



Информатика

абстрактные типы данных,
коллекции и структуры данных

План лекции

- Типы данных
- Обобщенные типы (generics)
- Абстрактные типы данных
- Коллекции и структуры данных
- Коллекции в C#

Понятие типа данных

- Тип данных:
 - значения
 - операций по работе со значениями
- В ООП - класс, характеризуемый полями и методами
 - + свойствами в С# (тоже методы по сути)

Обобщенные типы данных дженерики (generics)

- Параметризованные типы
 - шаблоны типов
- Могут переиспользоваться с подстановкой других типов данных

Зачем дженерики?

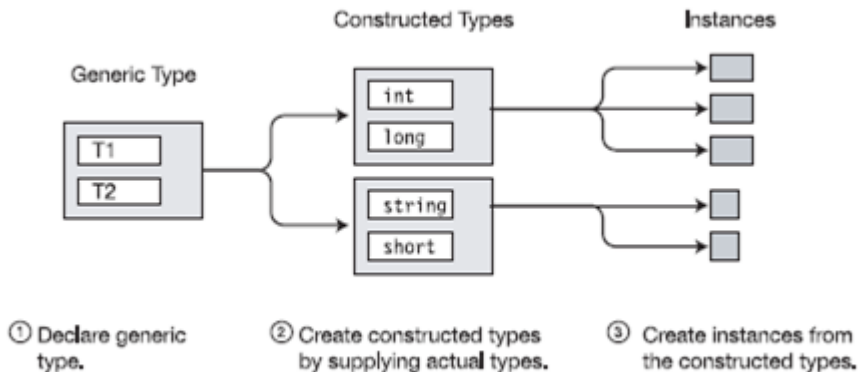
- Они типобезопасны
- Повышают производительность (исключают боксинг-анбоксинг при работе с типами значения)
- Уменьшают дублирование кода
- Облегчают повторное использование

Возможности

- позволяют инициализировать обобщенные переменные значениями по умолчанию `default(T)`
- позволяют накладывать ограничения на тип параметра
- могут наследоваться
- могут применяться к статическим членам
- **могут применяться к классам, методам, структурам, интерфейсам и делегатам**

Работа с обобщенными типами

- При объявлении использовать шаблоны типов
- Подставлять типы для получения сконструированного типа (constructed type)
- Создать экземпляр сконструированного типа



Объявление

- Параметры типов указываются при объявлении в угловых скобках через запятую (<, >)
- Параметры используются в теле классов и методов

```
class MyGenericCass<T1, T2>
{
    public T1 SomeVar;
    public T1 ReturnDefaultGenericValue1()
    {
        return default(T1);
    }


    public T2 ReturnDefaultGenericValue2()
    {
        return default(T2);
    }
}
```


Создание сконструированного типа

- Нельзя создавать экземпляры обобщенного типа не указав конкретные значения параметров
- Нужно сперва создать сконструированный тип (constructed type)

```
class SomeClass<T1, T2>
{
    T1 var1;
    T2 var2;
}
```


Type Parameters



```
class SomeClass< T1, T2 >
{
    ...
}
```

Generic Class Declaration

Type Arguments



```
SomeClass< short, int >
```

Constructed Type

```
class SomeClass<short, int>
{
    short var1;
    int var2;
}
```

**пример того, как представляется сконструированный тип.
не является допустимой синтаксической конструкцией!**

Подстановка типа и экземпляры

- Сконструированный тип используется так же, как обычный при создании экземпляров и объявлении ссылок

```
var someVar1 = new SomeClass();
```

```
var someVar2 = new SomeGenericClass<int, double>();
```

Generic vs Non-Generic

| Characteristics | Non generic | Generic |
|------------------------|---|---|
| Source Code Size | Larger: You need a new implementation for each type | Smaller: You need only one implementation regardless of the number of constructed types |
| Executable Size | The compiled version of each stack is present, regardless of whether it is used. | Only types for which there is a constructed type are present in the executable. |
| Ease of Writing | Easier to write because it's more concrete | Harder to write because it's more abstract |
| Difficulty to Maintain | More error-prone to maintain, since all changes need to be applied for each applicable type | Easier to maintain, because modifications are needed in only one place |


Ограничения (constraints)

- На все типы параметров могут накладываться ограничения
- **Типы ограничений:**
 - **<ClassName>** : разрешаются только классы из иерархии
 - **class** : разрешаются только ссылочные типы
 - **struct** : разрешаются только типы значения
 - **<InterfaceName>** : разрешаются только типы, реализующие интерфейс
 - **new()** : разрешаются только типы с беспараметрическим публичным конструктором

Синтаксис ограничений

- Ограничения для каждого типа параметра указываются в условии where
- В случае нескольких ограничений, они перечисляются через запятую в условии where
- Условия where могут описываться в произвольном порядке

```
class SomeClass<T1, T2> where T1 : class, new()  
                           where T2 : IEnumerable  
{  
    ...  
}
```



ограничение — конструктор
указывается последним

Обобщенные методы

- Методы с двумя типами параметров
 - формальными параметрами метода (в круглых скобках)
 - параметрами типа метода (в угловых скобках)
- Могут содержать ограничения (where)
- Могут вызываться только при подстановке типов в параметры

