№ 4 Обобщения, коллекции, LINQ

Задание

1. Создать **обобщенный тип (класс) CollectionType<T>**, в котором наследуйте стандартный интерфейс (например IList<T>, ISet, ICollection и т.п.) или вложите обобщённую коллекцию. CollectionType можно делать на стандартных коллекциях (List, Stack, Array и т.д.). Определите в классе конструкторы, методы добавления и удаления элементов, другие необходимые методы если требуется. Определить **индексатор.** Добавьте **обработку исключений** с finally. Наложите **ограничение** на обобщение.
2. Взять, любой созданный тип (класс) из лабораторной №2, наследуйте его от **интерфейсов** IComparable<T>, IEnumerable. При демонстрации задайте данный тип в качестве параметра вашего обобщенного класса CollectionType. Добавьте метод сохранения коллекции в файл.
3. Выполните сортировку и **запросы**, указанные в варианте. При этом используйте язык **запросов LINQ to Object**. Где будет возможно используйте **лямбда-выражения**.
4. В демонстрационной программе создайте обобщённую стандартную **коллекцию** из пространства имен System.Collections указанную в варианте с параметром строка и выполните ввод-вывод элементов, подсчет количества строк длины n, сортировку по длине строк (если не отсортированы).

Вариант	Задание
1, 9	создать **массив объектов** CollectionType, запросы – найти объекты размера n; выберите максимальный по размеру объект в массиве, проверьте все ли объекты в массиве соответствуют какому-либо условию. Обобщенная коллекция – LinkedList <T>
2, 10	создать массив объектов CollectionType, запросы – найти объекты с отрицательными элементами, верните три первые объекта в массиве, найдите все объекты размером меньше 2-х. Обобщенная коллекция – Dictionary<T>.
3, 11	создать массив объектов CollectionType, запросы – посчитать количество объектов равных заданному, найти максимальный и объект в массиве, Обобщенная коллекция – List<T>
4, 12	создать массив объектов CollectionType, запросы - найти количество объектов содержащих 0, посчитать количество объектов из заданного диапазона. Обобщенная коллекция – Queue<T>

5, 13	создать массив объектов CollectionType, запросы - найти количество объектов, содержащих заданное значение, получите все объекты из массива за исключением первых 3-х . Обобщенная коллекция – SortedSet<T>
6, 14	создать массив объектов CollectionType, запросы - найти количество объектов сумма элементов которых больше заданного значения, получите массив объектов, содержащий первый и последний элемент исходного массива. Обобщенная коллекция – ArrayList<T>
7, 15	создать массив объектов CollectionType, запросы - найти объекты, содержащие только положительные элементы, найти максимальный объект в массиве, верните массив без дублированных объектов. Обобщенная коллекция – SortedSet<T>
8, 16	создать массив объектов CollectionType, запросы - найти объекты заданной длины n, найти последний объект удовлетворяющий условию пользователя и вернуть N-ый объект в массиве. Обобщенная коллекция – ArrayList <T>

Вопросы

1. В какой строке определения интерфейса содержится ошибка?

```
public interface INumber //1
    {
        INumber();//2
        void Reset();//3
        void SetStart(int x);//4
             }
```

2. Что такое обобщение (generic)?
3. Пусть дан фрагмент листинга. В какой строчке содержится ошибка?

```
class Gen<T,G>  //1
    {
        G ob;//2
        T bo;//3
        public Gen(G o) { ob = o; } //4
        public T GetOb() { return bo; } //5
    }
```

4. Как можно наложить определенное ограничение на параметр?
5. Как можно наложить несколько ограничений на параметр?
6. Перечислите все существующие ограничения на типы данных обобщения?
7. Какое ограничение на тип задано в следующем фрагменте листинга?

```
class A   { }
class B : A { }
class C { }
class Test<T> where T : A {  }
```

8. Какое ограничение на тип задано в следующем фрагменте листинга?

```
interface A   { }
class Test<T> where T : class {  }
```

9. Какое ограничение на тип задано в следующем фрагменте листинга?

```
interface A   { }
class Test<T> where T : struct {  }
```

