

Коллекции



Коллекция

представляет собой совокупность объектов.

в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связанных списков, стеков, очередей и хеш-таблиц.

Типы коллекций

▶ необобщенные

- ▶ наличие разнотипных данных
- ▶ ссылки на данные типа object
(не обеспечивают типовую безопасность)
- ▶ **System.Collections**

коллекции, в которых элемент коллекции представлен как object (слаботипизированные коллекции)

▶ обобщенные

- ▶ обеспечивают типовую безопасность
- ▶ **System.Collections.Generic**

▶ специальные

- ▶ **System.Collections.Specialized**

▶ с поразрядной организацией

- ▶ BitArray

▶ параллельные

- ▶ многопоточный доступ к коллекции
- ▶ **System.Collections.Concurrent**

Каждый класс коллекции оптимизирован под конкретную форму хранения данных и доступа к ним,

и каждый из них предоставляет специализированные методы

Интерфейсы, используемые в коллекциях C#

► **IEnumerable<T>**

- для foreach
- GetEnumerator()

перечислитель, с помощью которого становится возможен последовательный перебор коллекции

► **IEnumerator<>**

позволяет перебирать элементы коллекции

► **ICollection<T>**

- Count
- CopyTo()
- Add(), Remove(), Clear()

► **IList<T>**

- Индексатор
- Insert()
- Remove()

позволяет получать элементы коллекции по порядку

► ISet<T>

► IDictionary<TKey, TValue>

► IComparer<T>

сравнения двух объектов

► ICollection

- определяет элементы

Классы необобщенных коллекций

▶ ArrayList - IList, ICollection, IEnumerable, ICloneable

▶ BitArray - Определяет динамический массив ICollection, IEnumerable, ICloneable

▶ Hashtable Определяет хеш-таблицу для пар "ключ-значение"

▶ Queue Определяет очередь

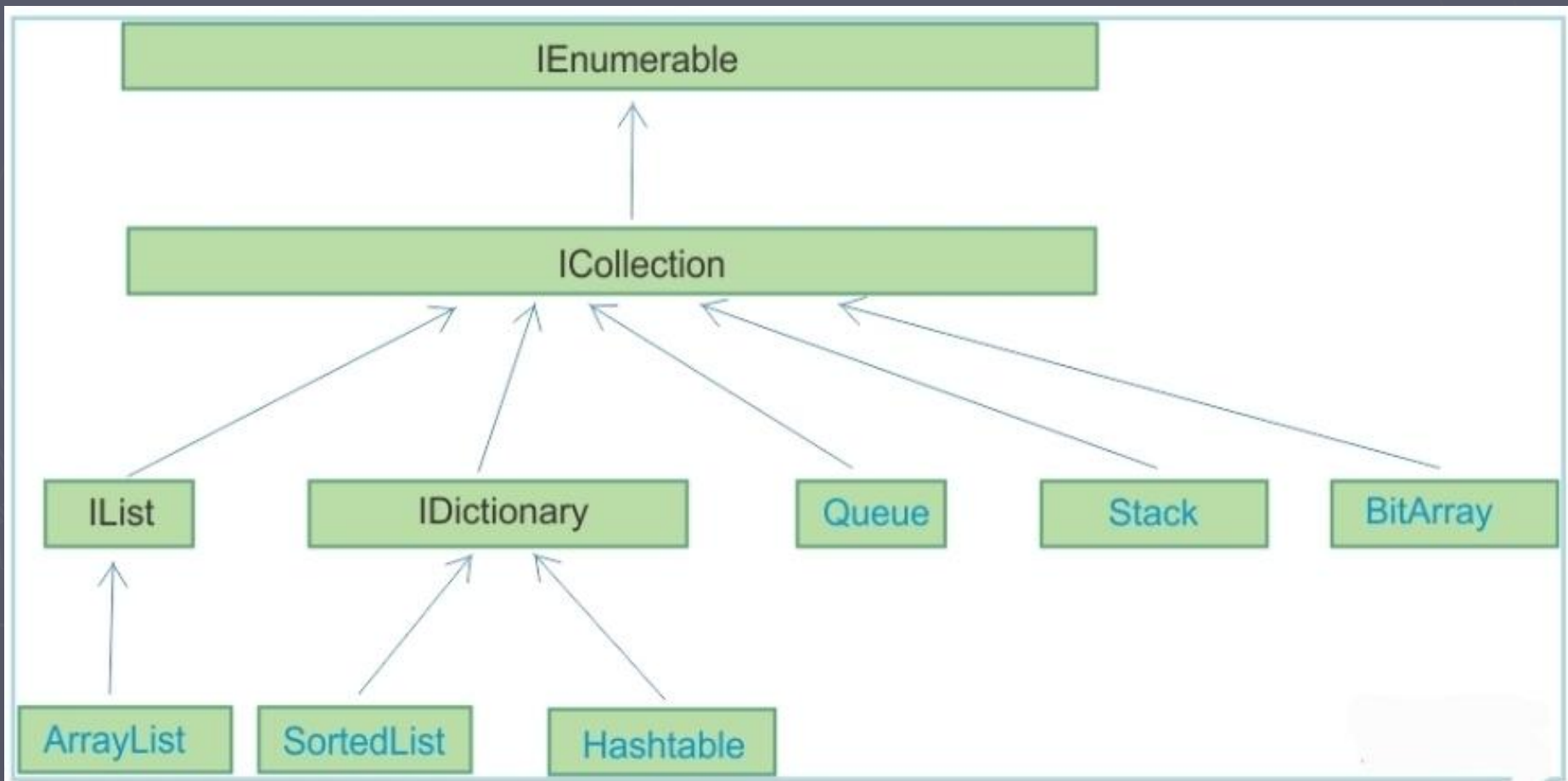
▶ SortedList - класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу

▶ Stack Определяет стек

1) хранят ссылки на объекты
при сохранении или извлечении элементов требуется приведение типов (исключение BitArray)

2) включены в библиотеку с целью обратной совместимости с существующими приложениями
применять не рекомендуется

3) В UWP (universal windows platform) эти классы недоступны



Класс ArrayList

определяется массив переменной длины, который состоит из ссылок на объекты и может динамически увеличивать и уменьшать свой размер

```
ArrayList arr2 = new ArrayList(1000); // 1000  
ArrayList arr3 = new ArrayList();
```

- ▶ Свойства – Capacity позволяет получать и устанавливать емкость вызывающей коллекции
- ▶ Count, Item

int Add(object value): добавляет в список объект value

void Clear(): удаляет из списка все элементы

bool Contains(object value): проверяет, содержится ли в списке объект value. Если содержится, возвращает true, иначе возвращает false

void CopyTo(Array array): копирует текущий список в массив array.

ArrayList GetRange(int index, int count): возвращает новый список ArrayList, который содержит count элементов текущего списка, начиная с индекса index

void Insert(int index, object value): вставляет в список по индексу index объект value

AddRange()

Добавляет элементы из коллекции в конец вызывающей коллекции типа ArrayList

BinarySearch()

Выполняет поиск в вызывающей коллекции значения. Возвращает индекс найденного элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован

CopyTo()

Копирует содержимое вызывающей коллекции в массив, который должен быть одномерным и совместимым по типу с элементами коллекции

FixedSize()

Заключает коллекцию в оболочку типа ArrayList с фиксированным размером и возвращает результат.

IndexOf()

Возвращает индекс первого вхождения объекта в вызывающей коллекции. Если искомый объект не обнаружен, возвращает значение -1

RemoveRange()

Удаляет часть вызывающей коллекции, начиная с элемента, указываемого по индексу `index`, и включая количество элементов, определяемое параметром `count`

Sort()

Сортирует вызывающую коллекцию по нарастающей

```
ArrayList list = new ArrayList(); // ArrayList list = new ArrayList(){1, 2, 5, "string", 7.7};
list.Add(2.3); // заносим в список объект типа double
list.Add(55); // заносим в список объект типа int
list.AddRange(new string[] { "Hello", "world" }); // заносим в список строковый массив

// перебор значений
foreach (object o in list)
{
    Console.WriteLine(o);
}

// удаляем первый элемент
list.RemoveAt(0);

// переворачиваем список
list.Reverse();

// получение элемента по индексу
Console.WriteLine(list[0]);

// перебор значений
for (int i = 0; i < list.Count; i++)
{
    Console.WriteLine(list[i]);
}
```

Обобщенные коллекции

- ▶ Dictionary <Tkey, TValue>
- ▶ LinkedList<T>
- ▶ List<T>
- ▶ Queue<T>
- ▶ SortedDictionary<Tkey, Tvalue>
- ▶ SortedList<T>
- ▶ HashSet<T> и SortedSet<T>
- ▶ Stack<T>

преимущества: повышение производительности (не надо тратить время на упаковку и распаковку объекта) и повышенная типобезопасность.

Классы обобщенных коллекций

System.Collections.Generic

Тип коллекции	Особенности
Dictionary <Tkey, TValue>	Идентификация и извлечение с помощью ключей, не сортирован
LinkedList<T>	Двусторонний упорядоченный список, оптимизация - вставка и удаление с любого конца, поддерживает произвольный доступ
List<T>	доступ по индексу, поиск и сортировка
Queue<T> и Stack<T>	
SortedList<T>	Отсортированный список пар «ключ–значение», ключи должны реализовывать IComparable<T>, не дублируется
SortedDictionary<Tkey, TValue>	Вставка медленнее, извлечение быстрее, использует больше памяти чем SortedList
HashSet<T>	Неупорядоченный набор значений, оптимизация - быстрое извлечение данных, объединений и пересечений наборов.

Stack<T>

представляет коллекцию, которая использует алгоритм LIFO ("последний вошел - первый вышел").