Обобщенные методы

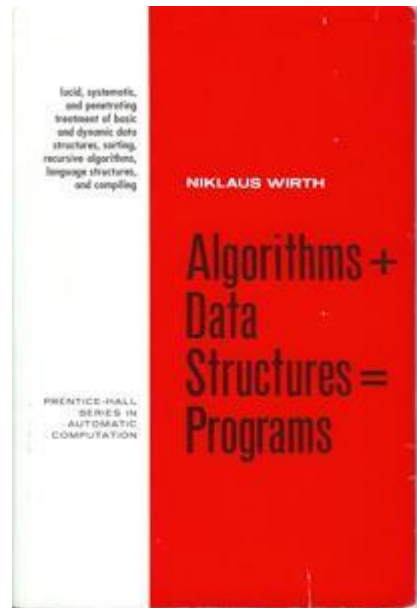
```
public static T1 SomeMethod<T1, T2>(T2 input) where T1 : class
                                         where T2 : new()
{
    ...
}
```

Вызов с подстановкой типов параметров

```
var val=SomeMethod<MyClass, MyClassWithCtor>(new MyClassWithCtor());
```

Если компилятор может определить тип по переданному параметру, указывать тип параметра в угловых скобках не обязательно

На этом всё про дженерики...



Далее поговорим про алгоритмы и структуры данных

Абстрактный тип данных

- Определяет множество значений типа и операции со значениями этого типа
- Не описывает способ реализации
- Модель, абстрактная сущность для упрощения описания абстрактных алгоритмов

Абстрактный тип данных

- Класс может реализовывать несколько абстрактных типов данных
- С другой стороны, один АТД могут реализовывать несколько классов

Хмм... абстрактность и реализация
что-то напоминают...

АТД и интерфейсы

- Абстрактные типы данных хорошо *представляются интерфейсами* (ключевое слово interface)
- Конкретные типы данных могут реализовать несколько интерфейсов – Абстрактных Типов Данных

АТД и проектирование

- Абстрактными типами данных описываются абстракции предметной области
- На АТД разбивается предметная область при её моделировании
- АТД – то, как следует думать о наших типах, когда разрабатываем программы

Пример: точки на плоскости

- **Задача:** реализовать **сложение** и **проверку на равенство** для точек на плоскости
- И ещё, точки могут быть представлены *в декартовых и полярных координатах*, поэтому нужно дать возможность **получать координаты** обоих видов

Интерфейс для точки

```
public interface IPoint
{
    // Добавление точки
    void Add(IPoint other);

    // Проверка двух точек на равенство с точностью
    bool EqualTo(IPoint other, double tolerance=0.0000001);

    // Получение декартовых координат точки
    Tuple<double, double> ToCartesian();

    // Получение полярных координат точки
    Tuple<double, double> ToPolar(bool degreeAngle=false);
}
```

Конкретные типы точек

- **CartesianPoint** – реализация точки с хранением координат в декартовой системе
 - координаты X и Y
- **PolarPoint** – реализация точки с хранением координат в полярной системе
 - радиус и угол

CartesianPoint

```
public class CartesianPoint : IPoint
{
    public double X { get; set; }
    public double Y { get; set; }

    public void Add(IPoint other)
    {
        var tmp = other.ToCartesian();
        X += tmp.Item1;
        Y += tmp.Item2;
    }
}
```

```
public bool EqualTo
(IPoint other, double tolerance=0.00000001)
{
    var tmp = other.ToCartesian();
    return Math.Abs(X - tmp.Item1) < tolerance
        && Math.Abs(Y - tmp.Item2) < tolerance;
}

public Tuple<double, double> ToCartesian()
=> new Tuple<double, double>(X, Y);

public Tuple<double, double> ToPolar
(bool degreeAngle = false)
=> new Tuple<double, double>
(Math.Sqrt(X*X + Y*Y),
 Math.Atan2(Y,X)
 *(degreeAngle?Math.PI/180:1));
}
```


PolarPoint

```
public class PolarPoint : IPoint
{
    public double Radius { get; set; }
    public double RadianAngle { get; set; }

    public void Add(IPoint other)
    {
        var tmp = other.ToPolar();
        Radius = Math.Sqrt(Radius*Radius
            + tmp.Item1*tmp.Item1 +
            2*Radius*tmp.Item1
            *Math.Cos(tmp.Item2
            - RadianAngle));

        RadianAngle = RadianAngle +
            Math.Atan2(tmp.Item2
            *Math.Sin(tmp.Item2-RadianAngle),
            Radius + tmp.Item1
            *Math.Cos(tmp.Item2-RadianAngle));
    }
}
```

```
public bool EqualTo
(IPoint other, double tolerance = 1E-07)
{
    var tmp = other.ToPolar();
    return Math.Abs(Radius - tmp.Item1)
        < tolerance
        && Math.Abs(RadianAngle - tmp.Item2)
        < tolerance;
}

public Tuple<double, double> ToCartesian()
=> new Tuple<double, double>
(Radius*Math.Cos(RadianAngle),
Radius*Math.Sin(RadianAngle));

public Tuple<double, double> ToPolar
(bool degreeAngle = false)
=> new Tuple<double, double>
(Radius, RadianAngle
    * (degreeAngle ? Math.PI / 180 : 1)); }
```