10. Приведите примеры, когда обобщенный класс может действовать как базовый или производный класс.
11. В каких случаях в обобщениях может использоваться оператор default?
12. Поясните как использовать статические переменные в обобщенных классах.
13. Приведите пример обобщенного интерфейса.
14. В чем отличие обобщенных классов от обобщенных структур?
15. Поясните, для чего классам может понадобиться реализовывать интерфейс IComparable?
16. Поясните, для чего классам может понадобиться реализовывать интерфейс IEquatable<T>?
17. Что такое ковариантность обобщенного интерфейса?
18. Перечислите стандартные коллекции NET Framework.
19. Охарактеризуйте необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные коллекции.
20. Выберите из списка, приведенного ниже, обобщенную коллекцию?

Stack<T>
NameValueCollection
StringCollection
ArrayList

21. Какие интерфейсы используются в коллекциях C#?
22. Для чего используется интерфейс IComparable?
23. Что содержит интерфейс IEnumerator или обобщенный интерфейс IEnumerator<T>? Где и как его можно использовать?
24. Как и когда вызывается блок finally при обработке исключений?
25. Что произойдет, если исключение не перехватывается в программе?
26. Перечислите и охарактеризуйте члены класса System.Exception.
27. Что будет выведено на консоль в результате выполнения фрагмента листинга?

```
static void Main(string[] args)
        {

            string[] str = new string[5];
            try
            {
                str[5] = "anything";
                Console.WriteLine("It's OK");
            }
            catch (IndexOutOfRangeException e)
            {
                Console.WriteLine("IndexOutOfRangeException");
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception");
            }

        }
```

28. Что такое LINQ?
29. В чем разница между отложенными операциями и не отложенными операциями LINQ to Object?
30. Что такое лямбда-выражения?
31. Как используется операция Where в LINQ to Object?
32. Как используется операция Select ?
33. Как используются операции Take, Skip?
34. Как используется операция Concat ?
35. Как используется операция OrderBy?
36. Как используется операция Join?
37. Как используются операции Distinct, Union, Except и Intersect?
38. Как используются операции First, Last, Any, All и Contains?
39. Как используются операции Count, Sum, Min и Max, Average?
40. Что выведет на экран данный код?

```
class Test
    {
        public static void Main()
        {
            List<int> list = new List<int>();
            list.AddRange(new int[] { 3, 1, 4, 8, 10, 4 });

            List<int> some = list.FindAll(i => (i>=9));

            foreach (int x in some)
                Console.Write(x);

            }
     }
```

Краткие теоретические сведения

Обобщения

Термин *обобщение* означает параметризированный тип. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра. Схожие с обобщениями черты имеют шаблоны C++. Однако между шаблонами C++ и обобщениями .NET есть большая разница. В C++ при создании экземпляра шаблона с конкретным типом необходим исходный код шаблонов. В отличие от шаблонов C++, обобщения являются не только конструкцией языка C#, но также определены для CLR. Это позволяет создавать экземпляры шаблонов с определенным типом-параметром на языке Visual Basic, даже если обобщенный класс определен на C#.

Общая форма объявления обобщенного класса:

```
class имя_класса<список_параметров_типа> { // ...
```

А вот как выглядит синтаксис объявления ссылки на обобщенный класс:

```
имя_класса<список_аргументов_типа> имя_переменной =
new имя_класса<список_параметров_типа> (список_аргументов_конструктора);
```

