Урок №6

Содержание

1. Понятие коллекции

И снова тема, которую с трудом можно назвать абсолютно новой, поскольку в языке C++ вы уже сталкивались с подобным явлением.

Необходимость возможности группирования данных неоспорима. Как правило, это первое, что обсуждается при рассмотрении темы массивы, в чём мы тоже не оригинальны. Любой набор данных, объектов можно назвать коллекцией. Массив из пространства имён System.Array – это тоже своего рода коллекция, но статичная, его нельзя ни расширить, ни сжать при необходимости, а это не всегда удобно.

.NET представляет множество классов коллекций, позволяющих работать с наборами данных более гибко, чем массив. Что собой представляет коллекция? Это группа элементов. В .NET коллекции содержат объекты, в том числе и упакованные типы значений. Каждый объект, содержащийся в коллекции, называется её элементом. Некоторые коллекции хранят данные как прямой список элементов, другие содержат списки пар ключей и значений. К первым относятся классы System.Collections.ArrayList, System.Collections.Queue, System.Collections.Stack, System.Collections.BitArray, ко вторым (называемым также словарями) относятся такие классы, как, например, System.Collections.SortedList, System.Collections.Hashtable.

2. Обсуждение существующих коллекций

Все классы коллекций находятся в пространстве имён System.Collections. В дополнение к этому пространству имён существует ещё одно пространство имён под названием System.Collections.Specialized, которое содержит ещё несколько полезных строго типизированных классов коллекций, не столь широко известных, как вышеперечисленные классы. Мы лишь приведём их перечень с кратким описанием каждого.

System.Collections.Specialized.ListDictionary class – класс ведёт себя очень похоже на Hashtable, но превосходит его по скорости при работе с 10 и менее элементами. При работе с большим количеством элементов его использовать не рекомендуется.

System.Collections.Specialized.HybridDictionary class – состоит из двух встроенных коллекций , ListDictionary и Hashtable. Одновременно используется только один из этих классов. ListDictionary используется, пока коллекция содержит 10 или менее элементов, а затем происходит переключение на использование Hashtable. Если коллекция уже переключилась на Hashtable, переключиться обратно на ListDictionary уже нельзя, даже если количество содержащихся элементов станет 10 и менее. При использовании элементов типа string в качестве ключа класс позволяет применять как чувствительный к регистру поиск, так и нечувствительный. Это определяется установкой булевой переменной в конструкторе.

System.Collections.Specialized.CollectionsUtil class – создаёт коллекции, игнорирующие регистр в строках. Этот класс содержит 2 статических метода: один для создания регистронезависимого Hashtable, а другой - для создания регистронезависимого SortedList.

System.Collections.Specialized.NameValueCollection class – эта коллекция состоит из пар ключ – значение, оба из которых относятся к типу string. Особенностью этой коллекции является то, что она может хранить множество значений при единственном ключе.

System.Collections.Specialized.StringCollection class – это обычный список элементов типа string. В неё можно помещать null элементы и дублированные элементы. Этот список чувствителен к регистру. Доступ к элементам данной коллекции можно получить по целочисленному индексу.

System.Collections.Specialized.StringDictionary class – это Hashtable, содержащий ключи и значения типа string. Прежде чем быть добавленными в коллекцию, ключи полностью преобразуются к

нижнему регистру, что позволяет осуществлять сравнение, не чувствительное к регистру. Ключ не может быть null, в отличие от значения.

3. Классы коллекций ArrayList, Hashtable,Stack, Queue, SortedList

Теперь рассмотрим самые распространённые и широко используемые классы коллекций. И первым из них будет…

System.Collections.ArrayList class – этот класс как никакой другой подобен массиву, но в отличие от массива, может расширяться.

Создать экземпляр этой коллекции очень просто:

```
ArrayList arr = new ArrayList();
```

В данном случае мы создаём объект коллекции, не задавая явно его вместимость. Вместимость **ArrayList** определяется количеством элементов, которые он может содержать. Просмотреть вместимость данной коллекции можно с помощью свойства Capacity:

```
Console.WriteLine("Capacity: {0}", arr.Capacity);
```

Можно задать вместимость коллекции при её создании, явно указав количество элементов в конструкторе:

```
ArrayList arr1 = new ArrayList(5);
Console.WriteLine("Capacity: {0}", arr1.Capacity);
// Capacity: 5
```

При добавлении первых элементов в **ArrayList** его вместимость автоматически устанавливается. Изначально она равна 4 элементам.

```
ArrayList arr = new ArrayList();
arr.Add("word");
Console.WriteLine("Capacity: {0}", arr.Capacity);
arr.Add("letter");
Console.WriteLine("Capacity: {0}", arr.Capacity);
```

```
// Capacity:4
// Capacity:4
```

Можно также явно задать вместимость **ArrayList** уже после его создания:

```
ArrayList arr = new ArrayList();
Console.WriteLine("Capacity: {0}", arr.Capacity);
arr.Capacity = 17;
Console.WriteLine("Capacity: {0}", arr.Capacity);
```

```
// Capacity:0
// Capacity:17
```

Если мы пытаемся поместить в ArrayList больше элементов, чем он может вмещать, его вместимость автоматически удваивается. Внутренне происходит выделение под коллекцию новой области памяти нового (двойного) размера.

```
ArrayList arr = new ArrayList(2);
Console.WriteLine("Capacity: {0}", arr.Capacity);
arr.AddRange(new int[] { 1, 2, 3 });
Console.WriteLine("Capacity: {0}", arr.Capacity);
```

```
// Capacity:2
// Capacity:4
```

Как видим из примеров, добавление элементов в данную коллекцию осуществляется с помощью методов Add() и AddRange() – первый для добавление единичного объекта, а второй – для массива объектов.

Вместимость **ArrayList** также может быть уменьшена до фактического количества элементов, которые он содержит.

```
ArrayList arr = new ArrayList(7);
Console.WriteLine("Capacity: {0}", arr.Capacity);
arr.AddRange(new int[] { 1, 2, 3 });
Console.WriteLine("Capacity: {0}", arr.Capacity);
arr.TrimToSize();
Console.WriteLine("Capacity: {0}", arr.Capacity);
```

```
// Capacity:7
// Capacity:7
// Capacity:3
```

Доступ к элементам этой коллекции может осуществляться по индексу.

```
ArrayList arr = new ArrayList(new int[]{1,2,3,4,5,6,7});
Console.WriteLine(arr[1]);
// 2
```

ArrayList может принимать null как значение и допускает добавление повторяющихся элементов. А свойство Count показывает фактичекое количество элементов коллекции.

```
ArrayList arr = new ArrayList(new string[]{"one", "one", "two",
null});
Console.WriteLine("Count: {0}", arr.Count);
// Count: 4
```

