

# Коллекции



# 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

▶ IComparer

- Compare()

# Классы необобщенных коллекций

- ▶ **ArrayList** - IList, ICollection, IEnumerable, ICloneable  
Определяет динамический массив
- ▶ **BitArray** - 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 эти классы недоступны

# Класс ArrayList

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

```
ArrayList arr1 = new ArrayList(); // 16  
ArrayList arr2 = new ArrayList(1000); // 1000  
ArrayList arr3 = new ArrayList();
```

- ▶ Свойства – Capacity, Count, Item
- ▶ Метод - Add , AddRange, BinarySearch, Clear, Clone, CopyTo, GetRange, Sort, RemoveRange, Reverse, IndexOf ....

# пример

```
ArrayList list = new ArrayList();  
list.Add(2.3);  
list.Add(55);  
list.AddRange(new string[] { "one", "two" });  
list.RemoveAt(0);  
list.Reverse();
```

# Обобщенные коллекции

- ▶ Dictionary <Tkey, TValue>
- ▶ LinkedList<T>
- ▶ List<T>
- ▶ Queue<T>
- ▶ SortedDictionary<Tkey, Tvalue>
- ▶ SortedList<T> (использовании памяти и в скорости вставки и удаления)
- ▶ 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 дублир.
SortedDictionary<Tkey, TValue>	Вставка медленнее, извлечение быстрее, использует больше памяти чем
HashSet<T>	Неупорядоченный набор значений, оптимизация - быстрое извлечение данных, объединений и пересечений наборов.

```
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);
```

```
}
```

```
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);
```

# Пример: связный список → класс `LinkedList<T>`

```
public static void Main()
```

```
{
```

```
    LinkedList<int> spisok = new LinkedList<int>();
```

```
    spisok.AddFirst(23);
```

```
    spisok.AddLast(234);
```

```
    LinkedListNode<int> node = spisok.First;
```

```
    node = node.Next;
```

```
    node = node.Previous;
```

```
    spisok.AddAfter(node, 111);
```

```
    spisok.RemoveFirst();
```

```
    spisok.AddLast(4563);
```

```
    for (node = spisok.First; node != null;
```

```
        node = node.Next)
```

```
        Console.Write(node.Value + "\t");
```

```
    node = spisok.Find(111);
```

```
}
```

ICollection,  
ICollection<T>,  
IEnumerable,  
IEnumerable<T>,  
ISerializable и  
IDeserializationCallback

# Пример: Dictionary<T, R>

```
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");
```

```
student.First(( n) => (n.Value==3));
```

```
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**

содержит фабричные методы для создания коллекций

► **HybridDictionary**

Для небольшого ListDictionary  
Для большого количества Hashtable

► **ListDictionary**

для хранения пар "ключ-значение" используется связный список (небольшое количество)

► **NameValueCollection**

пары "ключ-значение" относятся к типу string

► **OrderedDictionary**

индексируемые пары "ключ-значение"

► **StringCollection**

оптимизация для хранения символьных строк

► **StringDictionary**

пары ключ-значение типа string

# Битовые коллекции

## ► Класс BitArray

- Изменяемый размер
- ICollection, IEnumerable, ICloneable

System.Collections.Specialized

*And()  
Get  
Not  
Or  
Xor  
Set*

## ► Структура BitVector32

- 32 бита (целое) - хранение – стек → тип значения  
→ выше скорость работы

```
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");
```





# 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)
        {}
```

# Параллельные коллекции

System.Collections.Concurrent

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

► TryAdd() и TryTake()