Clear: очищает стек

Contains: проверяет наличие в стеке элемента и возвращает true при его наличии

Push: добавляет элемент в стек в верхушку стека

Pop: извлекает и возвращает первый элемент из стека

Peek: просто возвращает первый элемент из стека без его удаления

```
Stack<int> numbs = new Stack<int>();
```

```
numbs.Push(3); // в стеке 3
```

```
numbs.Push(5); // в стеке 5, 3
```

```
int stackElement = numbs.Pop();
```

```
Stack<Point> figure = new Stack<Point>();
```

```
figure.Push(new Point() );
```

```
foreach (Point p in figure)
```

```
{
```

```
    Console.WriteLine(p.x);
```

```
}
```

```
var people = new Stack<string>();
people.Push("Tom");
// people = { Tom }
people.Push("Sam");
// people = { Sam, Tom }
people.Push("Bob");
// people = { Bob, Sam, Tom }

// получаем первый элемент стека без его удаления
string headPerson = people.Peek();
Console.WriteLine(headPerson); // Bob

string person1 = people.Pop();
// people = { Sam, Tom }
Console.WriteLine(person1); // Bob

string person2 = people.Pop();
// people = { Tom }
Console.WriteLine(person2); // Sam

string person3 = people.Pop();
// people = { }
Console.WriteLine(person3); // Tom
```

Queue<T>

представляет обычную очередь, которая работает по алгоритму FIFO ("первый вошел - первый вышел").

void Clear(): очищает очередь

bool Contains(T item): возвращает true, если элемент item имеется в очереди

T Dequeue(): извлекает и возвращает первый элемент очереди

void Enqueue(T item): добавляет элемент в конец очереди

T Peek(): просто возвращает первый элемент из начала очереди без его удаления

```
var people = new Queue<string>();
```

```
// добавляем элементы
```

```
people.Enqueue("Tom"); // people = { Tom }
```

```
people.Enqueue("Bob"); // people = { Tom, Bob }
```

```
people.Enqueue("Sam"); // people = { Tom, Bob, Sam }
```

```
// получаем элемент из самого начала очереди
```

```
var firstPerson = people.Peek();
```

```
Console.WriteLine(firstPerson); // Tom
```

```
// удаляем элементы
```

```
var person1 = people.Dequeue(); // people = { Bob, Sam }
```

```
Console.WriteLine(person1); // Tom
```

```
var person2 = people.Dequeue(); // people = { Sam }
```

```
Console.WriteLine(person2); // Bob
```

```
var person3 = people.Dequeue(); // people = { }
```

```
Console.WriteLine(person3); // Sam
```

```
Queue<int> numbers = new Queue<int>();  
numbers.Enqueue(3);  
int queueElement = numbers.Dequeue();
```

```
Queue<Point> points = new Queue<Point>();  
points.Enqueue(new Point());  
Point pp = points.Peek();  
Console.WriteLine(pp.x);
```

```
class Person
{
    public string Name { get; }
    public Person(string name) => Name = name;
}

class Doctor
{
    public void TakePatients(Queue<Person> patients)
    {
        while(patients.Count > 0)
        {
            var patient = patients.Dequeue();
            Console.WriteLine($"Осмотр пациента {patient.Name}");
        }
        Console.WriteLine("Доктор закончил осматривать пациентов");
    }
}

var patients = new Queue<Person>();
patients.Enqueue(new Person("Tom"));
patients.Enqueue(new Person("Bob"));
patients.Enqueue(new Person("Sam"));

var practitioner = new Doctor();
practitioner.TakePatients(patients);
```

Осмотр пациента Tom

Осмотр пациента Bob

Осмотр пациента Sam

Доктор закончил осматривать пацие

LinkedList<T>

представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент.

Count: количество элементов в связанном списке

First: первый узел в списке

Last: последний узел в списке

AddFirst(T value): вставляет новый узел со значением value в начало списка

AddLast(T value): вставляет новый узел со значением value в конец списка

RemoveFirst(): удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным

RemoveLast(): удаляет последний узел из списка

```
var employees = new List<string> { "Tom", "Sam", "Bob" };

LinkedList<string> people = new LinkedList<string>(employees);
Console.WriteLine(people.Count);           // 3
Console.WriteLine(people.First?.Value);    // Tom
Console.WriteLine(people.Last?.Value);     // Bob
```

```
var people = new LinkedList<string>();
people.AddLast("Tom"); // вставляем узел со значением Tom на последнее место
                        //так как в списке нет узлов, то последнее будет также и первым
people.AddFirst("Bob"); // вставляем узел со значением Bob на первое место

// вставляем после первого узла новый узел со значением Mike
if (people.First != null) people.AddAfter(people.First, "Mike");

// теперь у нас список имеет следующую последовательность: Bob Mike Tom
foreach (var person in people) Console.WriteLine(person);
```

Dictionary<T, R>

Словарь хранит объекты, которые представляют **пару ключ-значение**.

