Разработка .NET приложений

Лекция 6

Коллекции
Итераторы

Сегодня

- Коллекции
- Итераторы

Сегодня

- Коллекции
- Итераторы

Коллекции

- Коллекция
  - Класс (объект), основное назначение которого – содержать в себе другие объекты по определенной дисциплине
- Примеры коллекций
  - Массив, динамический массив
  - Список, стек, очередь
  - Дерево, множество, словарь (хэш-таблица)

Коллекции в .NET

- Основные пространства имен:
- Не обобщенные коллекции
  - **System.Collections**
    - Нетипизированные коллекции
    - Специализированные коллекции
    - ArrayList, Hashtable и др.
  - **System.Collections.Specialized**
    - Специализированные коллекции

до .NET 2

- **Обобщенные коллекции**
  - **System.Collections.Generic**
    - Обобщенные коллекции
    - Список, очередь, словарь, стек
  - **System.Collections.ObjectModel**
    - Базовые реализации для собственных коллекция
    - Специальные обобщенные коллекции
  - **System.Collections.Concurrent**
    - Коллекции, позволяющие обращения из нескольких потоков

с .NET 2

с .NET 4

System.Collections

- **Не типизированные коллекции:**
- **ArrayList** – Массив object переменной длинны
- **HashTable** – Таблица пар ключ/значение, основанная на хранении хэш-кода ключа
- **BitArray** - Компактный массив битовых значений
- **Stack** – Стек object
- **Queue** – Очередь object
- **SortedList** - Сортированный по ключам словарь
- **Практически не используются, поскольку есть обобщенные версии (появились в .NET 2)**

System.Collections.Specialized

- **Старые специализированные коллекции:**
- **ListDictionary** – не типизированный словарь пар ключ/значение, ориентированный на хранение до 10 значений
- **HybridDictionary**– не типизированный словарь пар ключ/значение, который до 10 значений хранит данные как ListDictionary, а более – преобразуется в HashTable
- **OrderedDictionary**- словарь, позволяющий индексировать значения по номеру
- **StringCollection**– представляет собой коллекцию строк
- **StringDictionary**– словарь, где и ключ и значение - string
- **BitVector32**– структура для компактного хранения 32 bool значений.
- **Практически не используются, поскольку есть обобщенные версии (появились в .NET 2)**

System.Collections.Generic

- **Обобщенный коллекции:**
- **List<T>** - Список Переменной длинны
- **LinkedList<T>** - Двунаправленный связанный список переменной длинны
- **Stack<T>** - Стек
- **Queue<T>** - Очередь
- **Dictionary<TKey, TValue>** - Словарь
- **SortedList<TKey, TValue>** и **SortedDictionary <TKey, TValue>** - Сортированный по ключам словарь
- **HashSet <T>** - Множество
- **SortedSet<T>** - Сортированное множество

System.Collections.ObjectModel

- Содержит базовые классы для создания своих коллекций
  - Collection<T>
  - KeyedCollection <TKey, TItem>

- Тип ObservableCollection<T> - динамическая обобщенная коллекция, которая генерирует события при изменении коллекции (добавлении, удалении элемента или при обновлении всего списка)
  - Важное применение - **WPF**

- Коллекции обертки только для чтения
  - ReadOnlyCollection <T>
  - ReadOnlyDictionary<TKey, TValue>
  - ReadOnlyObservableCollection <T>

System.Collections.Concurrent

Потокобезопасные коллекции. Позволяют получать доступ к коллекции из нескольких потоков.
Содержат встроенные, облегченные механизмы синхронизации потоков

- ConcurrentQueue<T> - потокобезопасная очередь
- ConcurrentStack<T> - потокобезопасный стек
- ConcurrentDictionary<TKey, TValue> - потокобезопасный словарь
- ConcurrentBag<T> - потокобезопасный не упорядоченный список
- BlockingCollection<T> - потокобезопасная коллекция, реализующая Producer/Consumer паттерн

Коллекции

- Обобщенные **vs.** Необобщенные
  - Обобщенные коллекции – контроль типов во время компиляции, необобщенные – контроля типов нет или он осуществляется во время выполнения
  - В обобщенных коллекциях не используется **boxing/unboxing** при использовании типов-значений
  - Обобщенные коллекции обладают более высокой производительностью
- Не обобщенные коллекции используются редко (исключение – унаследованный код)

List<T>

- Список
  - Динамически изменяет размер
  - Обладает возможностью доступа по индексу
  - Позволяет осуществлять поиск и сортировку
- Создание
  - List<int> list1 = new List<int>();
  - List<int> list2 = new List<int>(10);
  - List<string> strList = new List<string>()
    {
    “один”,
    “два”
    };
  - List<Complex> complexList = new List<Complex>()
    { new Complex() {Re =5, Im =7},
    new Complex() {Re =3, Im =1}, };
- Добавление
  - Элемента(ов) (в конец)
    Add(T item),
    AddRange(IEnumerable<T> collection)
    - list1.Add(5);
    - list2.AddRange(list1);
  - элемента(ов) в произвольное место
    Insert(int index, T item),
    InsertRange(int index, IEnumerable<T> collection)
- Удаление
  - элемента Remove(T item)
    - complexList.Remove(complex2);
  - Элемента(ов) по индексу
  - RemoveAt(int index),
  - RemoveRange(int index, int count)
  - Всех элементов Clear()