Рассмотрим пример использования нескольких обобщенных классов:

```
namespace ConsoleApplication1
{
    // Создадим обобщенный класс имеющий параметр типа T
    class MyObj<T>
```

```
    {
        T obj;

        public MyObj(T obj)
        {
            this.obj = obj;
        }

        public void objectType()
        {
            Console.WriteLine("Тип объекта: " + typeof(T));
        }
    }

    // Обобщенный класс с несколькими параметрами
    class MyObjects<T, V, E>
    {
        T obj1;
        V obj2;
        E obj3;

        public MyObjects(T obj1, V obj2, E obj3)
        {
            this.obj1 = obj1;
            this.obj2 = obj2;
            this.obj3 = obj3;
        }

        public void objectsType()
        {
            Console.WriteLine("\nТип объекта 1: " + typeof(T) +
                "\nТип объекта 2: " + typeof(V) +
                "\nТип объекта 3: " + typeof(E));
        }
    }

    class Program
    {
        static void Main()
        {
            // Создадим экземпляр обобщенного класса типа int
            MyObj<int> obj1 = new MyObj<int>(25);
            obj1.objectType();

            MyObjects<string, byte, decimal> obj2 = new MyObjects<string,
byte, decimal>("Alex", 26, 12.333m);
            obj2.objectsType();
            Console.ReadLine();
        }
    }
}
```

Когда для класса MyObj указывается аргумент типа, например, int или string, то создается так называемый в C# закрыто сконструированный тип. В частности, MyObj<int> является закрыто сконструированным типом. А конструкция, подобная MyObj<T>, называется в C# открыто

сконструированным типом, поскольку в ней указывается параметр типа T, но не такой конкретный тип, как int.

Ограниченные типы

Допустим, что требуется создать метод, оперирующий содержимым потока, включая объекты типа FileStream или MemoryStream. На первый взгляд, такая ситуация идеально подходит для применения обобщений, но при этом нужно каким-то образом гарантировать, что в качестве аргументов типа будут использованы только типы потоков, но не int или любой другой тип. Кроме того, необходимо уведомить компилятор о том, что методы, определяемые в классе потока, будут доступны для применения. Так, в обобщенном коде должно быть каким-то образом известно, что в нем может быть вызван метод Read().

Для выхода из подобных ситуаций в C# предусмотрены ограниченные типы. Указывая параметр типа, можно наложить определенное ограничение на этот параметр. Это делается с помощью оператора where при указании параметра типа:

```
class имя_класса<параметр_типа> where параметр_типа : ограничения { // ...
```

где ограничения указываются списком через запятую.

Связь между параметрами типа с помощью ограничений

Существует разновидность ограничения на базовый класс, позволяющая установить связь между двумя параметрами типа. В качестве примера рассмотрим следующее объявление обобщенного класса:

```
class MyObj<T, V> where V : T { ...
```

В этом объявлении оператор where уведомляет компилятор о том, что аргумент типа, привязанный к параметру типа V, должен быть таким же, как и аргумент типа, привязанный к параметру типа T, или же наследовать от него. Если подобная связь отсутствует при объявлении объекта типа MyObj, то во время компиляции возникнет ошибка. Такое ограничение на параметр типа называется *неприкрытым ограничением типа*.

С параметром типа может быть связано несколько ограничений. В этом случае ограничения указываются списком через запятую. В этом списке первым должно быть указано ограничение class либо struct, если оно присутствует, или же ограничение на базовый класс, если оно накладывается. Указывать ограничения class или struct одновременно с ограничением на базовый класс не разрешается. Далее по списку должно следовать ограничение на интерфейс, а последним по порядку — ограничение new().

Например, следующее объявление считается вполне допустимым:

```
class MyObj<T> where T : MyClass, IMyInterface, new() {
      // ...
```

В данном случае параметр типа T должен быть заменен аргументом типа, наследующим от класса MyClass, реализующим интерфейс IMyInterface и использующим конструктор без параметра.

Если же в обобщении используются два или более параметра типа, то ограничения на каждый из них накладываются с помощью отдельного оператора where.

Обзор коллекций

В C# **коллекция** представляет собой совокупность объектов. В среде .NET Framework имеется немало интерфейсов и классов, в которых определяются и реализуются различные типы коллекций.

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе ArrayList, Hashtable, Stack и Queue, могут применяться в исходном виде и без каких-либо изменений. Имеется также возможность реализовать собственную коллекцию, хотя потребность в этом возникает редко.

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

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.

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**.

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>. Это означает, что, внеся минимальные изменения в код циклического обращения к коллекции одного типа, его можно использовать для аналогичного обращения к коллекции другого типа. Любопытно, что для поочередного обращения к содержимому коллекции в цикле foreach используется перечислитель.