Пример работы с точками

```
class Program
{
    static void Main()
    {
        CartesianPoint a = new CartesianPoint {X = 1d, Y = 2d};
        a.Add(new CartesianPoint {X = 3d, Y = 4d});
        Console.WriteLine(a.X + " " + a.Y);

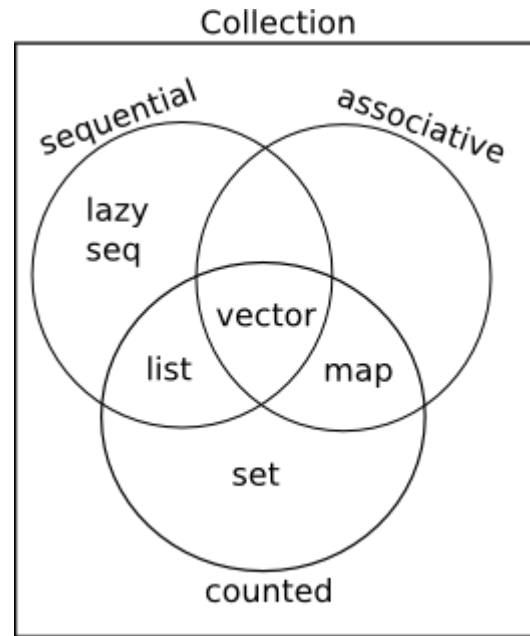
        var tmp = a.ToPolar();
        PolarPoint b = new PolarPoint
        { Radius = tmp.Item1, RadianAngle = tmp.Item2 };

        tmp = b.ToCartesian();
        Console.WriteLine(tmp.Item1 + " " + tmp.Item2);
        Console.WriteLine(a.EqualTo(b));
    }
}
```

| | |
|------|---|
| 4 | 6 |
| 4 | 6 |
| True | |

АТД специального вида

- **Коллекция** (Collection)
 - Абстрактный тип данных, соответствующий хранилищу однотипных элементов
 - Не описывающий деталей реализации



Виды коллекций

- **Линейные**
(последовательное хранение)
- **Ассоциативные**
(функция от значения элемента,
возвращающая некоторое значение)
- **Графы**
(нелинейное хранение)

Линейные коллекции

- **Список** (List)
- **Стек** (Stack)
- **Очередь** (Queue)
- **Очередь с приоритетом** (Priority queue)
- **Двусторонняя очередь** – дек
(Double-ended queue)
- **Двусторонняя очередь с приоритетом**
(Double-ended priority queue)

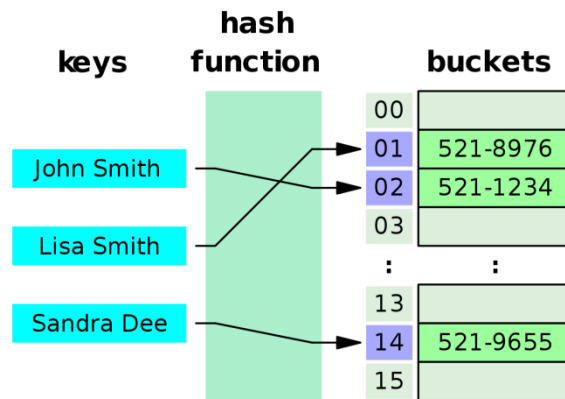
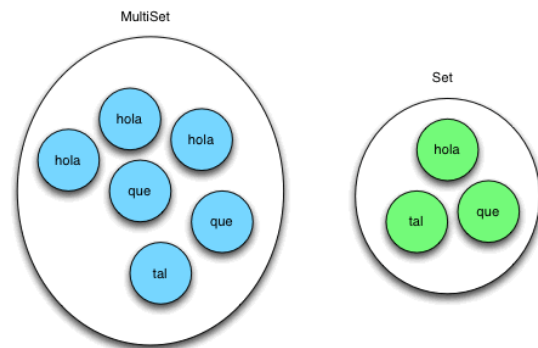