Коллекция **ArrayList** воспринимает свои элементы как объекты типа Object. Поэтому, в принципе, в неё можно поместить объекты разных типов. Только при этом для дальнейшего их применения нужно приводить их к соответствующему типу данных.

```
ArrayList arr = new ArrayList();
arr.Add("one");
arr.Add(10);
arr.Add(true);
foreach (object o in arr)
  Console.WriteLine(o.ToString());
int i = (int)arr[1];
  Console.WriteLine(i);
// one
// 10
// true
// 10
```

Ещё один интересный метод коллекции **ArrayList** – это GetRange(), он позволяет сформировать новую коллекцию со значениями элементов, взятых в указанном диапазоне из исходной коллекции. Первый параметр метода GetRange() указывает индекс начала диапазона, второй – количество элементов для копирования.

```
ArrayList days = new ArrayList(new string[]{"Sunday", "Monday",
"Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"});
```

```
ArrayList only = new ArrayList(days.GetRange(0,3));
foreach(string s in only)
    Console.WriteLine(s);
// Sunday
// Monday
// Tuesday
```

При добавлении в коллекцию **ArrayList** элементов с помощью метода Add() мы добавляет их в конец коллекции. Однако можно добавлять элементы и в заданное место коллекции, для этого применяется метод Insert (), первый параметр – индекс, куда добавлять новый элемент, а второй параметр – значение элемента.

```
ArrayList numbers = new ArrayList(new int[] { 1, 2, 3, 4 });
numbers.Insert(2, 22);
foreach (int i in numbers)
     Console.WriteLine(i);
// 1
// 2
// 22
// 3
// 4
```

Аналогичным путём можно удалять элементы коллекции ArrayList, явно указывая их индекс. Для этого применяется метод RemoveAt().

```
ArrayList numbers = new ArrayList(new int[] { 1, 2, 3, 4 });
numbers.RemoveAt(2);
foreach (int i in numbers)
    Console.WriteLine(i);
// 1
// 2
// 4
```

Метод Sort() сортирует элементы коллекции по принципу, зависящему от типа элементов:

```
ArrayList numbers = new ArrayList(new int[] { 5, 7, 4 });
numbers.Sort();
foreach (int i in numbers)
    Console.WriteLine(i);
// 4
// 5
```

```
// 7
```

System.Collections.Stack class – следующий класс коллекций, который мы рассмотрим. Это тип коллекции, идеально подходящий для временного размещения объектов, которые после использования будут удалены. Эта коллекция работает по принципу LIFO (last-in-first-out). То есть последовательность извлечения объектов противоположна последовательности их помещения в коллекцию.

Создать эту коллекцию так же просто, как предыдущую:

```
Stack s = new Stack();
```

Или как вариант можно сразу задать вместимость:

```
Stack s = new Stack(5);
```

Элементы добавляются в коллекцию в помощью метода Push():

```
Stack s = new Stack();
            s.Push("one");
            s.Push("two");
            s.Push("three");

            foreach (string str in s)
                Console.WriteLine(str);
// three
// two
// one
```

... а извлекаются при помощи метода Pop():

```
s.Pop();
foreach (string str in s)
  Console.WriteLine(str);
// two
// one
```

Обратим внимание, в какой последовательности распечатываются элементы и что удаляется в первую очередь. Это обусловлено принципом LIFO.

Если нам нужно получить значение элемента, который был помещён в коллекцию последним, необходимо применить метод Peek(), который не удаляет элемент из коллекции, в отличие от Pop():

```
  string var =  (string)s.Peek();
  Console.WriteLine(var);
  foreach (string str in s)
   Console.WriteLine(str);
// three
// three
// two
// one
```

Обратиться с помощью индекса к элементу данной коллекции нельзя.

Для проверки, содержит ли коллекция заданное значение применяется её свойство Contains, которая возвращает true в случае положительного результата и false в случае отрицательного.

```
Stack s = new Stack();
s.Push("one");
s.Push("two");
s.Push("three");
Console.WriteLine(s.Contains("ten"))
Console.WriteLine(s.Contains("two"));
// false
// true
```

Для очистки содержимого коллекции применяется метод Clear():

```
Stack s = new Stack();
s.Push("one");
s.Push("two");
s.Push("three");
Console.WriteLine(s.Count);
s.Clear();
Console.WriteLine(s.Count);
// 3
// 0
```

Для копирования содержимого коллекции в массив применяется метод CopyTo():

```
Stack s = new Stack();
s.Push("one");
s.Push("two");
s.Push("three");
```

```
string[] str = new string[s.Count];
s.CopyTo(str, 0);
foreach (string var in str)
  Console.WriteLine(var);
// three
// two
// one
```

System.Collections.Queue class – главным отличием данной коллекции от Stack заключается в принципе FIFO (first-in-first-out), по которому работает очередь. Это значит, что элементы извлекаются из коллекции в той же последовательности, в которой они были в неё помещены. Индексация к элементам данной коллекции не применяется.

Создание очереди:

```
Queue q = new Queue();
```

Основными методами очереди являются методы Enqueue() и Dequeue() для помещения элементов в коллекцию и извлечения соответственно.

```
Queue q = new Queue();
 for (int i = 1; i < 3; i++)
 {
      q.Enqueue(i);
      Console.WriteLine(q.Dequeue());
 }
Console.WriteLine(q.Contains(2));
// 1
// 2
// false
```

Основными методами Метод Peek() есть и у очереди, но, в отличие от Stack, он возвращает значение элемента, который был первым помещён в коллекцию (при этом не удаляя его).

```
Queue q = new Queue();
for (int i = 1; i < 3; i++)
{
   q.Enqueue(i);
}
```

```
Console.WriteLine(q.Peek());
foreach (int i in q)
   Console.WriteLine(i);
// 1
// 1
// 2
```

System.Collections.SortedList class – представляет собой коллекцию пар объектов ключ-значение, которые сортируются по ключу и доступ к элементам которой может быть получен по ключу и по индексу.

Если при создании списка мы явно не указываем в конструкторе его ёмкость, она по умолчанию равна нулю. Однако при добавлении первого элемента ёмкость устанавливается в 16 элементов. При превышении данной величины ёмкость автоматически удваивается (происходит перераспределение памяти) :

```
SortedList list = new SortedList();
Console.WriteLine(list.Capacity);
list.Add("Me", "Anastasia");
Console.WriteLine(list.Capacity);
// 0
// 16
```

Как видим, пары элементов добавляются в коллекцию при помощи метода Add(), причём задаётся сразу обе величины: и ключ, и значение.