С перечислителем непосредственно связано другое средство, называемое *итератором*. Это средство упрощает процесс создания классов коллекций, например специальных, поочередное обращение к которым организуется в цикле foreach.

Классы коллекций по своей сути подобны классам стандартной библиотеки шаблонов (Standard Template Library — STL), определенной в C++. То, что в программировании на C++ называется контейнером, в программировании на C# называется коллекцией.

Интерфейсы обобщенных коллекций

В пространстве имен System.Collections.Generic определен целый ряд интерфейсов обобщенных коллекций, имеющих соответствующие аналоги среди интерфейсов необобщенных коллекций:

ICollection<T>

Определяет основополагающие свойства обобщенных коллекций

IComparer<T>

Определяет обобщенный метод Compare() для сравнения объектов, хранящихся в коллекции

IDictionary<Tkey, TValue>

Определяет обобщенную коллекцию, состоящую из пар "ключ-значение"

IEnumerable<T>

Определяет обобщенный метод GetEnumerator(), предоставляющий перечислитель для любого класса коллекции

Enumerator<T>

Предоставляет методы, позволяющие получать содержимое коллекции по очереди

IEqualityComparer<T>

Сравнивает два объекта на предмет равенства

IList<T>

Определяет обобщенную коллекцию, доступ к которой можно получить с помощью индексатора

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

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

В этих свойствах хранятся ключ и значение соответствующего элемента коллекции.

Классы обобщенных коллекций

Классы обобщенных коллекций по большей части соответствуют своим необобщенным аналогам, хотя в некоторых случаях они носят другие имена. Отличаются они также своей организацией и функциональными возможностями. Классы обобщенных коллекций определяются в пространстве имен System.Collections.Generic:

Dictionary<Tkey, TValue>

Сохраняет пары "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс Hashtable

HashSet<T>

Сохраняет ряд уникальных значений, используя хештаблицу

LinkedList<T>

Сохраняет элементы в двунаправленном списке

List<T>

Создает динамический массив. Обеспечивает такие же функциональные возможности, как и необобщенный класс ArrayList

Queue<T>

Создает очередь. Обеспечивает такие же функциональные возможности, как и необобщенный класс Queue

SortedDictionary<TKey, TValue>

Создает отсортированный список из пар "ключ-значение"

SortedList<TKey, TValue>

Создает отсортированный список из пар "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс SortedList

SortedSet<T>

Создает отсортированное множество

Stack<T>

Создает стек. Обеспечивает такие же функциональные возможности, как и необобщенный класс Stack

LINQ - язык интегрированных запросов

LINQ to Objects

LINQ to Objects - название, данное API-интерфейсу IEnumerable<T> для *стандартных операций запросов (Standard Query Operators)*. Именно LINQ to Objects позволяет выполнять запросы к массивам и находящимся в памяти коллекциям данных. Стандартные операции запросов - это статические методы класса System.Linq.Enumerable, которые используются для создания запросов LINQ to Objects.

LINQ to XML

LINQ to XML — название, назначенное API-интерфейсу LINQ, который ориентирован на работу с XML. В Microsoft не только добавили необходимые библиотеки XML для работы с LINQ, но также восполнили недостатки стандартной модели XML DOM, существенно облегчив работу с XML. Прошли времена, когда нужно было создавать XmlDocument только для того, чтобы поработать с небольшим фрагментом XML-кода. Чтобы воспользоваться преимуществами LINQ to XML, в проект понадобится добавить ссылку на сборку System.Xml.Linq.dll и директиву using System.Xml.Linq.

LINQ to DataSet и SQL

LINQ to DataSet — название, данное API-интерфейсу LINQ, который предназначен для работы с DataSet. У многих разработчиков есть масса кода, полагающегося на DataSet. Те, кто не хотят отставать от новых веяний, но и не готовы переписывать свой код, благодаря этому интерфейсу могут воспользоваться всей мощью LINQ.