Linear Collections

Objects that store Objects in a line

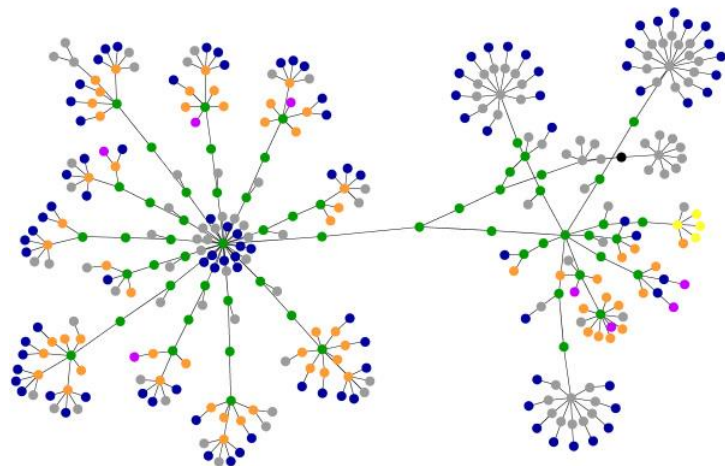
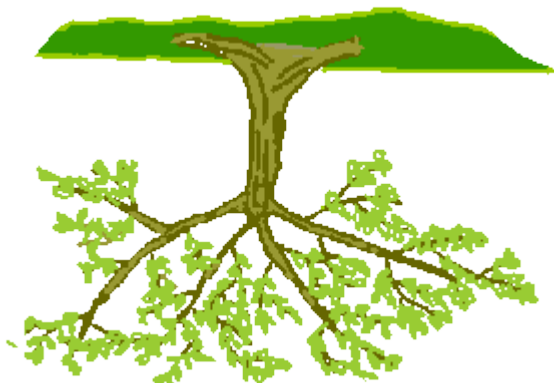
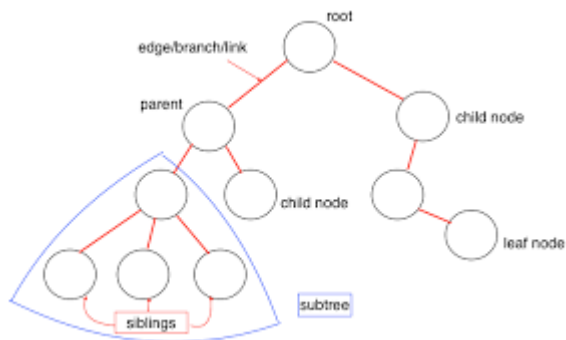
Ассоциативные коллекции

- **Множество** (Set)
- **Набор** (Bag, Multiset)
- **Словарь**,
Ассоциативный массив
(Dictionary, Map)

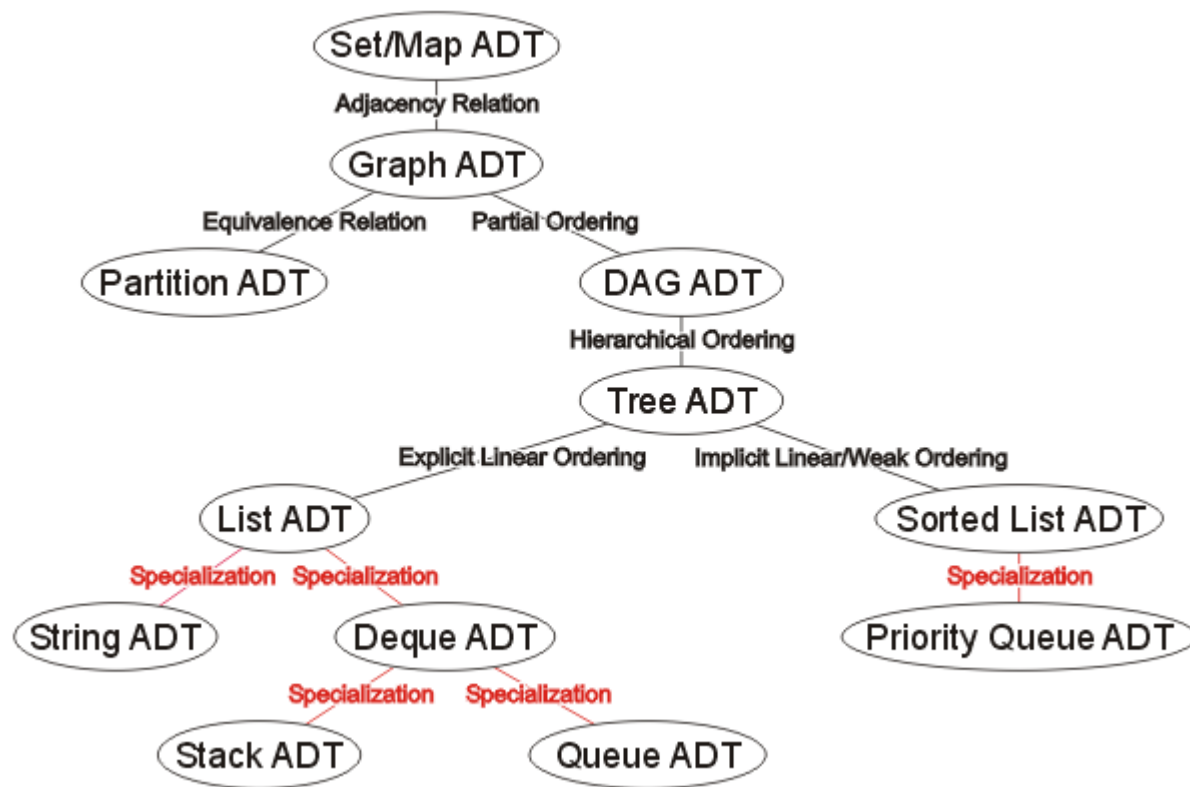


Графовые коллекции

- **Граф** (Graph)
- **Дерево** (Tree)