Класс словаря Dictionary<K, V> типизируется двумя типами:
параметр K представляет тип ключей
параметр V предоставляет тип значений.

```
Dictionary<int, string> people = new Dictionary<int, string>();
```

```
var people = new Dictionary<int, string>()  
{  
    { 5, "Tom"},  
    { 3, "Sam"},  
    { 11, "Bob"}  
};
```

Dictionary<T, R>

каждый элемент в словаре представляет структуру `KeyValuePair<TKey, TValue>`,
где `TKey` представляет тип ключа,
`TValue` - тип значений

```
var mike = new KeyValuePair<int, string>(56, "Mike");  
var employees = new List<KeyValuePair<int, string>>() { mike };  
var people = new Dictionary<int, string>(employees)  
{  
    [5] = "Tom",  
    [6] = "Sam",  
    [7] = "Bob",  
};
```

key: 5 value: Tom
key: 6 value: Sam
key: 7 value: Bob

```
foreach(var person in people)  
{  
    Console.WriteLine($"key: {person.Key} value: {person.Value}");  
}
```

Dictionary<T, R>

Для обращения к элементам из словаря применяется их ключ, который передается в квадратных скобках:

словарь[ключ]

```
var people = new Dictionary<int, string>()
{
    [5] = "Tom",
    [6] = "Sam",
    [7] = "Bob",
};

// получаем элемент по ключу 6
string sam = people[6]; // Sam
Console.WriteLine(sam); // Sam

// переустанавливаем значение по ключу 6
people[6] = "Mike";
Console.WriteLine(people[6]); // Mike

// добавляем новый элемент по ключу 22
people[22] = "Eugene";
Console.WriteLine(people[22]); // Eugene
```

Dictionary<T, R>

void Add(K key, V value): добавляет новый элемент в словарь

void Clear(): очищает словарь

bool ContainsKey(K key): проверяет наличие элемента с определенным ключом и возвращает true при его наличии в словаре

bool ContainsValue(V value): проверяет наличие элемента с определенным значением и возвращает true при его наличии в словаре

bool Remove(K key): удаляет по ключу элемент из словаря

```
// условная телефонная книга
var phoneBook = new Dictionary<string, string>();

// добавляем элемент: ключ - номер телефона, значение - имя абонента
phoneBook.Add( "+123456", "Tom" );
// альтернативное добавление
// phoneBook[ "+123456" ] = "Tom";

// Проверка наличия
var phoneExists1 = phoneBook.ContainsKey( "+123456" );           // true
Console.WriteLine( $" +123456: {phoneExists1} " );
var phoneExists2 = phoneBook.ContainsKey( "+567456" );           // false
Console.WriteLine( $" +567456: {phoneExists2} " );
var abonentExists1 = phoneBook.ContainsValue( "Tom" );           // true
Console.WriteLine( $" Tom: {abonentExists1} " );
var abonentExists2 = phoneBook.ContainsValue( "Bob" );           // false
Console.WriteLine( $" Bob: {abonentExists2} " );

// удаление элемента
phoneBook.Remove( "+123456" );

// проверяем количество элементов после удаления
Console.WriteLine( $" Count: {phoneBook.Count} " ); // Count: 0
```

```
Dictionary<string, int> student = new Dictionary<string, int>();
```

```
student.Add("Анна", 8);  
student.Add("Никита", 3);  
student["Алексей"] = 1;  
student["Елена"] = 3;  
student.Remove("Никита");
```

```
Console.WriteLine("The Dictionary contains:");  
foreach (KeyValuePair<string, int> element in student)  
{  
    Console.WriteLine($"Name: { element.Key},  
                        Age: {element.Value}");  
}
```

```
The Dictionary contains:  
Name: Анна, Age: 8  
Name: Алексей, Age: 1  
Name: Елена, Age: 3
```

Инициализация словаря

```
Dictionary<string, string> fit =  
    new Dictionary<string, string>  
    {  
        [ "ИСИТ" ] = "Понедельник",  
        [ "ДЭВИ" ] = "Вторник",  
        [ "ПОИТ" ] = "Среда",  
        [ "ПОБМС" ] = "Четверг"  
    };
```

System.Collections.Specialized

- ▶ **CollectionsUtil**
- ▶ **HybridDictionary**
- ▶ **ListDictionary**
- ▶ **NameValueCollection**
- ▶ **OrderedDictionary**
- ▶ **StringCollection**
- ▶ **StringDictionary**

CollectionsUtil	Содержит фабричные методы для создания коллекций
HybridDictionary	Предназначен для коллекций, в которых для хранения небольшого количества пар "ключ-значение" используется класс ListDictionary. При превышении коллекцией определенного размера автоматически используется класс Hashtable для хранения ее элементов
ListDictionary	Предназначен для коллекций, в которых для хранения пар "ключ-значение" используется связный список. Такие коллекции рекомендуются только для хранения небольшого количества элементов
NameValueCollection	Предназначен для отсортированных коллекций, в которых хранятся пары "ключ-значение", причем и ключ, и значение относятся к типу string
OrderedDictionary	Предназначен для коллекций, в которых хранятся индексируемые пары "ключ-значение"
StringCollection	Предназначен для коллекций, оптимизированных для хранения символьных строк
StringDictionary	Предназначен для хеш-таблиц, в которых хранятся пары "ключ-значение", причем и ключ, и значение относятся к типу string

Битовые коллекции

► Класс BitArray

System.Collections.Specialized

- Изменяемый размер
- ICollection, IEnumerable, ICloneable

*And()
Get
Not
Or
Xor
Set*

► Структура BitVector32

- 32 бита (целое) - хранение – стек тип-значение выше скорость работы
- Если необходимое количество бит известно заранее

каждый элемент массива values становится отдельным битом в коллекции

```
byte[] d = { 12, 100 };
```

```
BitArray bits = new BitArray(d);
```

bits [0] обозначает первые 8 битов, элемент bits [1] — вторые 8 битов и т.д.