LINQ to SQL — наименование, присвоенное API-интерфейсу IQueryable<T>, который позволяет запросам LINQ работать с базой данных Microsoft SQL Server. Чтобы воспользоваться преимуществами LINQ to SQL в проект понадобится добавить ссылку на сборку System.Data.Linq.dll, а также директиву using System.Data.Linq.

LINQ to Entities

LINQ to Entities — альтернативный API-интерфейс LINQ, используемый для обращения к базе данных. Он отделяет сущностную объектную модель от физической базы данных, вводя логическое отображение между ними двумя. С таким отделением возрастает мощь и гибкость, но также растет и сложность. Если нужна более высокая гибкость, чем обеспечивается LINQ to SQL, имеет смысл рассмотреть эту альтернативу.

В частности, когда необходимо ослабить связь между сущностной объектной моделью и базой данных, если сущностные объекты конструируются из нескольких таблиц или требуется большая гибкость в моделировании сущностных объектов, то в этом случае LINQ to Entities может стать оптимальным выбором.

Parallel LINQ

Формально отдельного продукта LINQ, который нужно было бы получать отдельно, не существует. LINQ полностью интегрирован в .NET Framework, начиная с версии 3.5 и Visual Studio 2008. В NET 4.0 и Visual Studio 2010 добавлена поддержка средств Parallel LINQ.

Запросы LINQ

Одним из привлекательных для разработчиков средств LINQ является SQL-подобный синтаксис, доступный в LINQ-запросах. Синтаксис предоставлен через расширение языка C#, которое называется выражения запросов. Выражения запросов позволяют запросам LINQ принимать форму, подобную SQL, всего лишь с рядом небольших отличий.

Для выполнения запроса LINQ выражения запросов не обязательны. Альтернативой является использование стандартной точечной нотации C# с вызовом методов на объектах и классах. Во многих случаях применение стандартной точечной нотации оказывается более предпочтительным, поскольку она более наглядно демонстрирует, что в действительности происходит и когда.

При записи запроса в стандартной точечной нотации не происходит никакой трансформации при компиляции. Именно поэтому во многих примерах не используется синтаксис выражений запросов, а предпочтение отдается стандартному синтаксису точечной нотации. Получить

представление о различиях между этими двумя синтаксисами лучше всего на примере:

```
string[] names = {
        "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
        "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford",  "Garfield",
        "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
        "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
        "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
        "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington",
"Wilson"};

      // Использование точечной нотации
      IEnumerable<string> sequence = names
        .Where(n => n.Length < 6)
        .Select(n => n);

      // Использование синтаксиса выражения запроса
      IEnumerable<string> sequence = from n in names
                                     where n.Length < 6
                                     select n;

      foreach (string name in sequence)
      {
        Console.WriteLine("{0}", name);
      }
```

Синтаксис выражений запросов поддерживается только для наиболее распространенных операций запросов: Where, Select, SelectMany, Join, GroupJoin, GroupBy, OrderBy, ThenBy, OrderByDescending и ThenByDescending.

Выражения запросов должны подчиняться перечисленным ниже правилам:

1. Выражение должно начинаться с конструкции from.
2. Остальная часть выражения может содержать ноль или более конструкций from, let или where. **Конструкция from** — это генератор, который объявляет одну или более переменных диапазона, перечисляющих последовательность или соединение нескольких последовательностей. **Конструкция let** представляет переменную диапазона и присваивает ей значение. **Конструкция where** фильтрует элементы из входной последовательности или соединения несколько входных последовательностей в выходную последовательность.
3. Остальная часть выражения запроса может затем включать конструкцию orderby, содержащую одно или более полей сортировки с необязательным направлением упорядочивания. Направлением может быть *ascending (по возрастанию)* или*descending (по убыванию)*.
4. Затем в оставшейся части выражения может идти конструкция select или group.

5. Наконец в оставшейся части выражения может следовать необязательная конструкция продолжения. Такой конструкцией может быть либо into, ноль или более конструкций join, или же другая повторяющаяся последовательность перечисленных элементов, начиная с конструкций из правила 2. **Конструкция into** направляет результаты запроса в воображаемую выходную последовательность, которая служит конструкцией from для последующих выражения запросов, начиная с конструкций из правила 2.