List<T>

- Доступ к элементу по индексу [int index]
  - int j = l[3];
- Количество элементов - свойство Count
  - int k = l.Count;
- Сортировка элементов Sort()
  - list1.Sort();
- Преобразование в массив T[] ToArray()
  - int[] arr = list1.ToArray();
- Поиск
  - bool Contains(T item) – проверка – содержится ли элемент в списке
  - bool Exists(Predicate<T> match) – проверка – содержится ли элемент удовлетворяющий условию
  - T Find(Predicate<T> match)– Поиск первого элемента, удовлетворяющего условию
  - int IndexOf(T item) – возвращает индекс элемента

Демонстрации

Работа со списком

Сортировка комплексных чисел

ReadOnlyCollection

Stack<T>

- Обобщенный стек
  - Last In First Out (LIFO). Последним вошёл, первым вышел
  - Динамически изменяет размер
- Создание
  - Stack<Complex> s = new Stack<Cpmplex>();
- Добавление элемента Push()
  - s.Push(3);
- Удаление элемента T Pop(). Очистка всего стека Clear()
  - int k = s.Pop();
- Просмотр элемента в вершине стека T Peek()
  - int k = s. Peek();
- Количество элементов - свойство Count
  - if (s.Count == 0) …
- bool Contains(T item) – проверка, содержится ли данный элемент в стеке
- Преобразование в массив T[] ToArray()
  - int[] k = s. ToArray();

Queue<T>

- Обобщенная очередь
  - first-in, first-out (FIFO). Первым вошел, первым вышел
  - динамически изменяет размер
- Создание
  - Queue<Complex> q = new Queue<Cpmplex>();
- Добавление элемента Enqueue(T item)
  - q.Enqueue(new Complex(4,5));
- Удаление элемента T Dequeue(). Очистка всей очереди Clear()
  - int k = s. Dequeue();
- Просмотр элемента в начале очереди T Peek()
  - int k = s. Peek();
- Количество элементов - свойство Count
  - if (s.Count == 0) …
- bool Contains(T item) – проверка, содержится ли данный элемент в очереди
- Преобразование в массив T[] ToArray()
  - int[] k = s. ToArray();

Dictionary<TKey, TValue>

- Словарь
  - Содержит коллекцию ключей и соответствующих им значений
  - Динамически изменяет свой размер
- Создание
  - Dictionary<string, Complex> d = new Dictionary<string, Complex>();
- Добавление Add(TKey key, TValue value)
  - d.Add("два", new Complx(4,5));
  - s.Add("один", "one");
- Удаление Remove(TKey key), всех записей Clear()
  - d.Remove(myComplexVar);
- Доступ по ключу [Tkey key]
  - Complex complexVar = d["два"];
- Безопасное получение значения по ключу bool TryGetValue(TKey key, out TValue value)

  if (d.TryGetValue("два", out var myComplexVar)) …
- Проверка, содержится ли в словаре
  - Ключ - bool ContainsKey(TKey key)
  - Значение - bool ContainsValue(TValue value)
- Количество элементов в словаре – свойство Count
- Коллекции (свойства)
  - Keys – коллекция ключей
  - Values – коллекция значений

Инициализация

```
Dictionary<int, string> numbers = new Dictionary<int, string>
    {
        {7, "seven"},
        { 9, "nine"},
        { 13, "thirteen"}
    };
```

- **Ключом может быть произвольный тип**

```
Dictionary<Complex, Vector3d> projection = new Dictionary<Complex, Vector3d>()
    {
        {new Complex(4, 5),  new Vector3d(3, 4, 5)},
        {new Complex(5, 6),  new Vector3d(4, 5, 6)},
        {new Complex(5, 7),  new Vector3d(4, 5, 6)},
        { complex, vector3d}
    };
```

- **Способ инициализации с указанием ключа**

```
Dictionary<int, string> numbers = new Dictionary<int, string>
    {
        [7] = "seven",
        [9] = "nine",
        [13] = "thirteen"
    };
```

Множества

- **HashSet<T>** - Множество
- **SortedSet<T>** - Сортированное множество
- Не позволяют дублировать значения (не добавляют дубли)
- Имеют операции на объединение, вычитание, пересечение множеств

- Add(T item) – добавление элемента в множество
- bool Remove(T item) – удаление элемента из множества. RemoveWhere(Predicate<T> match) – по условию