And() Выполняет операцию логического умножения (И) битов вызывающего объекта и коллекции value. Возвращает коллекцию типа BitArray, содержащую результат

Get() Возвращает значение бита, указываемого по индексу

Not() Выполняет операцию поразрядного логического отрицания (НЕ) битов вызывающей коллекции и возвращает коллекцию типа BitArray, содержащую результат

Or()

Выполняет операцию логического сложения (ИЛИ) битов вызывающего объекта и коллекции value. Возвращает коллекцию типа BitArray, содержащую результат

Set()

Устанавливает бит, указываемый по индексу `index`, равным значению `value`

SetAll()

Устанавливает все биты равными значению `value`

Xor()

Выполняет логическую операцию исключающее (ИЛИ) над битами вызывающего объекта и коллекции `value`. Возвращает коллекцию типа `BitArray`, содержащую результат.

Свойство **Length** позволяет установить или получить количество битов в коллекции.

```
byte[] d = { 12, 100 };  
    BitArray bits = new BitArray(d);  
    bits.SetAll(false);  
    bits.Set(2, true);  
    bits[2] = true;  
    bits[8] = true;  
    foreach (bool b in bits)  
        Console.Write(b ? 1 : 0);  
    Console.WriteLine("\n");
```



```
0010000010000000
```

Наблюдаемые коллекции

System.Collections.ObjectModel

► ObservableCollection<T>

- пользовательский интерфейс получает информацию об изменениях коллекции
- унаследован от Collection<T>, использует внутри себя List<T>, INotifyCollectionChanged

```
var obsev = new ObservableCollection<int>();
            obsev.CollectionChanged += CollectionChanged;
            obsev.Add(23);
            obsev.Add(675);
            obsev.Insert(1,78);
        }
        private static void CollectionChanged(object sender,
System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
        {}
```

Создание и инициализация ObservableCollection

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

типизируется типом string, поэтому может хранить только строки.

```
ObservableCollection<string> people = new ObservableCollection<string>();
```

позволяет передать в ObservableCollection объекты из другой коллекции или массива:

```
var people = new ObservableCollection<string>( new string[] { "Tom", "Bob", "Sam" });
```

Методы ObservableCollection

void Insert(int index, T item): вставляет элемент item в коллекцию по индексу index. Если такого индекса в коллекции нет, то генерируется исключение

bool Remove(T item): удаляет элемент item из коллекции, и если удаление прошло успешно, то возвращает true. Если в коллекции несколько одинаковых элементов, то удаляется только первый из них

void RemoveAt(int index): удаление элемента по указанному индексу index. Если такого индекса в коллекции нет, то генерируется исключение

void Move(int oldIndex, int newIndex): перемещает элемент с индекса oldIndex на позицию по индексу newIndex

Методы ObservableCollection

void Add(T item): добавление нового элемента в коллекцию

void CopyTo(T[] array, int index,): копирует в массив array элементы из коллекции начиная с индекса index

bool Contains(T item): возвращает true, если элемент item есть в коллекции

void Clear(): удаляет из коллекции все элементы

int IndexOf(T item): возвращает индекс первого вхождения элемента в коллекции

```
using System.Collections.ObjectModel;

var people = new ObservableCollection<string>();

// добавляем элемент
people.Add( "Bob" );
// вставляем элемент по индексу 0
people.Insert(0, "Tom" );

// проверка наличия элемента
bool bobExists = people.Contains( "Bob" );      // true
Console.WriteLine( $"Bob exists: {bobExists}" );
bool mikeExists = people.Contains( "Mike" );    // false
Console.WriteLine( $"Mike exists: {mikeExists}" );

// удаляем элемент
people.Remove( "Tom" );
// удаляем элемент по индексу 0
people.RemoveAt(0);
```

Уведомление об изменении коллекции

```
void NotifyCollectionChangedEventHandler(object?  
sender, NotifyCollectionChangedEventArgs e);
```

событие **CollectionChanged**, подписавшись на которое, мы можем обработать любые изменения коллекции.

событие представляет делегат
NotifyCollectionChangedEventHandler

Параметр **NotifyCollectionChangedEventArgs**
хранит всю информацию о событии.