Отношения между коллекциями



Интерфейсы коллекций в C#

```
interface IEnumerable
```

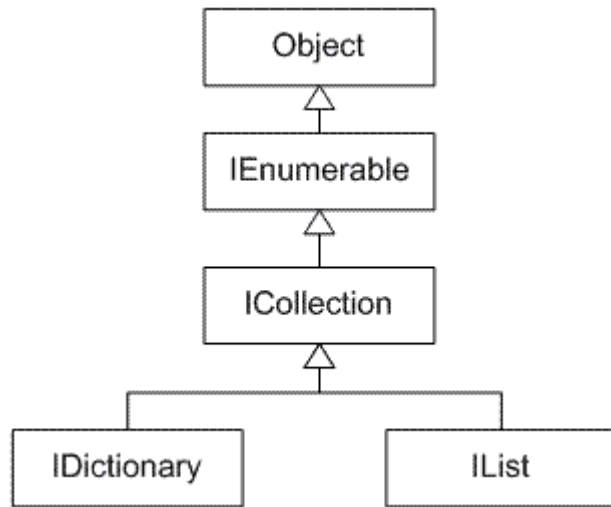
```
{  
    IEnumerator GetEnumerator();  
}
```

```
interface ICollection : IEnumerable
```

```
{  
    IEnumerator GetEnumerator();  
    void CopyTo(Array array, int index);  
    int Count { get; }  
    object SyncRoot { get; }  
    bool IsSynchronized { get; }  
}
```

