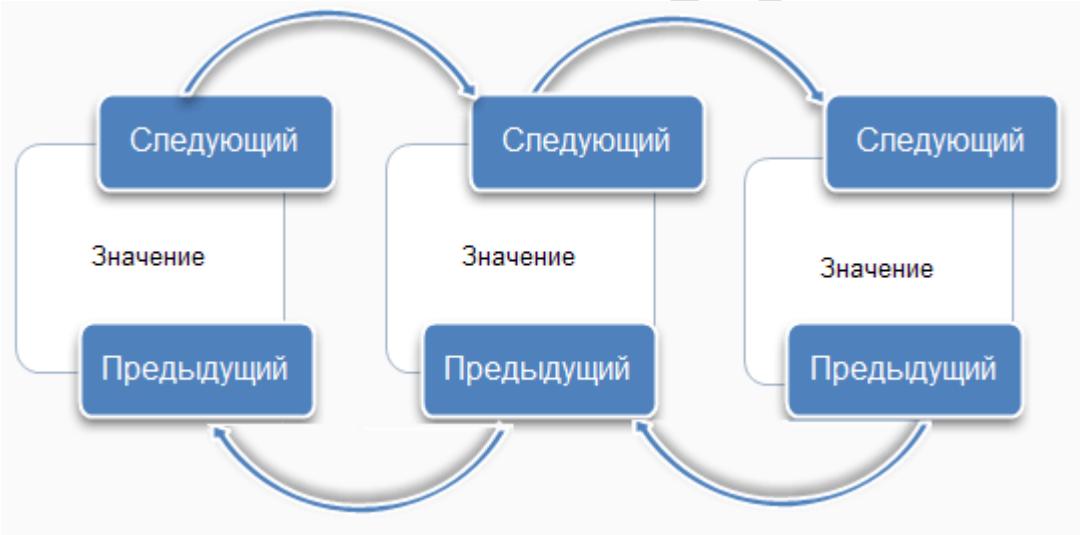
Пример.

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

class Program
{
    public static void Main()
    {
        List<int> numbers = new List<int>() { 4, 2, 3 };
        numbers.Add(6);             //  4, 2, 3, 6
        numbers.AddRange(new int[] { 7, 8, 9 });   //  4, 2, 3, 6, 7, 8, 9
        numbers.Insert(0, 66);      //  66, 4, 2, 3, 6, 7, 8, 9
        numbers.RemoveAt(1);        //  66, 2, 3, 6, 7, 8, 9
        numbers.Remove(8);          //  66, 2, 3, 6, 7, 9
        numbers.InsertRange(2, new int[] { 4, 5 }); //  66, 2, 4, 5, 3, 6, 7, 9
        numbers.Sort();             //  2, 3, 4, 5, 6, 7, 9, 66
        numbers.Reverse();          //  66, 9, 7, 6, 5, 4, 3, 2
        foreach (int i in numbers) // вывод элементов списка в столбик
        {
            Console.WriteLine(i);
        }
        // вывод элементов в виде: 66, 9, 7, 6, 5, 4, 3, 2
        Console.WriteLine(string.Join(", ", numbers));
        Console.WriteLine("Количество элементов: " + numbers.Count); // 8
        int[] arr = numbers.ToArray(); // преобразование в массив

        List<Person> people = new List<Person>(); // пустой список
        Person p1 = new Person("Иван", 20);
        people.Add(p1);
```

```
            people.Add(new Person("Сергей", 25));
            foreach (Person p in people)
            {
                Console.WriteLine("Имя: {0}\nВозраст: {1}", p.Name, p.Age);
            }

            Console.ReadLine();
        }

        class Person
        {
            public string Name;
            public int Age;