его свойство **Action** позволяет узнать характер
изменений

`NotifyCollectionChangedAction.Add`: добавление

`NotifyCollectionChangedAction.Remove`: удаление

`NotifyCollectionChangedAction.Replace`: замена

`NotifyCollectionChangedAction.Move`: перемещение
объекта внутри коллекции на новую позицию

`NotifyCollectionChangedAction.Reset`: сброс
содержимого коллекции (например, при ее очистке
с помощью метода `Clear()`)

```
class Person
{
    public string Name { get; }
    public Person(string name) => Name = name;
}
```

// обработчик изменения коллекции

```
void People_CollectionChanged(object? sender, NotifyCollectionChangedEventArgs e)
{
    switch (e.Action)
    {
        case NotifyCollectionChangedAction.Add: // если добавление
            if(e.NewItems?[0] is Person newPerson)
                Console.WriteLine($"Добавлен новый объект: {newPerson.Name}");
            break;
        case NotifyCollectionChangedAction.Remove: // если удаление
            if (e.OldItems?[0] is Person oldPerson)
                Console.WriteLine($"Удален объект: {oldPerson.Name}");
            break;
        case NotifyCollectionChangedAction.Replace: // если замена
            if ((e.NewItems?[0] is Person replacingPerson) &&
                (e.OldItems?[0] is Person replacedPerson))
                Console.WriteLine($"Объект {replacedPerson.Name} заменен объектом {replacingPerson.Name}");
            break;
    }
}
```

```
using System.Collections.ObjectModel;
using System.Collections.Specialized;

var people = new ObservableCollection<Person>()
{
    new Person("Tom"),
    new Person("Sam")
};
// подписываемся на событие изменения коллекции
people.CollectionChanged += People_CollectionChanged;

people.Add(new Person("Bob")); // добавляем новый элемент

people.RemoveAt(1);           // удаляем элемент
people[0] = new Person("Eugene"); // заменяем элемент

Console.WriteLine("\nСписок пользователей:");
foreach (var person in people)
{
    Console.WriteLine(person.Name);
}
```

Добавлен новый объект: Bob
Удален объект: Sam
Объект Tom заменен объектом Eugene

Список пользователей:
Eugene
Bob

Параллельные коллекции

System.Collections.Concurrent

коллекции классов, предназначенные для безопасной работы в многопоточной среде, которыми можно воспользоваться при создании многопоточных приложений

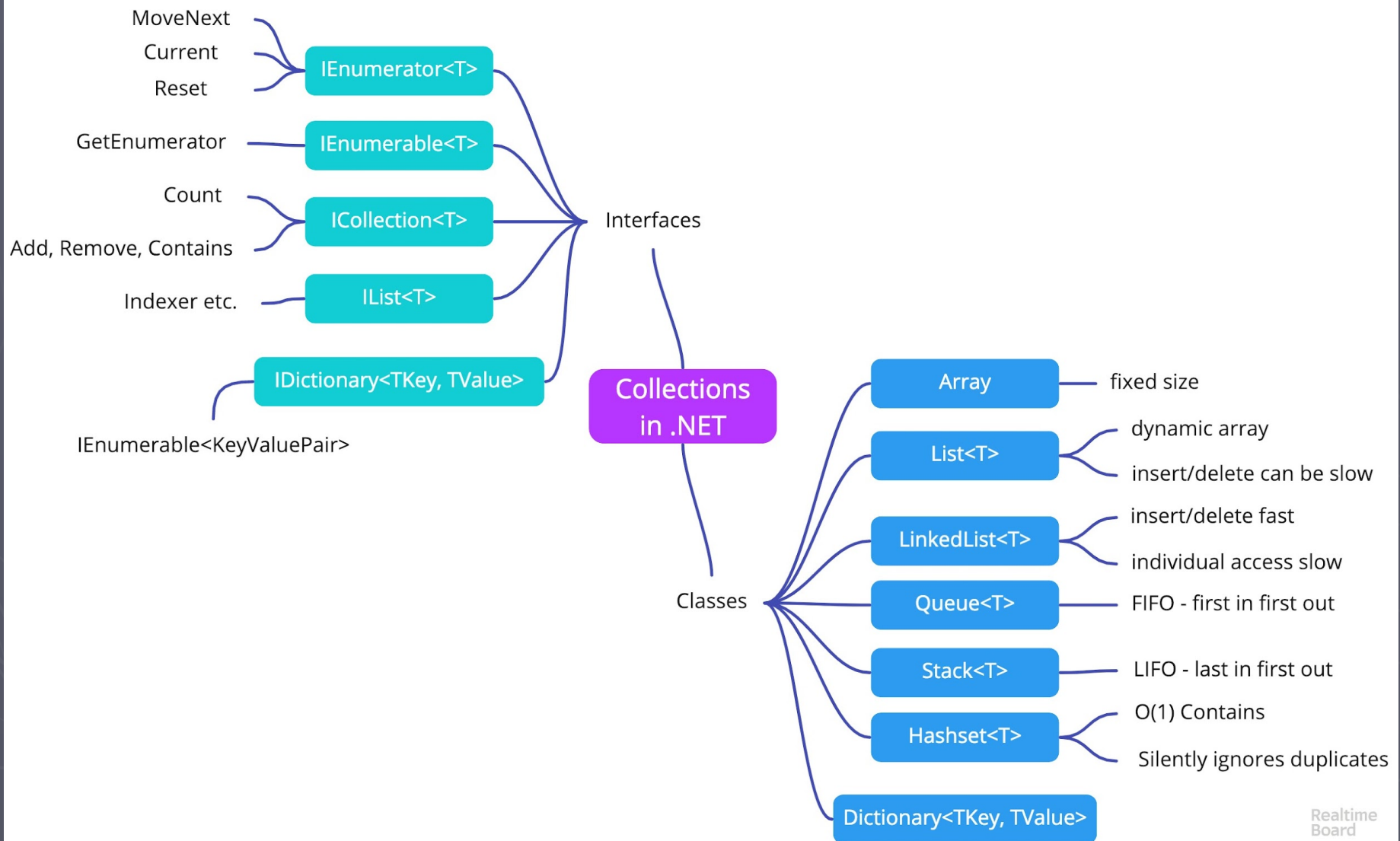
- ▶ `IProducerConsumerCollection<T>`
- ▶ Его методы `TryAdd()` и `TryTake()`