Существует несколько способов, как можно просмотреть значения, находящиеся в списке:

```
SortedList list = new SortedList();
list.Add(2, 20);
list.Add(1, 100);
list.Add(3, 3);
foreach (int i in list.GetKeyList())
   Console.WriteLine(i);
foreach (int i in list.GetValueList())
   Console.WriteLine(i);
for (int i = 0; i < list.Count; i++)
 Console.WriteLine("Key- " + list.GetKey(i) + " Value - " +
```

```
   list.GetByIndex(i));
//1
//2
//3
//100
//20
//3
//Key - 1 Value - 100
//Key - 2 Value - 20
//Key - 3 Value - 3
```

Обратим внимание, что элементы, помещённые в SortedList автоматически будут отсортированы по ключу.

Для удаления элемента из списка по заданному значению ключа применяется метод Remove() , а для удаления по заданному индексу – метод RemoveAt().

```
list.RemoveAt(1);
foreach (int i in list.GetKeyList())
Console.WriteLine(i);
list.Remove(3);
foreach (int i in list.GetKeyList())
 Console.WriteLine(i);
//1
//3
//1
```

Пары ключ-значение не обязательно должны быть одного типа, т.е., например, ключи могут быть типа int, а значения – string или наоборот. Но все пары должны быть однородными, т.е. все ключи одного и того же типа и все значения одного и того же типа. Это связано с тем, что коллекция не может отсортировать объекты (ключи) разных типов.

```
SortedList list = new SortedList();
list.Add(1, "one");
list.Add(2, "two");
foreach (int i in list.GetKeyList())
    Console.WriteLine(i);
   SortedList list1 = new SortedList();
   list1.Add("one", 1);
   list1.Add("two", 2);
   foreach (string i in list1.GetKeyList())
```

```
    Console.WriteLine(i);
//1
//2
//one
//two
```

System.Collections.Hashtable class – представляют собой наборы данных, содержащих пары ключ-значение, причём оба они относятся к типу Object, а это значит, что тип и ключа, и его парного значения может быть совершенно любой. Эта коллекция относится к так называемой группе словарей (по аналогии с реальным словарём), где по ключу мы можем получить доступ к значению.

Создать коллекцию и просмотреть её данные можно так:

```
Hashtable hash = new Hashtable();
hash.Add(1, "one");
hash.Add(2, "two");
hash.Add("three", 3);
foreach (object o in hash.Keys)
    Console.Write(o.ToString() + " ");
Console.WriteLine();
foreach (object o in hash.Values)
      Console.Write(o.ToString() + " ");
//three 2 1
//3 two one
```

При создании словаря можно явно указать его вместимость, причём для этого рекомендуется использовать простые числа – это связано с алгоритмом работы словарей, так они работают эффективнее:

```
Hashtable hash = new Hashtable(17);
```

Заметим, что ключи и значения могут быть разных типов не только в пределах пары, но и в пределах коллекции, т.е. одна пара может быть, например, int – string, другая – string int, третья – string-string и так далее.

Словари позволяют легко добавлять (с помощью метода Add()) и удалять (с помощью метода Remove()) данные, но нужно знать, что добавление данных осуществляется у них не так, как в других коллекциях, пары

элементов могут добавляться не только к конец коллекции, но и в середину. Вы спросите, как вычисляется позиция по ключу? Вам это знать совсем не обязательно, главное, что, зная ключ, вы всегда можете получить его значение, а словарь сам его для вас найдёт. Конечно, если описать этот процесс очень схематично, то можно сказать, что словарь использует для вычисления расположения значения специальный алгоритм, состоящим из двух стадий, одна из которых должна быть представлена ключём. Поэтому если вы хотите использовать в качестве ключа пользовательский тип, а не предоставленный Microsoft (у этих типов данный алгоритм уже реализован), необходимо будет самостоятельно решать пролему написания этого алгоритма. Тогда ваш класс должен переопределить метод GetHashCode() и Equals() типа Object.

```
hash.Add("four", 4);
hash.Remove(1);
foreach (object o in hash.Keys)
Console.Write(o.ToString() + " ");
//three four 2
```

Проверить, содержит ли коллекция заданное значение, можно указав его ключ в качестве метода Contains():

```
Console.WriteLine(hash.Contains("three"));
//true
```

4. Интерфейсы коллекций IList, IEnumerator, IEnumerable, ICollection, IDictionary, IComparer, IDictionaryEnumerator

Теперь пройдёмся по интерфейсам, представленым пространством имён System.Collections. Как правило, они реализованы для обеспечения доступа к содержимому коллекции.

System.Collections.ICollection interface - определяет размер, перечислители и синхронизированные методы для необобщённых коллекций. Он является базовым интерфейсом для классов пространства имён System.Collections.

```
public interface ICollection: IEnumerable
{
   public void CopyTo(Array array, int index);
   public int Count { get; }
   public bool IsSynchronized { get; }
   public object SyncRoot { get; }
}
```

System.Collections.IDictionary interface – это базовый интерфейс необобщённый коллекций, представляющих наборы пар ключ-значение. Каждая пара значений должна иметь уникальный ключ. Значение может быть null и ему не обязательно быть уникальным. Реализация интерфейса делится на 3 категории: read-only, fixed-sized, variable-size. Объект IDictionary read-only не может быть модифицирован. Объект IDictionary fixed-sized не разрешает удаление и добавление элементов, но разрешает модифицировать существующие элементы. Объект IDictionary variable-size разрешает добавление, удаление и модификацию элементов.

```
public interface IDictionary : ICollection, IEnumerable
{
   public void Add(object key, object value);
   public void Clear();
   public bool Contains(object key);
   public IDictionaryEnumerator GetEnumerator();
   public bool IsFixedSize { get;}
   public bool IsReadOnly{ get;}
```

```
    public ICollection Keys{ get;}
    public void Remove(object key);
    public ICollection Values{ get;}
    public object this[object key]{ get; set;}
}
```

System.Collections.IDictionaryEnumerator interface – перебирает элементы необобщённой коллекции. Оператор foreach скрывает сложность перечислителей. Поэтому foreach рекомендуется использовать вместо перечислителей. Перечислитель может быть использован для чтения данных из коллекции, но не изменять их при этом.

```
public interface IDictionaryEnumerator: IEnumerator
{
    public DictionaryEntry Entry{get;}
    public object Key{get;}
    public object Value{get;}
}
```

System.Collections.IList interface – представляет необобщённую коллекцию объектов, доступ к которым можно получить по индексу. Реализация интерфейса делится на 3 категории: read-only, fixed-sized, variable-size. Объект IList read-only не может быть модифицирован. Объект IList fixed-sized не разрешает удаление и добавление элементов, но разрешает модифицировать существующие элементы. Объект IList variable-size разрешает добавление, удаление и модификацию элементов.

```
public interface IList: ICollection, IEnumerable
{
    public int Add(object value);
    public void Clear();
    public bool Contains(object value);
    public int IndexOf(object value);
    public void Insert(int index, object value);
    public bool IsFixedSize{get;}
```

```
        public bool IsReadOnly{get;}
        public void Remove(object value);
        public void RemoveAt(int index);
        public object this[int index]{get; set;}
    }
```