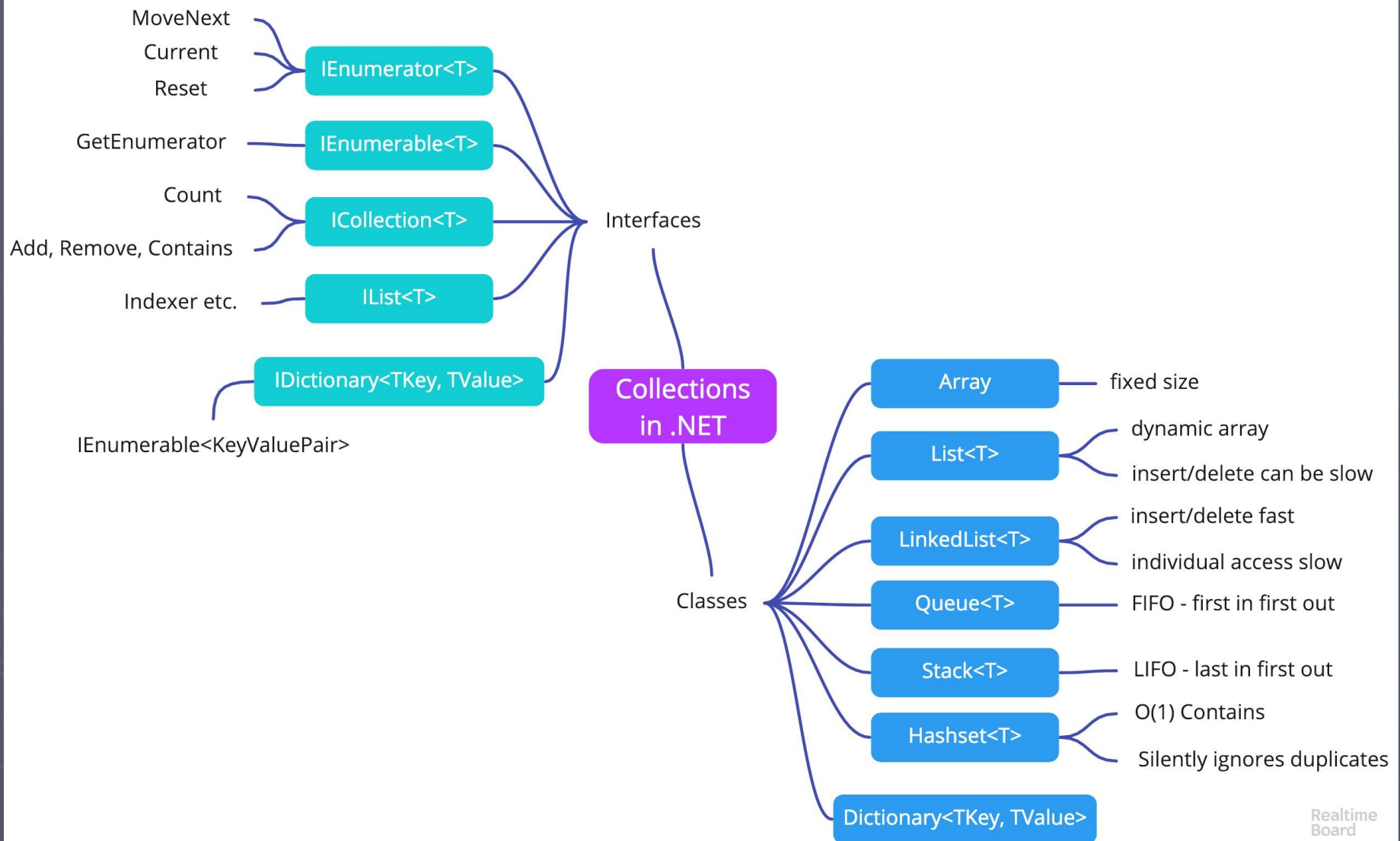
*ConcurrentStack<T>*

*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

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

# Универсальный интерфейс ICollection<T>

```
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);

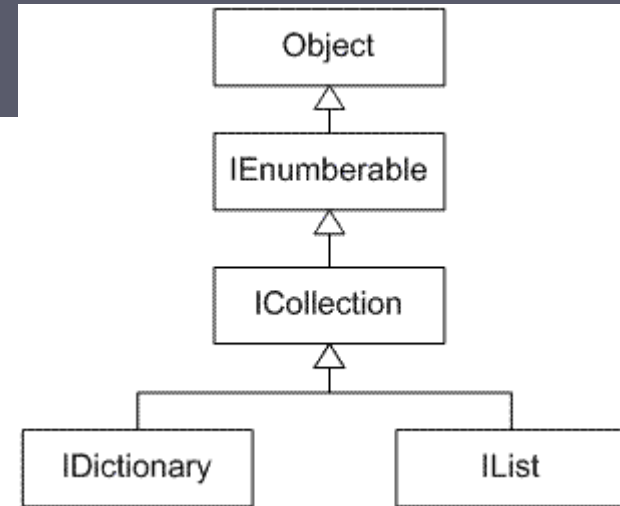
    // свойства
    int Count { get; }
    bool IsReadOnly { get; }
}
```

# Интерфейс IList

- описывает набор данных, которые проецируются на массив

```
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; }
}
```





# 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
}
```

```
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");
```

# Перечеслители

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

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

# Стандартные интерфейсы коллекций