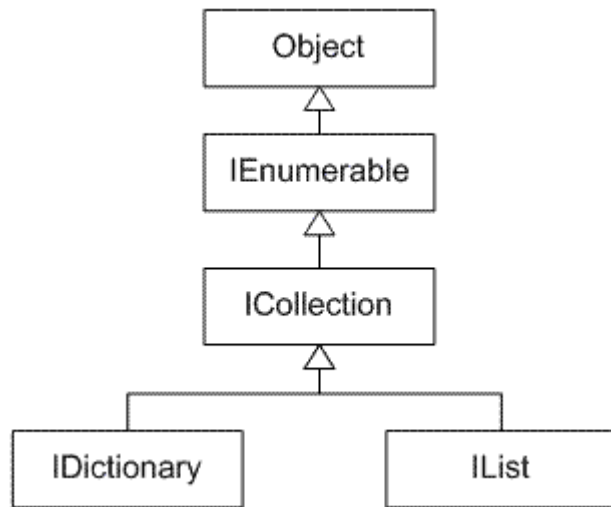
```
interface IEnumerator
```

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



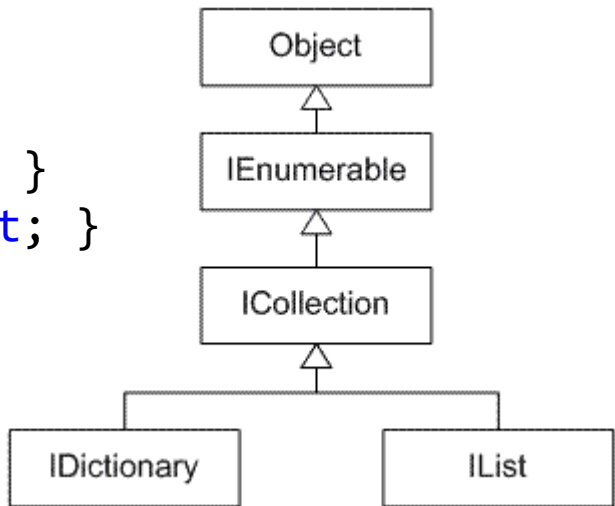
IList

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



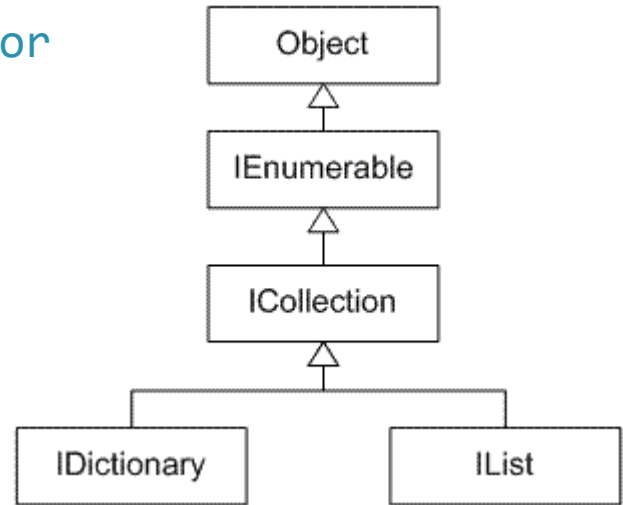
IDictionary

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

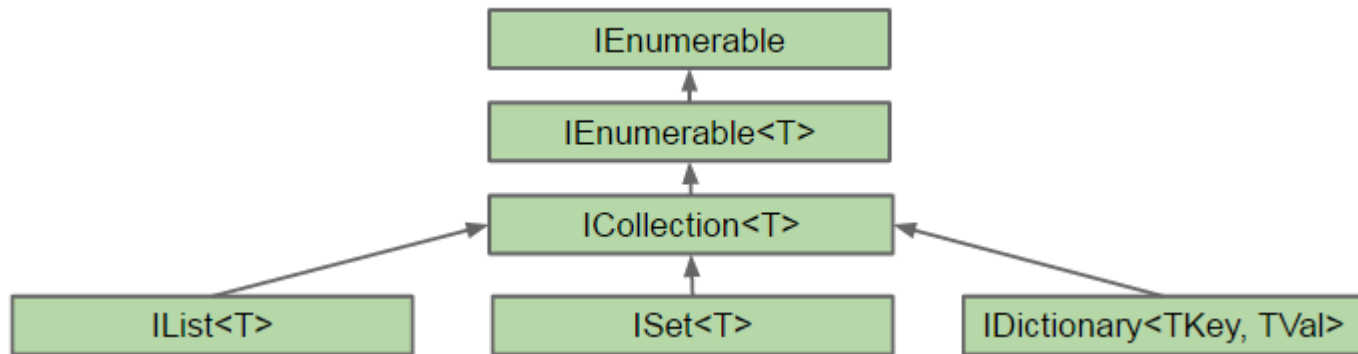


IDictionaryEnumerator

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



Обобщенные коллекции в C#



Обобщенные коллекции в C#

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

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

IList<T>

```
interface IList<T> :  
    ICollection<T>, IEnumerable<T>, IEnumerable  
{  
    int IndexOf(T value);  
    void Insert(int index, T value);  
    void RemoveAt(int index);  
    T this[int index] { get; set; }  
    bool IsFixedSize { get; }  
}
```

IDictionary<TKey,TValue>

```
interface IDictionary<TKey,TValue> :  
    ICollection<KeyValuePair<TKey, TValue>>,  
    IEnumerable<KeyValuePair<TKey, TValue>>,  
    IEnumerable  
{  
    bool Contains(TKey key);  
    void Add(TKey key, TValue value);  
    void Remove(TKey key);  
    object this[TKey key] { get; set; }  
    ICollection<TKey> Keys { get; }  
    ICollection<TValue> Values { get; }  
    bool IsFixedSize { get; }  
}
```


ISet<T>

```
interface ISet<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    void UnionWith(System.Collections.Generic.IEnumerable<T> other);
    void IntersectWith(System.Collections.Generic.IEnumerable<T> other);
    void ExceptWith(System.Collections.Generic.IEnumerable<T> other);
    void SymmetricExceptWith(System.Collections.Generic.IEnumerable<T> other);
    bool IsSubsetOf(System.Collections.Generic.IEnumerable<T> other);
    bool IsSupersetOf(System.Collections.Generic.IEnumerable<T> other);
    bool IsProperSupersetOf(System.Collections.Generic.IEnumerable<T> other);
    bool IsProperSubsetOf(System.Collections.Generic.IEnumerable<T> other);
    bool Overlaps(System.Collections.Generic.IEnumerable<T> other);
    bool SetEquals(System.Collections.Generic.IEnumerable<T> other);
}
```

Реализации коллекций

- Коллекции – абстрактные типы данных
- Коллекции описывают контейнеры элементов
- В ОО ЯП представляются интерфейсами
- Реализации - классы

Структуры данных

- Конкретные реализации одной или нескольких коллекций (АТД)
- У одной коллекции может быть несколько реализующих её структур данных