System.Collections.IEnumerable interface – поддерживает перечислитель, который производит итерации по необобщённой коллекции. Должен быть реализован для поддержки оператора foreach.

```
public  interface IEnumerable
{
    public IEnumerator GetEnumerator();
}
```

System.Collections.IComparer interface – предоставляет метод для сравнения двух объектов.

```
public interface IComparer
{
     public int Compare(object x, object y);
}
```

System.Collections.IEnumerator interface – поддерживает простую итерацию по необобщённой коллекции. Является базовым интерфейсом для необобщённых перечислителей.

```
public interface IEnumerator
{
      public object Current{get;}
      public bool MoveNext();
      public void Reset();
}
```

5. Примеры использования классов коллекций для хранения стандартных и пользовательских типов

Для поиска элементов в коллекции ArrayList мы используем метод BinarySearch(), возвращающий индекс элемента, если он там есть, или отрицательное значение, если его нет в коллекции. Но для того чтобы он работал корректно, колекция должна быть сначала отсортирована. Но можно просто забыть отсортировать коллекцию до осуществления поиска. Было бы просто замечательно, если бы коллекция ArrayList всегда была отсортирована. Мы создадим собственный класс, производный от класса ArrayList, и определим в нём свой метод для добавления элементов в колекцию так, чтобы они сразу были отсортированы, а также метод для изменения значения элемента под заданным индексом, который также гарантирует расположение элементов в правильном порядке.

```
using System;
using System.Collections;

public class SortedArrayList : ArrayList
{
    public void AddSorted(object item)
    {
        int position = this.BinarySearch(item);

        if (position < 0)
        {
            position = ~position;
        }

        this.Insert(position, item);
    }

    public void ModifySorted(object item, int index)
    {
        this.RemoveAt(index);

        int position = this.BinarySearch(item);

        if (position < 0)
        {
            position = ~position;
        }

        this.Insert(position, item);
    }
}
```

```
class CTest
{
  static void Main()
  {
        SortedArrayList sortedAL =
       new SortedArrayList();
      sortedAL.AddSorted(200);
      sortedAL.AddSorted(-7);
      sortedAL.AddSorted(0);
      sortedAL.AddSorted(-20);
      sortedAL.AddSorted(56);
      sortedAL.AddSorted(200);

      // Display it
      foreach (int i in sortedAL)
      {
            Console.Write(i+ " ");
      }
     Console.WriteLine();
      // Now modify a value at a particular index
      sortedAL.ModifySorted(3, 5);
      sortedAL.ModifySorted(-1, 2);
      sortedAL.ModifySorted(2, 4);
      sortedAL.ModifySorted(7, 3);
      // Display it
      Console.WriteLine();

      foreach (int i in sortedAL)
      {
            Console.Write(i+ " ");
      }
  }
}
```

//-20 -7 0 56 200 200
//-20 -7 -1 3 7 200

Рассмотрим следующий пример. Предположим, мы хотим отсортировать ключи или значение, хранящиеся к коллекции Hashtable и представить пользователю всю коллекцию отсортированной по возрастанию или убыванию. Для этого мы воспользуемся свойствами Keys и Values коллекции.

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

```
public class mySortedHashtable
{
    private Hashtable hash = new Hashtable();
    // Return an ArrayList of Hashtable keys
    public ArrayList GetKeys(Hashtable table)
    {
        return (new ArrayList(table.Keys));
    }

    // Return an ArrayList of Hashtable values
    public ArrayList GetValues(Hashtable table)
    {
        return (new ArrayList(table.Values));
    }

    public void FillIn(object x, object y)
    {
        hash.Add(x, y);
    }

    public void TestSortKeyValues()
    {
    // Get all the keys in the hashtable and sort them
        ArrayList keys = GetKeys(hash);
        keys.Sort();

        // Display sorted key list
        foreach (object obj in keys)
            Console.WriteLine("Key: " + obj +
            "    Value: " + hash[obj]);

        // Reverse the sorted key list
        Console.WriteLine();
        keys.Reverse();

        // Display reversed key list
      foreach (object obj in keys)
        Console.WriteLine("Key: " + obj +
"    Value: " + hash[obj]);

    // Get all the values in the hashtable and sort them
        Console.WriteLine();
        Console.WriteLine();

        ArrayList values = GetValues(hash);
        values.Sort();

        foreach (object obj in values)
            Console.WriteLine("Value: " + obj);

        // Reverse the sorted value list
        Console.WriteLine();
        values.Reverse();
```

```
                foreach (object obj in values)
                    Console.WriteLine("Value: " + obj);
            }
        }
        public class Tester
        {
            public static void Main()
            {
                mySortedHashtable coll = new mySortedHashtable();
                coll.FillIn(2, "two");
                coll.FillIn(1, "one");
                coll.FillIn(5, "five");
                coll.FillIn(3, "three");
                coll.TestSortKeyValues();
            }
        }
```

```
//Key: 1  Value: one
//Key: 2  Value: two
//Key: 3  Value: three
//Key: 5  Value: five

//Key: 5  Value: five
//Key: 3  Value: three
//Key: 2  Value: two
//Key: 1  Value: one

//Value: five
//Value: one
//Value: three
//Value: two

//Value: two
//Value: three
//Value: one
//Value: five
```

Для следующего примера требуется наличие изображений с расширениями ***.bmp, *.gif, *.jpg, *.png в любой папке компьютера (желательно корневой для простоты). Программа просит ввести путь к папке, содержащей изображения и выводит на экран имя изображения и его расширение. В программе заданы расширения, которые необходимо считать. Коллекция ArrayList содержит объекты, содержащие информацию о файлах.**

```
using System;
using System.Collections;
using System.IO;

namespace ArrayListExz
{
    class Program
    {
        static void Main(string[] args)
        {
            String pathName;
            Console.Write("Введите путь к папке с
изображениями: ");
            pathName = Console.ReadLine();
            DirectoryInfo di = new DirectoryInfo(pathName);
            string[] fileTypes = { "*.bmp", "*.gif", "*.jpg",
"*.png" };
            ArrayList images = new ArrayList();

            foreach (string ext in fileTypes)
            {
                FileInfo[] files = di.GetFiles(ext);
                if (files.Length > 0)
                {
                    images.AddRange(files);
                }
            }

            Console.WriteLine("В Данной директории имеются
следующие Изображения:");

            for (int i = 0; i < images.Count; i++)
            {
                FileInfo file = (FileInfo)images[i];

                Console.WriteLine("Имя файла: {0}, его
расширение: {1}", file.Name, file.Extension);
            }

            Console.ReadLine();
        }
    }
}
```

А теперь рассмотрим вариант предыдущей задачи. Отличие заключается в том, что теперь пользователь может сам задать расширение файла с изображением.

```
using System;
using System.Collections;
using System.IO;

namespace ArrayListExz
{
    class Program
    {
        static void Main(string[] args)
        {
            String pathName;
            String extension;

            Console.Write("Введите путь к папке с
изображениями: ");
            pathName = Console.ReadLine();
            Console.Write("Введите расширение для считываемых
изображений в формате *.extension: ");
            extension = Console.ReadLine();

            DirectoryInfo di = new DirectoryInfo(pathName);

            ArrayList images = new ArrayList();

                FileInfo[] files = di.GetFiles(extension);
                if (files.Length > 0)
                {
                    images.AddRange(files);
                }

            Console.WriteLine("В данной директории имеются
следующие изображения:");

            for (int i = 0; i < images.Count; i++)
            {
                FileInfo file = (FileInfo)images[i];

                Console.WriteLine("Имя файла: {0}" ,
file.Name);
            }

            Console.ReadLine();
        }
    }
}
```

6. Generics

a. Что такое generics?