`ConcurrentStack<T>`

`ConcurrentBag<T>`

`ConcurrentDictionary<TKey, TValue>`

`BlockingCollection<T>`



Реализация интерфейса

► Comparable

- Для сортировки и сравнения объектов (SortedList)
- Требуется реализации
 - `int CompareTo(object obj)`

► Comparer

- **`int Compare(object x, object y)`**

```
class Air : IComparable<Air>
{
    public int Number { set; get; }
    public int CompareTo(Air obj)
    {
        if (this.Number > obj.Number)
            return 1;
        if (this.Number < obj.Number)
            return -1;
        else
            return 0;
    }
}

static class Run
{
    public static void Main()
    {
        List<Air> minsk2 = new List<Air>();
        minsk2.Add(new Air());
        minsk2.Sort();
    }
}
```

Интерфейс ICollection

Определяет размер, перечислители и методы синхронизации для всех неуниверсальных коллекций.

```
public interface ICollection : IEnumerable  
{
```

```
    // метод
```

Копирует элементы коллекции ICollection в массив Array, начиная с указанного индекса массива Array.

```
    void CopyTo(Array array, int index);
```

```
    // свойства
```

```
    int Count { get; } Получает элементов
```

```
    bool IsSynchronized { get; }
```

Возвращает значение, показывающее, является ли доступ к коллекции ICollection потокобезопасным

```
    object SyncRoot { get; }
```

```
}
```

Получает объект, с помощью которого можно синхронизировать доступ к коллекции

Универсальный интерфейс ICollection<T>

Определяет методы для управления универсальными коллекциями.

```
public interface ICollection<T> : IEnumerable<T>
{
    // методы
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool Remove(T item);

    // свойства
    int Count { get; }
    bool IsReadOnly { get; }
}
```

содержит ли коллекция указанное значение.

Удаляет первое вхождение указанного объекта

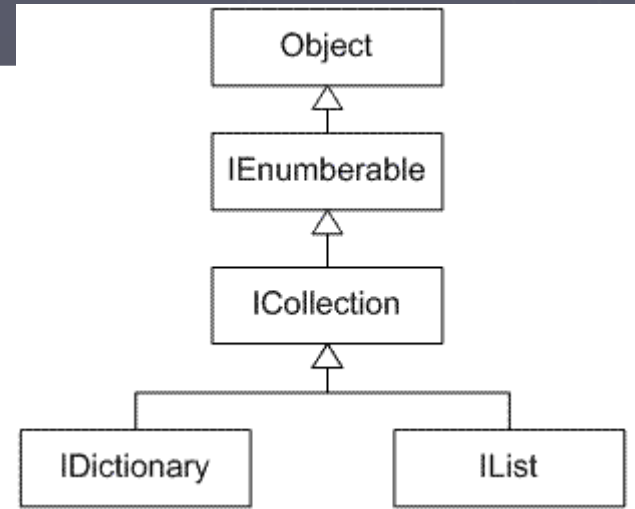
Получает значение, указывающее, является ли объект ICollection<T> доступным только для чтения.

Интерфейс IList

- Представляет неуниверсальную коллекцию объектов, к каждому из которых можно получить индивидуальный доступ по индексу.

```
public interface IList : ICollection
{
    // методы
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);

    // свойства
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
}
```



Получает значение, указывающее, имеет ли список `IList` фиксированный размер.