            public Person(string name, int age)
            {
                Name = name;
                Age = age;
            }
        }
}
```

Связный список LinkedList <T>

Класс `LinkedList<T>` представляет собой двухсвязный список, в котором каждый элемент ссылается на следующий и предыдущий, как показано на рисунке:



Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. При этом только ссылки `Next` (следующий) предыдущего элемента и `Previous` (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент. В классе `List<T>` при вставке нового элемента все последующие должны быть сдвинуты.

Естественно, у связных списков есть и свои недостатки. Так, например, все элементы связных списков доступны лишь друг за другом. Поэтому для нахождения элемента, находящегося в середине или конце списка, требуется довольно много времени. Связный список не может просто хранить элементы внутри себя. Вместе с каждым из них ему необходимо иметь информацию о следующем и предыдущем элементах. Вот почему `LinkedList<T>` содержит элементы типа `LinkedListNode<T>`. С помощью класса `LinkedListNode<T>` появляется возможность обратиться к предыдущему и последующему элементам списка. Класс `LinkedListNode<T>` определяет свойства `List`, `Next`, `Previous` и `Value`. Свойство `List` возвращает объект `LinkedList<T>`, ассоциированный с узлом. Свойства `Next` и `Previous` предназначены для итераций по списку и для доступа к следующему и

предыдущему элементам. Свойство `Value` типа `T` возвращает элемент, ассоциированный с узлом.

Сам класс `LinkedList<T>` определяет члены для доступа к первому (`First`) и последнему (`Last`) элементам в списке, для вставки элементов в определенные позиции (`AddAfter()`, `AddBefore()`, `AddFirst()`, `AddLast()`), для удаления элементов из заданных позиций (`Remove()`, `RemoveFirst()`, `RemoveLast()`) и для нахождения элементов, начиная поиск либо с начала (`Find()`), либо с конца (`FindLast()`) списка.

В классе `LinkedList<T>` определяется немало методов. Наиболее часто используемые методы, определенные в классе `LinkedList<T>` представлены ниже:

- `AddAfter()` – добавляет в список узел со значением непосредственно после указанного узла. Указываемый узел не должен быть пустым (`null`). Метод возвращает ссылку на узел, содержащий значение.
- `AddBefore()` – добавляет в список узел со значением value непосредственно перед указанным узлом. Указываемый узел не должен быть пустым (`null`). Метод возвращает ссылку на узел, содержащий значение.
- `AddFirst()`, `AddLast()` – добавляют узел со значением в начало или в конец списка.
- `Find()` – возвращает ссылку на первый узел в списке, имеющий передаваемое значение. Если искомое значение отсутствует в списке, то возвращается пустое значение.
- `Remove()` – удаляет из списка первый узел, содержащий передаваемое значение. Возвращает логическое значение `true`, если узел удален, т.е. если узел со значением обнаружен в списке и удален; в противном случае возвращает логическое значение `false`.

Пример.

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

class Program
{
    public static void Main()
    {
        LinkedList<int> numbers = new LinkedList<int>();
        numbers.AddLast(1); // вставляем узел со значением 1 на последнее место
        // так как в списке нет узлов, то последнее будет также и первым
        numbers.AddFirst(2); // вставляем узел со значением 2 на первое место
        numbers.AddAfter(numbers.Last, 3);    // вставляем после последнего узла
                                              // новый узел со значением 3
        // теперь список имеет следующую последовательность: 2, 1, 3
        numbers.AddBefore(numbers.Find(1), 5); // вставляем узел со значением 5
                                               //перед узлом со значением 1
        // теперь список имеет следующую последовательность: 2, 5, 1, 3
        foreach (int i in numbers)
        {
            Console.WriteLine(i);
        }
    }
}
```

Очередь

Очередь (queue) – это коллекция, в которой элементы обрабатываются по схеме "первый вошел, первый вышел" (first in, first out – FIFO). Элемент, вставленный в очередь первым, первым же и читается. Примерами очередей могут служить очередь в аэропорту, очередь претендентов на трудоустройство, очередь печати принтера либо циклическая очередь потоков на выделение ресурсов процессора. Часто встречаются очереди, в которых элементы обрабатываются по-разному, в соответствии с приоритетом. Например, в очереди в аэропорту пассажиры бизнес-класса обслуживаются перед пассажирами эконом-класса. Здесь может

использоваться несколько очередей – по одной для каждого приоритета. В аэропорту это можно видеть наглядно, поскольку там предусмотрены две стойки регистрации для пассажиров бизнес-класса и эконом-класса. То же справедливо и для очередей печати и диспетчера потоков. У вас может быть массив списка очередей, где элемент массива означает приоритет. Внутри каждого элемента массива будет очередь, и обработка будет выполняться по принципу FIFO.

Очередь реализуется с помощью классов `Queue` из пространства имен `System.Collections` и `Queue<T>` из пространства имен `System.Collections.Generic`.

У класса `Queue<T>` можно отметить следующие методы:

- `Dequeue()` – извлекает и возвращает первый элемент очереди
- `Enqueue()` – добавляет элемент в конец очереди
- `Peek()` – просто возвращает первый элемент из начала очереди без его удаления

Пример.