Многие алгоритмы не зависят от типов данных, с которыми они работают. Например, алгоритмы сортировки и поиска. В C++ для реализации таких алгоритмов использовались шаблонные классы. Например, библиотека STL содержит классы контейнеров, в которых могут храниться данные произвольного типа. При создании класса контейнера указывается тип данных, для которого этот контейнер будет использоваться.

В .Net Framework, начиная с версии 2.0, также появилась возможность использовать в качестве параметров типы данных. Эта возможность реализуется с помощью generics.

Generics позволяют также создавать обобщенные:

- ✓ классы
- ✓ структуры
- ✓ интерфейсы
- ✓ делегаты
- ✓ методы

Параметры типов используются для указания типов:

- ✓ переменных класса
- ✓ параметров функция
- ✓ возвращаемых значений функций
- ✓ локальных переменных

b. Необходимость использования generics.

Для понимания необходимости использования generics имеет смысл рассмотреть сложности, которые возникают при использовании необобщенных классов. Наиболее наглядно эти сложности проявляются при работе с коллекциями.

В коллекции ArrayList хранятся данные типа object. Это сделано для того чтобы в эту коллекцию можно было поместить переменные любого типа. Такая гибкость в некоторых случаях удобна, однако чаще всего в коллекции хранятся переменные одного и того же типа. Из-за того что данные имеют тип object, при извлечении данных из коллекции надо указываться явное приведение типа. Можно легко допустить ошибку приведения при извлечении данных из коллекции, т.е. поместить в коллекцию переменную одного типа, а при извлечении выполнить приведение к другому типу. Такая ошибка не будет выявлена на этапе компиляции, а проявится только при работе программы.

Пример 1. Ошибка преобразования типа при извлечении данных из коллекции.

```
using System;
using System.Collections;

namespace GenericsBenefits
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList al = new ArrayList();
            //помещаем в коллекцию переменную типа int
            al.Add(10);
            try
            {
                //при извлечении выполняем приведение типа
                //из-за ошибочного указания типа возникает исключение
                short a = (short)al[0];
            }
            catch (InvalidCastException e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

```
Заданное приведение является недопустимым.
```

Другим существенным недостатком при использовании необобщенных коллекций является снижение производительности из-за выполнения упаковки и распаковки при хранении в коллекции

переменных структурных типов. Так, метод коллекции ArrayList Add(object) принимает переменную типа object. Т.е. при передаче в этот метод переменной структурного типа (например, int) будет выполняться упаковка. Для обращения к элементам коллекции используется индексатор, который также возвращает результат типа object. Т.е. при присвоении этого результата структурному типу будет выполняться распаковка.

Методы generic коллекций принимают и возвращают параметры обобщенного типа. Поэтому при помещении данных в коллекцию и обращении к этим данным упаковка и распаковка не происходит.

Пример 2. Сравнения временных затрат при хранении структурных типов в необобщенной и обобщенной коллекции. Этот пример был предложен Jeffrey Richter в книге «The CLR via C#».

Для выполнения профилирования используется вспомогательный класс OperationTimer. Его назначение – точное измерение времени выполнения участка кода, а также подсчет количества сборок мусора. Если реализация этого класса в настоящий момент Вам не совсем понятна, Вы сможете вернуться к нему после изучения темы «Сборка мусора». Этот класс является удачным паттерном и его можно использовать для профилирования в собственных проектах.

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

public static class Program
{
    public static void Main()
    {
        ValueTypePerfTest();
    }

    /// <summary>
    /// метод для тестирования производительности
    /// обобщенного и необобщенного списка
    /// </summary>
   private static void ValueTypePerfTest()
    {
        const Int32 count = 10000000;
```

```
        //объект OperationTimer
        //создается перед началом использования коллекции
        //после завершения ее использования
        //выводит время, затраченное на работу с коллекцией
        using (new OperationTimer("List"))
        {
            //использование обобщенного списка
            List<int> l = new List<int>(count);
            for (Int32 n = 0; n < count; n++)
            {
                l.Add(n);
                Int32 x = l[n];
            }
            l = null;  // для гарантированного выполнения сборки мусора
        }

        using (new OperationTimer("ArrayList of Int32"))
        {
            //использование необобщенного списка
            ArrayList a = new ArrayList();
            for (Int32 n = 0; n < count; n++)
            {
                a.Add(n); //выполняется упаковка
                Int32 x = (Int32)a[n]; //выполняется распаковка
            }
            a = null;  // для гарантированного выполнения сборки мусора
        }
    }
}

/// <summary>
/// Вспомогательный класс для профилирования участка кода
/// выполняет измерения времени выполнения
/// и подсчет количества сборок мусора
/// </summary>
internal sealed class OperationTimer : IDisposable
{
    private Int64 m_startTime;
    private String m_text;
    private Int32 m_collectionCount;

    public OperationTimer(String text)
    {

        PrepareForOperation();

        m_text = text;

        //Сохраняется количество сборок мусора,
        //выполненных на текущий момент
        m_collectionCount = GC.CollectionCount(0);

        //Сохранеяется начальное время
        m_startTime = Stopwatch.GetTimestamp();
    }

    /// <summary>
    /// Вызывается при разрушении объекта
    /// Выводит:
```

```
    /// значение времени, прошедшего с момента создания объекта
    /// до момента его удаления
    /// количество выполненных сборок мусора, выполненных за это время
    /// </summary>
    public void Dispose()
    {
        Console.WriteLine("{0,6:0.00} seconds (GCs={1,3}) {2}",
           (Stopwatch.GetTimestamp() - m_startTime) /
              (Double)Stopwatch.Frequency,
           GC.CollectionCount(0) - m_collectionCount, m_text);
    }
    /// <summary>
    ///Метод удаляются все неиспользуемые объекты
    ///Это надо для "чистоты эксперимента",
    ///т.е. чтобы сборка мусора происходила только для объектов,
    ///которые будут создаваться в профилируемом блоке кода
    ///</summary>
    private static void PrepareForOperation()
    {
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();
    }
}
```

```
 0,23 seconds (GCs=  0) List
 2,96 seconds (GCs= 35) ArrayList of Int32
```

Из полученных результатов видно, что использование необобщенной коллекции ArrayList приводит к значительным временным затратам, а также к интенсивному выполнению сборок мусора.