- UnionWith(IEnumerable<T> other) – объединение 2 множеств
- Overlaps(IEnumerable<T> other) – пересечение множеств
- Проверка на вхождение одного множества в другое
  - IsSubsetOf(IEnumerable<T> other), IsProperSubsetOf(IEnumerable<T> other)
  - IsSupersetOf(IEnumerable<T> other), IsProperSupersetOf(IEnumerable<T> other)
- ExceptWith(IEnumerable<T> other) – вычитание множеств

- Contains(T item) – проверка на вхождение элемента во множество
- Свойство Count – количество элементов в множестве

Интерфейсы коллекций

```
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface ICollection<T> : IEnumerable<T>
  {
    int Count { get; }
    void Add(T item);
    bool Remove(T item);
    void Clear();
    bool Contains(T item);
    bool IsReadOnly { get; }
    void CopyTo(T[] array, int arrayIndex);
  }

public interface IList<T> : ICollection<T>, ICollection
  {
    T this[int index] {get;set; }
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);
  }

public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>
  {
    void Add(TKey key, TValue value);
    bool Remove(TKey key);
    bool TryGetValue(TKey key, out TValue value);
    TValue this[TKey key] { get; set; }
    bool ContainsKey(TKey key);
    ICollection<TKey> Keys { get; }
    ICollection<TValue> Values { get; }
  }
```

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

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

public interface IList : ICollection
  {
    int Add(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);
    void Clear();
    object this[int index] { get; set; }
    int IndexOf(object value);
    bool IsReadOnly { get; }
    bool IsFixedSize {get; }
    bool Contains(object value);
  }

public interface IDictionary : ICollection
  {
    void Add(object key, object value);
    void Remove(object key);
    void Clear();
    object this[object key] { get; set; }
    bool Contains(object key);
    ICollection Keys { get; }
    ICollection Values { get; }
    bool IsReadOnly { get; }
    bool IsFixedSize { get; }
}
```

Сегодня

- Коллекции
- Итераторы

Цикл foreach

- Цикл по всем элементам массива или коллекции (реализующий интерфейс IEnumerable или IEnumerable<T>)
- Синтаксис

```
foreach (Тип_элемента Имя_переменной in Коллекция)
{
        // Можно использовать элемент коллекции через Имя_переменной
}
```

- Запрещено изменять саму коллекцию (добавлять, удалять элементы) внутри цикла

- Пример:

```
List<Complex> collection = new List<Complex>();
…        // Заполнение коллекции
foreach (Complex comp in collection)
{
        Console.WriteLine(comp);
        // collection.Add(new Complex(5,4));  Так нельзя. Нельзя изменять саму коллекцию
        comp.Re = 7; // Так можно. Не меняет саму коллекцию, а меняет внутренности элемента
}
```

Итераторы

- Итератор
  - Предоставляет единый способ перебора элементов коллекции
  - Не обязательно перебрать все элементы. Последовательность может быть бесконечной
  - Можно использовать в специальном цикле **`foreach`**`()`
- По чему можно итерировать (перебирать элементы)
  - По классу, реализующему интерфейс **IEnumerable** или **IEnumerable<T>**
  - По методу (свойству), возвращающему **IEnumerator** или **IEnumerator<T>**
- Все **.NET** коллекции реализуют интерфейс **IEnumerable** или **IEnumerable<T>**. Следовательно по ним можно итерировать

IEnumerable и IEnumerator

- Для возможности перебора элементов класс должен реализовывать интерфейс **IEnumerable**
- **IEnumerable:**

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

- Метод **GetEnumerator()** должен возвращать тип, реализующий интерфейс **IEnumerator**
- **IEnumerator:**

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

IEnumerable<T> и IEnumerator<T>

- Интерфейс **IEnumerable<T>** - наследник от **IEnumerable**

```
interface IEnumerable<T> : IEnumerable
{
        IEnumerator<T> GetEnumerator();
        IEnumerator GetEnumerator();
}
```

- Метод **GetEnumerator()** должен возвращать тип, реализующий интерфейс **IEnumerator** и **IEnumerator <T>** (необходима явная реализация одного из методов)

```
interface IEnumerator<T> : IEnumerator, IDisposable
{
        T Current {get;}
        object Current {get;}
        bool MoveNext();
        void Reset();
        Dispose();
}
```

Демонстрации

Собственный итератор

Реализация итератора

- Квазиключевое слово **yield**
  - **yield return** *[expression]*
  - **yield break;**

- Когда встречается **yield**
  - Запоминается состояние и возвращается текущее значение
  - На следующей итерации продолжаем с этого состояния
- Количество итерируемых элементов не обязательно конечно

Примеры:

```
public IEnumerator<double> GetEnumerator()
    {
        yield return x;
        yield return y;
        yield return z;
        yield break;
    }
```

```
int odd = -1;
public IEnumerable<int> GetCustomCollection()
{
    while (true) yield return odd+=2;
}
```

Демонстрации

Коллекции и Итераторы (yield)
Бесконечные последовательности