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

class Program
{
    public static void Main()
    {
        Queue<int> numbers = new Queue<int>();
        numbers.Enqueue(3);             // очередь: 3
        numbers.Enqueue(5);             // очередь: 3, 5
        numbers.Enqueue(8);             // очередь: 3, 5, 8
        // извлекаем первый элемент из очереди
        int queueElement = numbers.Dequeue();       //теперь очередь: 5, 8
        Console.WriteLine(queueElement);      // выводит 3
        Console.WriteLine(numbers.Count);     // количество элементов очереди: 2
    }
}
```

Стек

Стек (stack) – это контейнер, работающий по принципу "последний вошел, первый вышел" (last in, first out – LIFO). 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<T>` можно выделить следующие методы, 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

Пример.

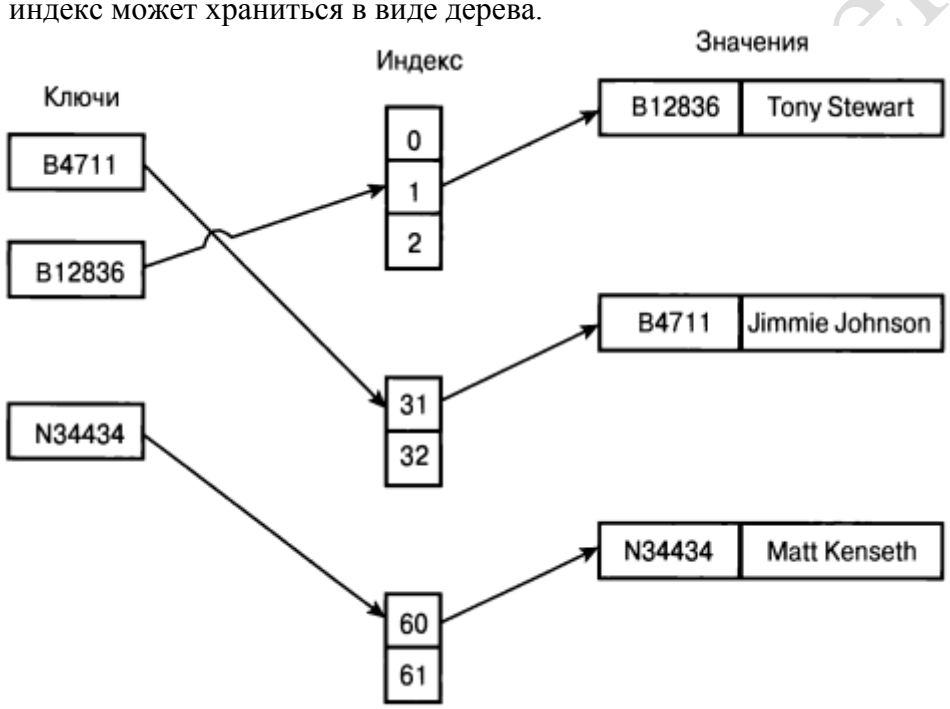
```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        Stack<int> numbers = new Stack<int>() ;
        numbers.Push(3); // содержимое стека: 3
        numbers.Push(5); // содержимое стека: 5, 3
        numbers.Push(8); // содержимое стека: 8, 5, 3
        Console.WriteLine(numbers.Peek()); // выводит 8, содержимое стека: 8, 5, 3
        Console.WriteLine(numbers.Pop());  // выводит 8, содержимое стека: 5, 3
```