Следует отметить, что если в необобщенных коллекциях хранить объекты ссылочных типов данных, то снижение производительности происходить не будет, т.к. в этом случае не будет выполняться упаковка и распаковка.

Приведенные примеры показывают, что использование generics дает следующие преимущества:

- ✓ безопасность типов – в необобщенные коллекции можно было помещать любые объекта, в generic коллекцию можно поместить только объекты определенного типа (указанного при раскрытии шаблона);
- ✓ более простой и понятный код, т.к. ненужно выполнять приведения тип от object к конкретным типам;

✓ повышение производительности – при использовании generics структурные типы передаются по значению, упаковка и распаковка не происходит.

c. Создание generic классов.

При создании generic класса параметр типа указывается в угловых скобках после имени класса, затем этот тип используется также как обычные типы. Обобщенных параметров типа может быть несколько.

При использовании generic класса вместо параметра типа подставляют реальный тип данных. Создание конкретного экземпляра generic класса называется инстанцированием или специализацией.

Для задания значения по умолчанию переменным обобщенного типа используется выражение default(T). При этом значениям ссылочного типа присваивается null, а структурного 0.

В отличие от шаблонных классов C++ в generics с переменными обобщенного типа нельзя выполнять арифметические действия и операции сравнения. Это связано с тем, что при инстанцировании вместо параметра типа может быть использован тип данных, который не поддерживает эти операции.

Пример 3. Создание generic класса точки.

```
using System;

namespace GeneicClass
{
    /// <summary>
    /// Обобщенный класс точки
    /// </summary>
    /// <typeparam name="T">
    /// координаты точки могут быть любого типа
    /// </typeparam>
    public class Point2D<T>
    {
        //параметр типа используется для задания типа переменных класса
        T x;
        T y;
        //параметр типа используется для задания типа свойства
        public T X
```

```
        {
            get { return x; }
            set { x = value; }
        }
        public T Y
        {
            get { return y; }
            set { y = value; }
        }
        //параметр типа используется для задания типов параметров метода
        public Point2D(T x, T y)
        {
            this.x = x; this.y = y;
        }
        public Point2D()
        {
            this.x = default(T); this.y = default(T);
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            //тестирование обобщенного класса - точки в 2D
            Point2D<int> p1 = new Point2D<int>(10, 20);
            Console.WriteLine("x = {0} y = {1}", p1.X, p1.Y);
            Point2D<double> p2 = new Point2D<double>(10.5, 20.5);
            Console.WriteLine("x = {0} y = {1}", p2.X, p2.Y);
            Console.WriteLine(typeof(Point2D<int>).ToString());
            Console.WriteLine(typeof(Point2D<double>).ToString());
        }
    }
}
```

```
x = 10 y = 20
x = 10,5 y = 20,5
GeneicClass.Point2D`1[System.Int32]
GeneicClass.Point2D`1[System.Double]
```

Как видно из результатов выполнения программы, из generic класса создаются классы для конкретных типов данных. Эти классы называются constructed type (сконструированный тип). Для объявления Point2D<int> создается класс с именем Point2D`1[System.Int32], для Point2D<double> - класс Point2D`1[System.Double]. Имена этих классов образуются по следующему правилу: Point2D - имя generic класса, 1 – количество параметров типа, [имя тип] – тип, который использовался при специализации шаблона. Эти классы создается из generic класса Point2D<T> во время выполнения программы, когда JIT впервые

выполняет компиляцию метода, использующего переменную такого классы.

Generic классы могут использоваться в качестве базовых классов. Следовательно, generic классы могут иметь виртуальные и абстрактные методы.

Правила наследования от generic классов:

✓ если от generic класса наследуется необобщенный, класс – наследник должен конкретизировать параметр типа

✓ при реализации generic виртуальных методов производный необобщенный класс должен конкретизировать параметр типа

✓ если от generic класса наследуется другой generic класс, в нем необходимо учитывать ограничения типа, указанные в базовом классе.

d. Сравнительный анализ generic классов и классов коллекций.

С появлением концепции generics в FCL практически для всех классов необобщенных коллекций появились соответствующие обобщенные. Эти классы, а также обобщенные интерфейсы находятся в пространстве имен System.Collections.Generic.

В табл. 1 и 2. представлены необобщенные и соответствующие им обобщенные интерфейсы и классы коллекций.

Табл. 1.

Интерфейсы

Generic интерфейсы	Необобщенные интерфейсы
ILits<T>	ILits
IDictionary<T>	IDictionary
ICollection<T>	ICollection
IEnumerator<T>	IEnumerator
IComparer<T>	IComparer
IComparable<T>	IComparable

Табл. 2

Классы коллекций

Generic коллекции	Необобщенные коллекции
Collection<T>	CollectionBase
List<T>	ArrayList
Dictionary<TKey, TValue>	Hashtable
SortedList<TKey, TValue>	SortedList
Stack<T>	Stack
Queue<T>	Queue
LinkedList<T>	Нет

Отличие generic коллекцией от необобщенных коллекций состоит в том, что вместо типа данных object в generic коллекциях используется обобщенный параметр типа. При специализации generic коллекции указывается, какой тип данных будет в этой коллекции содержаться. При первом вызове метода класса коллекции JIT компилирует реализацию этого метода для конкретного параметра типа. В результате те методы и свойства, которые в необобщенной коллекции использовали тип данных object, в generic коллекции используют конкретный тип данных.

Пример 4. Использование generic коллекции List<T>

```
      //Пример использования обобщенного списка
       static void ListExample()
       {
        //Коллекция для хранения целых чисел
        List<int> li = new List<int>();
        Random rnd = new Random();
        for (int i = 0; i < 10; i++) li.Add(rnd.Next(100));
        Console.WriteLine("Int collection: ");
        //при обращении к элементам коллекции возвращается резульат тип
int
        foreach (int i in li) Console.Write("{0} ", i);
        Console.WriteLine();

        Console.WriteLine("String collection: ");
        //Коллекция для хранения строк
         List<string> ls = new List<string>();
         ls.Add("Hello");
         ls.Add("from");
         ls.Add("generics");
       //при обращении к элементам коллекции возвращается резульат тип
string
        foreach (string s in ls) Console.Write("{0} ", s);
        Console.WriteLine();
```

```
    }
```

```
Int collection:
1 57 26 70 81 93 29 40 43 62
String collection:
Hello from generics
```

Если коллекция является словарем, ее класс имеет 2 параметра типа: тип ключа и тип значения.