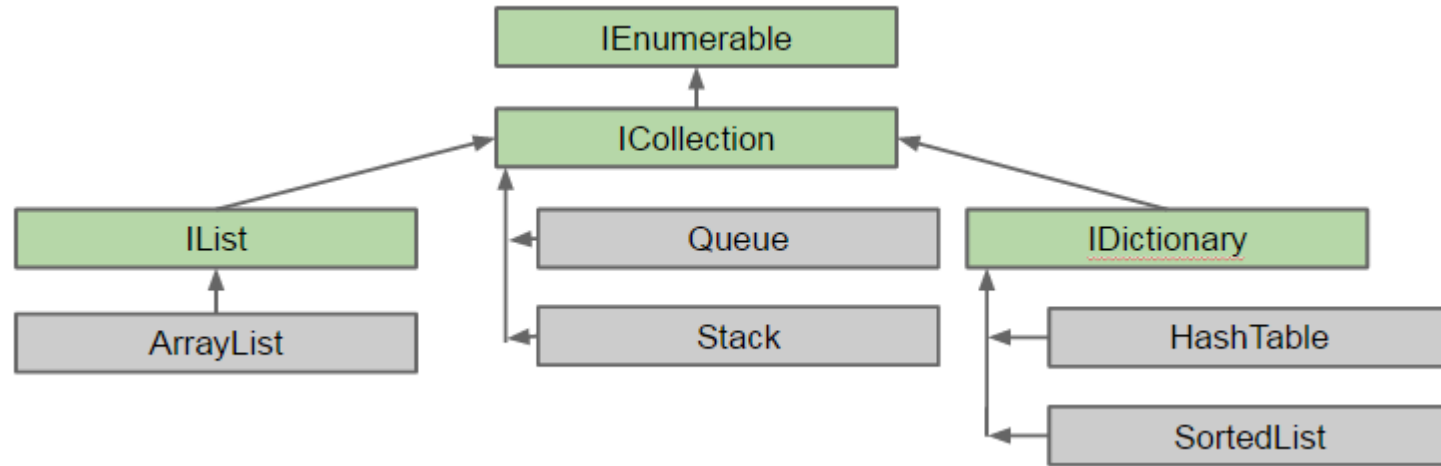
Структуры данных : операции

| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|---------------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|---------------------|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| <u>Array</u> | $\theta(1)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ |
| <u>Stack</u> | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(1)$ | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(1)$ | $\theta(n)$ |
| <u>Queue</u> | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(1)$ | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(1)$ | $\theta(n)$ |
| <u>Singly-Linked List</u> | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(1)$ | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(1)$ | $\theta(n)$ |
| <u>Doubly-Linked List</u> | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(1)$ | $\theta(n)$ | $\theta(n)$ | $\theta(1)$ | $\theta(1)$ | $\theta(n)$ |
| <u>Skip List</u> | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n \log(n))$ |
| <u>Hash Table</u> | N/A | $\theta(1)$ | $\theta(1)$ | $\theta(1)$ | N/A | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ |
| <u>Binary Search Tree</u> | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ |
| <u>Cartesian Tree</u> | N/A | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | N/A | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ |
| <u>B-Tree</u> | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(n)$ |
| <u>Red-Black Tree</u> | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(n)$ |
| <u>Splay Tree</u> | N/A | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | N/A | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(n)$ |
| <u>AVL Tree</u> | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(n)$ |
| <u>KD Tree</u> | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(\log(n))$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $\theta(n)$ |

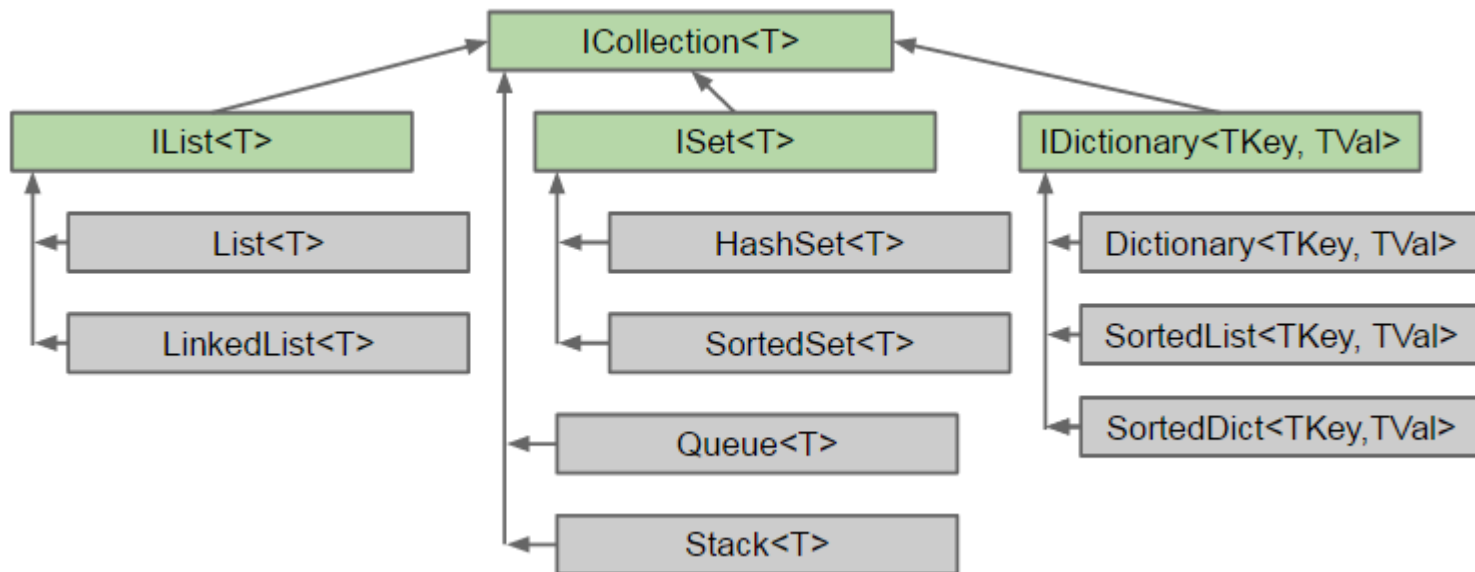
Стандартные структуры данных в C#

- Обобщённые и необобщённые (универсальные, неуниверсальные)
- Реализуют интерфейсы ICollection, ICollection<T>