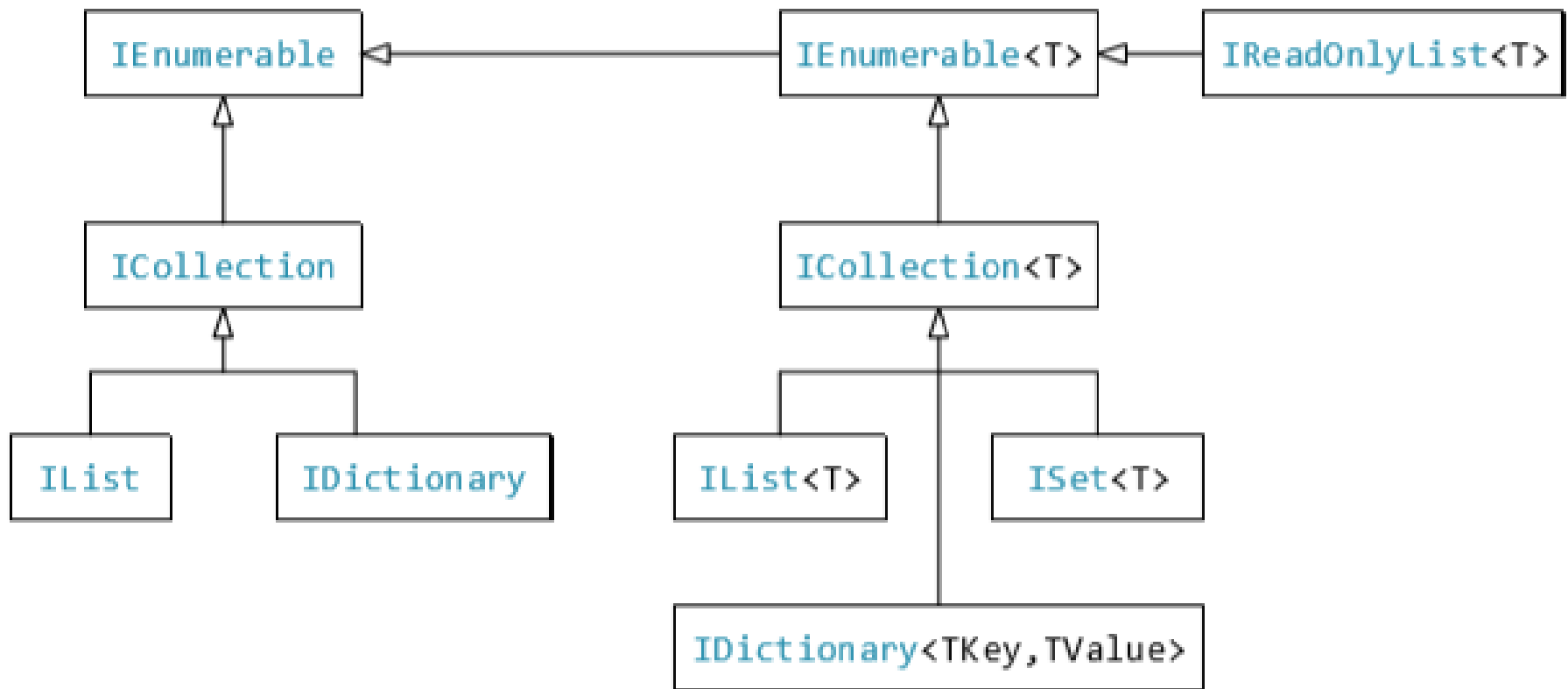
Интерфейс IDictionary

- ▶ протокол взаимодействия для коллекций-словарей (KeyValuePair<TKey, TValue> – это вспомогательная структура, у которой определены свойства Key и Value)

```
public interface IDictionary : ICollection
{
    // методы
    void Add(object key, object value);
    void Clear();
    bool Contains(object key);
    IDictionaryEnumerator GetEnumerator();
    void Remove(object key);

    // свойства
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[object key] { get; set; }
    ICollection Keys { get; }           // все ключи словаря
    ICollection Values { get; }         // все значения словаря
}
```

Стандартные интерфейсы коллекций



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

интерфейс ISet<T>

```
public interface ISet<T> : ICollection<T>
{
    bool Add(T item);
    void ExceptWith(IEnumerable<T> other);
    void IntersectWith(IEnumerable<T> other);
    bool IsProperSubsetOf(IEnumerable<T> other);
    bool IsProperSupersetOf(IEnumerable<T> other);
    bool IsSubsetOf(IEnumerable<T> other);
    bool IsSupersetOf(IEnumerable<T> other);
    bool Overlaps(IEnumerable<T> other);
    bool SetEquals(IEnumerable<T> other);
    void SymmetricExceptWith(IEnumerable<T> other);
    void UnionWith(IEnumerable<T> other);
}
```

- необобщенный интерфейс IEnumerator или обобщенный интерфейс IEnumerator<T> (Перечислители)

- ▶ Реализация **object** Current { **get**; }

- ▶ **bool** MoveNext()

- ▶ **void** Reset()

- ▶ Доступ только для чтения

```
List<int> arrayList = new List<int>();
    Random ran = new Random();

    for (int i = 0; i < 10; i++)
        arrayList.Add(ran.Next(1, 20));

    // Используем перечислитель
    IEnumerator<int> e = arrayList.GetEnumerator();
    e.MoveNext() ;
    Console.Write(e.Current + "\t");
```

Перечислители

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

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