Пример 5. Использование generic коллекции Dictionary<TKey, TValue>

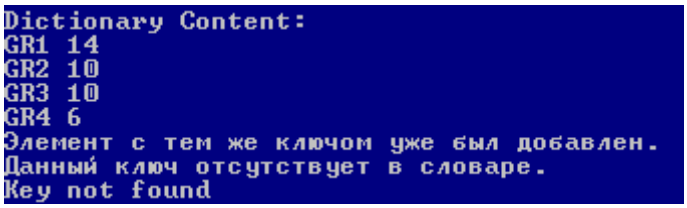
```
// Пример использования обобщенного словаря
 static void DictionaryExample()
 {
     //Словарь для хранения пар:
     //название группы - количество студентов
     Dictionary<string, int> gr = new Dictionary<string, int>();
     //добавление значений в список
     gr["GR1"] = 12;
     gr.Add("GR2", 10);
     gr.Add("GR3", 10);
     gr.Add("GR4", 6);
     //изменение значения
     gr["GR1"] = 14;

     //вывод всех элементов словаря
     Console.WriteLine("Dictionary Content: ");
     foreach (KeyValuePair<string, int> p in gr)
        Console.WriteLine("{0} {1}", p.Key, p.Value);

     //удаление по значению ключа
     gr.Remove("GR4");

     //попытка добавления существующего ключа
     try
     {
         gr.Add("GR1", 15);
     }
     catch (ArgumentException e)
     {
         Console.WriteLine(e.Message);
     }

     //попытка обращения к несуществующему ключу
     try
     {
         Console.WriteLine(gr["GR5"]);
     }
     catch (KeyNotFoundException e)
     {
         Console.WriteLine(e.Message);
     }
     //проверка существования ключа и получение зачнения
     int val;
     if (gr.TryGetValue("GR5", out val)) Console.WriteLine(val);
     else Console.WriteLine("Key not found");
 }
```



Как было показано ранее (п. 6.b) generic коллекции обеспечивают безопасность типов и увеличение производительности при работе со структурными типами данных. Поэтому Microsoft рекомендует везде, где это возможно, использовать обобщенные коллекции вместо необобщенных. Использование необобщенной коллекции может быть оправданным только если действительно существует необходимотсь хранить в одной коллекции объекты разных типов.

e. Вложенные типы внутрь generic класса.

Внутри generic класса можно объявлять вложенные классы. Эти классы также будут являться обобщенными, даже если они не имеют собственных параметров типа. Во вложенных классах можно использовать параметр типа, указанный во внешнем классе. Во вложенном классе можно также объявлять собственный список параметров типа. В этом случае вложенный класс будет параметризирован двумя наборами типов: собственным и внешнего класса.

Пример 6. Создание класса, вложенного в generic класс.

```
using System;

namespace InnerClass
{
    class Program
    {
        class A<T>
        {
            public class Inner
            {
            }
        }
```

```
        class B<T>
        {
           //вложенный класс имеет собственный список параметров типа
            public class Inner<U>
            {
            }
        }
  static void Main(string[] args)
    {
      //для использования вложенного класса
      //необходимо указать реальный тип вместо параметра типа внешнего
класса
        A<int>.Inner a = new A<int>.Inner();
        Console.WriteLine(a);
        A<double>.Inner a1 = new A<double>.Inner();
        Console.WriteLine(a1);

    //для использования вложенного класса
    //необходимо указать реальный тип вместо параметра типа вложенного
класса
     B<int>.Inner<string> b = new B<int>.Inner<string>();
     Console.WriteLine(b);
   }
 }
}
```

```
InnerClass.Program+A`1+Inner[System.Int32]
InnerClass.Program+A`1+Inner[System.Double]
InnerClass.Program+B`1+Inner`1[System.Int32,System.String]
```

f. Использование ограничений.

Для параметра типа можно указать ограничения, указывающие каким требованиям должен удовлетворять тип данных, используемый вместо этого параметра.

Список возможных ограничений:

Ограничение обобщения	Описание
where T: struct	Параметр типа должен наследоваться от system.ValueType, т.е. быть структурным типом
Where T: class	Параметр типа не должен наследоваться от system.ValueType, т.е. быть ссылочным типом
where T: new()	Класс должен иметь конструктор по умолчанию (указывается последним)

where T: BaseClass	Параметр типа должен быть производным классом от указанного базового класса
where T: Interface	Параметр типа должен реализовать указанный интерфейс

Синтаксис задания ограничения:

```
class ИмяКласс<T> where T: ограничения
```

Для одного параметра можно указать несколько ограничений.

За счет использования ограничений правильность и типобезопасность кода может быть проверена на этапе компиляции. При попытке скомпилировать специализацию для конкретного типа компилятор будет проверять, выполняются ли для этого типа указанные ограничения.

Например:

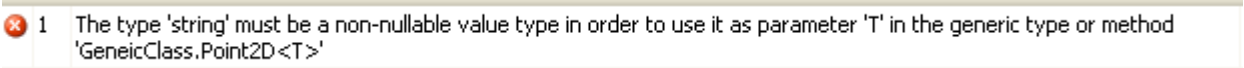
```
class MyGenericClass<T> where T: class, IComparable, new()
```

Эти ограничения требуют, чтобы тип, который используется вместо параметра T, являлся ссылочным, имел конструктор по умолчанию и реализовал интерфейс IComparable.

Пример 7. Задание ограничения параметра типа для класса точки.

```
Public class Point2D<T>  where T: struct
{
  //та же реализация, что и в предыдущем примере
}
static void Main(string[] args)
   {
     Point2D<string> p3 = new Point2D<string>("0", "0");
   }
```

При компиляции возникает ошибка:



Эта ошибка возникает из-за того, что тип String является ссылочным, а для параметра типа T разрешено использование только структурных типов.

g. Создание generic интерфейсов

При создании интерфейсов, как и при создании классов, можно использовать обобщенные параметры типа. Особенно удобно использовать обобщенные интерфейсы при работе со структурными типами, т.к. работа со структурными типами через необобщенный интерфейс привела бы к упаковке и распаковке, а также к потере безопасности типов во время компиляции.

Некоторые необобщенные интерфейсы имеют соответствующие им обобщенные. Например, обобщенный интерфейс IComparable имеет метод int CompareTo(object), который в качестве параметра принимает переменную типа object. Ему соответствует generic интерфейс IComparable <T>, в котором метод CompareTo(T) принимает переменную обобщенного типа.

При реализации generic интерфейса в необобщенном классе необходимо конкретизировать аргументы типа, в generic классе может реализовать интерфейс, используя обобщенные параметры типа.