Необобщенные коллекции



Обобщенные коллекции



Списки на массивах

- Классы:
 - `System.Collections.ArrayList`
 - `System.Collections.Generic.List<T>`
- Временная сложность операций:
 - **Добавление элемента:** $O(1)$, **Вставка элемента:** $O(n)$
 - **Поиск элемента:** $O(n)$
 - **Доступ по индексу:** $O(1)$
 - **Сортировка:** $O(n \log n)$
 - **Удаление элемента:** $O(n)$

Стеки и очереди на массивах

- Классы:
 - `System.Collections.Stack`, `System.Collections.Queue`
 - `System.Collections.Generic.Stack<T>`, `System.Collections.Generic.Queue<T>`
- Сложность операций:
 - **Добавление элемента** (Push, Enqueue): $O(1)$
 - **Добавление при превышении ёмкости** массива: $O(n)$
 - **Извлечение элемента** (Pop, Dequeue): $O(1)$
 - **Поиск элемента**: $O(n)$

Словари на хеш-таблицах

- Классы:
 - `System.Collections.Hashtable`
 - `System.Collections.Generic.Dictionary<TKey,TValue>`
- Сложность операций:
 - **Добавление элемента:** $O(1)$, $O(n)$ при коллизии
 - **Добавление при превышении ёмкости:** $O(n)$
 - **Удаление элемента:** $O(1)$, $O(n)$ при коллизии
 - **Доступ по индексу:** $O(1)$, $O(n)$ при коллизии

Упорядоченные списки на массивах

- Классы:
 - `System.Collections.SortedList`
 - `System.Collections.Generic.SortedList<TKey,TValue>`
- Сложность операций:
 - **Вставка/добавление:** $O(n)$, $O(\log n)$ при добавлении в конец списка
 - **Добавление при превышении ёмкости:** $O(n)$
 - **Удаление:** $O(n)$
 - **Доступ по индексу:** $O(1)$
 - **Поиск элемента:** $O(\log n)$

Обобщенное множество на хеш-таблицах

- Класс:
 - `System.Collections.Generic.HashSet<T>`
- Сложность операций:
 - **Добавление элемента:** $O(1)$, $O(n)$ при коллизии
 - **Добавление при превышении ёмкости:** $O(n)$
 - **Удаление элемента:** $O(1)$, $O(n)$ при коллизии
 - **Доступ по ключу/поиск:** $O(1)$, $O(n)$ при коллизии

Упорядоченные словари и множества на красно-черных деревьях

- Классы:
 - `System.Collections.Generic.SortedDictionary<TKey,TValue>`
 - `System.Collections.Generic.SortedSet<T>`
- Сложность операций:
 - **Добавление элемента:** $O(\log n)$
 - **Удаление элемента:** $O(\log n)$
 - **Доступ по ключу/поиск:** $O(\log n)$
 - **Получение итератора:** $O(\log n)$

Сводная таблица по дженерикам

| | Internal Implement-ation | Add/insert | Add beyond capacity | Queue/Push | Dequeue/Pop/Peak | Remove/RemoveAt | Item[index]/ElementAt(index) | GetEnumerator | Contains(value)/IndexOf/ContainsValue/Find |
|------------------|---|--|---------------------|------------|------------------|----------------------------------|------------------------------|---------------|--|
| List | Array | $O(1)$ to add, $O(n)$ to insert | $O(n)$ | - | - | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| LinkedList | Doubly linked list | $O(1)$, before/after given node | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$, before/after given node | $O(n)$ | $O(1)$ | $O(n)$ |
| Stack | Array | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | - | - | $O(1)$ | $O(n)$ |
| Queue | Array | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | - | - | $O(1)$ | $O(n)$ |
| Dictionary | Hashtable with links to another array index for collision | $O(1)$, $O(n)$ if collision | $O(n)$ | - | - | $O(1)$, $O(n)$ if collision | $O(1)$, $O(n)$ if collision | $O(1)$ | $O(n)$ |
| HashSet | Hashtable with links to another array index for collision | $O(1)$, $O(n)$ if collision | $O(n)$ | - | - | $O(1)$, $O(n)$ if collision | $O(1)$, $O(n)$ if collision | $O(1)$ | - |
| SortedDictionary | Red-black tree | $O(\log n)$ | $O(\log n)$ | - | - | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(n)$ |
| SortedList | Array | $O(n)$, $O(\log n)$ if added to end of list | $O(n)$ | - | - | $O(n)$ | $O(\log n)$ | $O(1)$ | $O(n)$ |
| SortedSet | Red-black tree | $O(\log n)$ | $O(\log n)$ | - | - | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | - |

$O(1)$

$O(\log n)$

Вопросы для самоконтроля

- Чем абстрактный тип данных отличается от конкретного?
- Как в C# представляются коллекции?
- Какие отношения между коллекциями и структурами данных?
- Чем отличаются обобщенные коллекции от необобщенных?

Вопросы для самоконтроля

- Какую структуру данных выбрать для доступа к элементам по индексу?
- Какую структуру данных выбрать для доступа к элементам по ключу?
- Какую структуру данных выбрать для проверок наличия элемента?
- Какую структуру данных выбрать для LIFO обработки?
- Какую структуру данных выбрать для FIFO обработки?



Вопросы?

e-mail: marchenko@it.kfu.ru