```
        Console.WriteLine(numbers.Pop());  // выводит 5, содержимое стека:  3
    }
}
```

Словарь

Словарь (dictionary) представляет собой сложную структуру данных, позволяющую обеспечить доступ к элементам по ключу. Главное свойство словарей – быстрый поиск на основе ключей. Можно также свободно добавлять и удалять элементы, подобно тому, как это делается в `List<T>`, но без накладных расходов производительности, связанных с необходимостью смещения последующих элементов в памяти.

На следующем рисунке представлена упрощенная модель словаря. Здесь ключами словаря служат идентификаторы сотрудников, такие как B4711. Ключ трансформируется в хеш. В хеше создается число для ассоциации индекса со значением. После этого индекс содержит ссылку на значение. Изображенная модель является упрощенной, поскольку существует возможность того, что единственное вхождение индекса может быть ассоциировано с несколькими значениями, и индекс может храниться в виде дерева.



В `.NET Framework` предлагается несколько классов словарей. Главный класс, который можно использовать – это `Dictionary<TKey, TValue>`.

В классе `Dictionary<TKey, TValue>` определяется ряд методов:

- `Add()` – добавляет в словарь пару "ключ-значение", определяемую параметрами `key` и `value`. Если ключ `key` уже находится в словаре, то его значение не изменяется, и генерируется исключение `ArgumentException`
- `ContainsKey()` – возвращает логическое значение `true`, если вызывающий словарь содержит объект `key` в качестве ключа; а иначе – логическое значение `false`
- `ContainsValue()` – возвращает логическое значение `true`, если вызывающий словарь содержит значение `value`; в противном случае – логическое значение `false`
- `Remove()` – удаляет ключ `key` из словаря. При удачном исходе операции возвращается логическое значение `true`, а если ключ `key` отсутствует в словаре – логическое значение `false`

Пример.

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

class Program
{
```

```
    public static void Main()
    {
        Dictionary<int, string> countries = new Dictionary<int, string>();
        countries.Add(1, "Russia");
        countries.Add(3, "Great Britain");
        countries.Add(2, "USA");
        countries.Add(4, "France");
        countries.Add(5, "China");
        foreach (KeyValuePair<int, string> pair in countries)
            Console.WriteLine("{0} - {1}", pair.Key, pair.Value);
    }
}
```

Словари можно инициализировать следующим образом:

```
Dictionary<string, string> countries = new Dictionary<string, string>
{
    {"Франция", "Париж"},
    {"Германия", "Берлин"},
    {"Великобритания", "Лондон"}
};
```

Множества

Коллекция, содержащая только отличающиеся элементы, называется ***множеством*** (set). В составе .NET 4 имеются два множества – `HashSet<T>` и `SortedSet<T>`. Класс `HashSet<T>` содержит неупорядоченный список различающихся элементов, а в `SortedSet<T>` элементы упорядочены.

Некоторые методы класса `HashSet<T>`:

- `UnionWith()` – добавляет элементы множества в коллекцию (исключая дубликаты)
- `IntersectWith()` – удаляет элементы, которые не присутствую сразу в обоих наборах
- `ExceptWith()` – удаляет переданные элементы из коллекции
- `SymmetricExceptWith()` – удаляет все элементы, за исключением уникальных в одном или другом наборе

Пример.

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

class Program
{
    public static void Main()
    {
        HashSet<int> s1 = new HashSet<int>();
        HashSet<int> s2 = new HashSet<int>();
        s1.Add(1);
        s1.Add(3);
        s1.Add(5);
        s1.Add(7);
        Console.WriteLine("Первое множество: " + string.Join(", ", s1)); // 1, 3, 5, 7
        s2.Add(5);
        s2.Add(6);
        s2.Add(7);
        Console.WriteLine("Второе множество: " + string.Join(", ", s2)); // 5, 6, 7
        s1.UnionWith(s2);
        Console.WriteLine("Объединение множеств: " + string.Join(", ", s1)); // 1, 3, 5, 7, 6
        s1.ExceptWith(s2);
        Console.WriteLine("Вычитание множеств: " + string.Join(", ", s1)); // 1, 3
    }
}
```

LINQ

Linq – это одновременно и библиотека, и язык, позволяющий выполнять запросы к коллекциям или базам данных. Эта технология одинаково хорошо работает со всеми встроенными коллекциями: массивами, списками и т. д.

При подключении пространства имён System.Linq у всех объектов, реализующих обобщённый интерфейс IEnumerable<T> (этот интерфейс реализуют все коллекции) появляются дополнительные методы. Некоторые из них приведены в таблице ниже.

Метод	Описание	Пример
`Aggregate`	Объединить элементы коллекции с помощью указанной операции и начального значения.	// Поиск произведения `c.Aggregate(1, (x, y)=>x * y)`
`All`	Проверяет, выполняется ли указанное условие для всех элементов коллекции.	// Все ли чётные? `c.All(x=>x%2==0)`
`Any`	Проверяет, выполняется ли указанное условие хотя бы для одного элемента коллекции.	// Есть ли хоть один чётный? `c.Any(x=>x%2==0)`
`Average`	Вычисляет арифметическое среднее элементов числовой коллекции.	`c.Average()`
`Contains`	Проверяет, содержит ли коллекция указанное значение.	// Есть ли пятёрка? `c.Contains(5)`
`Distinct`	Возвращает копию коллекции, в которой повторяющиеся элементы удалены.	`c.Distinct()`
`Except`	Удаляет из коллекции элементы с указанными значениями.	// Удалить все 2 и 7 `c.Except(new[]{2, 7})`
`First`	Возвращает первый элемент коллекции.	`c.First()`

`Intersect`	Возвращает пересечение текущей коллекции и указанной как аргумент.	// Пересечение с массивом
		// {1, 2, 3}
		`c.Intersect(new[]{1, 2, 3})`

`Last`	Последний элемент коллекции.	`c.Last()`
`Max`	Максимальный элемент коллекции.	`c.Max()`
`Min`	Минимальный элемент коллекции.	`c.Min()`

`OrderBy`	Возвращает коллекцию, упорядоченную по возрастанию значений указанной функции от элементов.	// Упорядочить по возрастанию
		// последней цифры
		`c.OrderBy(x=>x%10)`

`Reverse`	Меняет порядок элементов на противоположный.	`c.Reverse()`

`Select`	Возвращает новую коллекцию, формируя её из элементов текущей, применяя к каждому из них некоторую функцию.	// Получить коллекцию из
		// квадратов элементов данной
		`c.Select(x=>x*x)`

`Skip`	Удаляет из коллекции указанное число первых элементов.	// Выбросить первые 5 чисел `c.Skip(5)`
`Sum`	Сумма элементов.	`c.Sum()`
`Take`	Формирует новую коллекцию из указанного числа первых элементов текущей коллекции.	// Оставить первые 5 чисел `c.Take(5)`
`Where`	Фильтрует коллекцию по указанному	// Оставить только чётные

Метод	Описание	Пример
	критерию.	`c.Where(x=>x%2==0)`

Одна из особенностей Linq – повсеместное использование лямбда-функций. Так, например, выбор чётных элементов коллекции выполняется следующим образом:

```
var c = new List<int>() {3, 7, 9, 4, 5, 6, 6, 1, 2};
var c2 = c.Where(x => x % 2 == 0);
Console.WriteLine (string.Join(" ", c2)); // 4 6 6 2
```

Указанная в качестве параметра лямбда-функция применяется к каждому элементу коллекции. Если результат её вычисления равен истине, то элемент добавляется к результату, если нет, то пропускается.

Вообще говоря, точный тип коллекции-результата может быть неизвестен на этапе компиляции. Поэтому для объявления переменной с результатом обычно используют `var`. Часто это неважно, но если нужно преобразовать результат в массив, список или словарь, то можно воспользоваться методами `ToArray`, `ToList` или `ToDictionary` соответственно.

Рассмотренные методы можно объединять в одно выражение, где методы вызываются «по цепочке». Такой способ записи называется точечной нотацией.

Рассмотрим следующую задачу. В коллекции требуется:

- оставить только элементы, больше 3;
- упорядочить по возрастанию последней цифры квадрата;
- поменять порядок элементов на противоположный;
- заменить элементы их квадратами;
- оставить 5 элементов начиная с элемента с индексом 3;
- убрать повторяющиеся элементы;
- найти сумму.

Эту задачу можно решить, фактически одной строкой. Ниже приведена полная программа с решением этой задачи.

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

class Program
{
    public static void Main()
    {
        var c = new List<int>() {3, 7, 9, 4, 5, 6, 6, 1, 2};
        var c2 = c
            .Where(x => x > 3)
            .OrderBy(x => (x * x) % 10).Reverse()
            .Select(x => x * x)
            .Skip(3)
            .Take(5)
            .Distinct()
            .Sum();
        Console.WriteLine (string.Join(" ", c2));
    }
}
```

Обилие лямбда-функций часто затрудняет чтение программы. Поэтому для сложных запросов удобно использовать так называемые Linq-выражения. Они пишутся на специальном языке запросов, являющимся частью языка C#.

Рассмотрим типичный вид запроса, решающего предыдущую задачу (порядок частей важен, хотя некоторые из них необязательны).

```
var c2 = (from x in c                  // для всех x в c,
    where x > 3                        // где x > 3,
    orderby (x * x) % 10 descending    // упорядочен
                                       // по посл. цифре квадрата
                                       // в порядке убывания,
    select x * x)                      // выбрать x * x,
    .Skip(3)                           // пропустить 3
    .Take(5)                           // взять первые 5,
    .Distinct()                        // убрать повторы,
    .Sum();                            // найти сумму
```

Выражение в первых скобках и есть Linq-выражение.

Задания для самостоятельной работы

Разработайте приложения для решения задач из сборника задач по программированию согласно вашему варианту.