Обобщенные интерфейсы удобны тем, что их можно указать в качестве ограничений параметра типа при создании generic класса, и при реализации этого класса использовать методы, объявленные в интерфейсе.

Пример 8. Необходимо создать generic класс, в котором будет содержаться коллекция данных обобщенного типа, и в этом классе реализовать метод, который должен возвращать сумму элементов коллекции. Сумма должна иметь тот же тип, что и тип данных в коллекции. Для того чтобы элементы коллекции можно было суммировать надо создать интерфейс, содержащий метод вычисления суммы, и указать этот интерфейс в качестве ограничения параметра типа.

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

namespace GenericInterface
```

```
{
    /// <summary>
    /// Обобщенный интерфейс с методам вычисления суммы
    /// </summary>
    /// <typeparam name="T"></typeparam>
    interface ICalc<T>
    {
        T Sum(T b);
    }
    class Program
    {

        /// <summary>
        /// Необобщенный класс, реализующий интерфейс ICalc
        /// </summary>
        class calcInt : ICalc<calcInt>
        //при наследовании указывается реальный тип данных
        {
            int a = 0;
            public calcInt(int a)
            {
                this.a = a;
            }
            #region ICalc<int> Members
           //при реализации методов вместо обобщенного типа
           //используется тип calcInt
            public calcInt Sum(calcInt b)
            {
                return new calcInt(this.a + b.a);
            }
            #endregion
            public override string ToString()
            {
                return a.ToString();
            }
        }

        /// <summary>
        /// Обобщенный класс, который содержит в себе
        /// коллекцию данных обобщеного типа
        /// и имеет метод вычисления суммы
        /// Для вычисления суммы задается оганичение:
        /// параметр типа должен реализовать интерфейс ICalc<T>
        /// </summary>
        /// <typeparam name="T"></typeparam>
        class MyArray<T> where T : ICalc<T>
        {
            //коллекция данных обобщенного типа
            List<T> li = new List<T>();
            //метод добавления данных в коллекцию
            public void Add(T t)
            {
              li.Add(t);
            }
            //метод вычисления суммы
            public T Sum()
            {
                if (li.Count == 0) return default(T);
                T res = li[0];
                //для суммирования используется метод интерфейса ICalc<T>
```

```
                for (int i = 1; i < li.Count; i++) res = res.Sum(li[i]);
                return res;
            }
        }

        static void Main(string[] args)
        {
            MyArray<calcInt> a = new MyArray<calcInt>();
            a.Add(new calcInt(10));
            a.Add(new calcInt(20));
            Console.WriteLine(a.Sum());
        }
    }
}
```

h. Создание generic делегатов

Поддержка обобщенных делегатов позволяет передавать методам обратного вызова любые типы объектов, обеспечивая при этом безопасность вызовов.

В FCL есть много обобщенных типов делегатов.

Например:

public delegate void Action<T>(T obj) – для выполнения некоторых действий над элементами коллекции

public delegate int Comparison<T>(T *x*,T *y*) – для сравнения 2 объектов одинакового типа

public delegate bool Predicate<T>(T *obj*) - представляет метод для проверки, удовлетворяет ли объект заданным критериям

Пример 9. Использование generic делегатов:

1. вывод массива на печать с использование делегата Action
2. поиск отрицательных элементов массива с использованием делегата Predicate

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

namespace DelegateExample
{
    class Program
    {
        static void print(int arg)
        {
            Console.WriteLine(arg);
        }
        static bool neg(int arg)
```

```
        {
            return (arg < 0);
        }

        static void Main(string[] args)
        {
            List<int> li = new List<int>(10);
            Random rnd = new Random();
            //заполнение массива случайными числами
            for (int i = 0; i < 10; i++)
                li.Add(rnd.Next(20) - 10);
            //печать массива с использованием
            //предиката Action<T>
            Console.WriteLine("Исходный массив");
            li.ForEach(new Action<int>(print));
            //поиск отрицательных элементов с использованием
            //предиката Predicate<T>
            List<int> li_neg = li.FindAll(new Predicate<int>(neg));

            Console.WriteLine("Отрицательные элементы массива");
            li_neg.ForEach(new Action<int>(print));
        }
    }
}
```

i. Создание generic методов

В некоторых случаях удобно иметь метод, параметризированный каким-то типом данных. Такой метод может находиться в необобщенном классе. При создании generic метода параметр типа указывается в угловых скобках после имени метода.

При вызове метода можно после имени указывать реальный тип данных, но можно этого и не делать, т.к. тип данных определяется автоматически по типу параметров, переданных в метод.

Пример 10. Метода нахождения максимального элемента массива. Для поиска максимального элемента необходимо выполнять сравнение элементов массива, поэтому необходимо потребовать, чтобы тип элементов массива реализовал интерфейс IComparable.

```
using System;

namespace GenericMethod
{
    class Program
    {
        /// <summary>
        /// Обобщенный метод поиска максимального элемента в массиве
        /// </summary>
```

```
        /// <typeparam name="T">тип элементов массива</typeparam>
        /// <param name="a">исходный массив</param>
        /// <returns>найденный максимальный элемент</returns>
        static T MaxElem<T>(T [] a) where T:  IComparable<T>
        {
            T m = a[0];
            foreach (T t in a)
            {
                if (t.CompareTo(m) > 0) m = t;
            }
            return m;
        }
        static void Main(string[] args)
        {
            int[] a = new int[] { 2, 6, 8 };

            //реальный тип для параметра типа указывается явно
            Console.WriteLine(MaxElem<int>(a));
            //реальный тип определяется по типу переданного массива
            Console.WriteLine(MaxElem(a));

        }
    }
}
```

Домашнее задание

1. Создать строго типизированную коллекцию, не принимающую значения никаких других типов, кроме исходного (на ваше усмотрение).
2. Разработать собственный класс, имитирующий работу стека.
3. Создать примитивный англо-русский и русско-английский словарь, содержащий пары слов –названий стран на русском и английском языках. Пользователь должен иметь возможность выбирать направление перевода и запрашивать перевод.
4. Создать необобщенный класс точки в 3-х мерном пространстве с целочисленными координатами (Point3D), который наследуется от generic класса Point2D<T>, рассмотренного в уроке. В классе предусмотреть поле для хранения значения координаты z. Реализовать в классе:
   a. конструктор с параметрами, который принимает начальные значения для координат точки
   b. метод ToString()
5. Создать обобщенный класс прямой на плоскости. В классе предусмотреть 2 поля типа обобщенной точки – точки, через которые проходит прямая. Реализовать в классе:
   a. конструктор, который принимает 2 точки
   b. конструктор, которые принимает 4 координаты (x и y координаты для 1-ой и 2-ой точки)
   c. метод ToString()
6. Подсчитать, сколько раз каждое слово встречается в заданном тексте. Результат записать в коллекцию Dictionary<TKey, TValue>