Collection

Коллекция – это класс, предназначенный для группировки связанных объектов, управления ими и обработки их в циклах

> Коллекции являются важным инструментом программиста, но решение о их применении не всегда оказывается очевидным

Use of Collections

- Отдельные элементы используются для одинаковых целей и одинаково важны.
- На момент компиляции число элементов неизвестно или незафиксировано.
- Необходима поддержка операции перебора всех элементов.
- Необходима поддержка упорядочивания элементов.
- Необходимо использовать элементы из библиотеки, от которой потребитель ожидает наличия типа коллекции.

Список List<T>

Класс List<T> представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

- **void Add(T item)**: добавление нового элемента в список
- **void AddRange(ICollection collection)**: добавление с список коллекции или массива
- **int BinarySearch(T item)**: бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
- **int IndexOf(T item)**: возвращает индекс первого вхождения элемента в списке
- **void Insert(int index, T item)**: вставляет элемент item в списке на позицию index
- **bool Remove(T item)**: удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true
- **void RemoveAt(int index)**: удаление элемента по указанному индексу index

- **void Sort()**: сортировка списка

Двухсвязный список LinkedList<T>

Класс `LinkedList<T>` представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент.

Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел представляет объект класса `LinkedListNode<T>`. Этот класс имеет следующие свойства:

- **Value**: само значение узла, представленное типом T
- **Next**: ссылка на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null
- **Previous**: ссылка на предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null

Используя методы класса LinkedList<T>, можно обращаться к различным элементам, как в конце, так и в начале списка:

- **AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет узел newNode в список после узла node.
- **AddAfter(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением value после узла node.
- **AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет в список узел newNode перед узлом node.
- **AddBefore(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением value перед узлом node.
- **AddFirst(LinkedListNode<T> node)**: вставляет новый узел в начало списка
- **AddFirst(T value)**: вставляет новый узел со значением value в начало списка
- **AddLast(LinkedListNode<T> node)**: вставляет новый узел в конец 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

```
    LinkedList<Person> persons = new LinkedList<Person>();

    // добавляем persona в список и получим объект LinkedListNode<Person>, в котором
хранится имя Tom
    LinkedListNode<Person> tom = persons.AddLast(new Person() { Name = "Tom" });
    persons.AddLast(new Person() { Name = "John" });
    persons.AddFirst(new Person() { Name = "Bill" });
```

```
    Console.WriteLine(tom.Previous.Value.Name); // получаем узел перед томом и его
значение
    Console.WriteLine(tom.Next.Value.Name); // получаем узел после тома и его значение

    Console.ReadLine();
```

Очередь Queue<T>

Класс `Queue<T>` представляет обычную очередь, работающую по алгоритму FIFO ("первый вошел - первый вышел").

У класса `Queue<T>` можно отметить следующие методы:

- **Dequeue**: извлекает и возвращает первый элемент очереди
- **Enqueue**: добавляет элемент в конец очереди
- **Peek**: просто возвращает первый элемент из начала очереди без его удаления

```
            Queue<int> numbers = new Queue<int>();

            numbers.Enqueue(3); // очередь 3
            numbers.Enqueue(5); // очередь 3, 5
            numbers.Enqueue(8); // очередь 3, 5, 8

            // получаем первый элемент очереди
            int queueElement = numbers.Dequeue(); //теперь очередь 5, 8
            Console.WriteLine(queueElement);
```

Коллекция Stack<T>

Класс `Stack<T>` представляет коллекцию, которая использует алгоритм LIFO ("последний вошел - первый вышел"). При такой организации каждый следующий добавленный элемент помещается поверх предыдущего. Извлечение из коллекции происходит в обратном порядке - извлекается тот элемент, который находится выше всех в стеке.

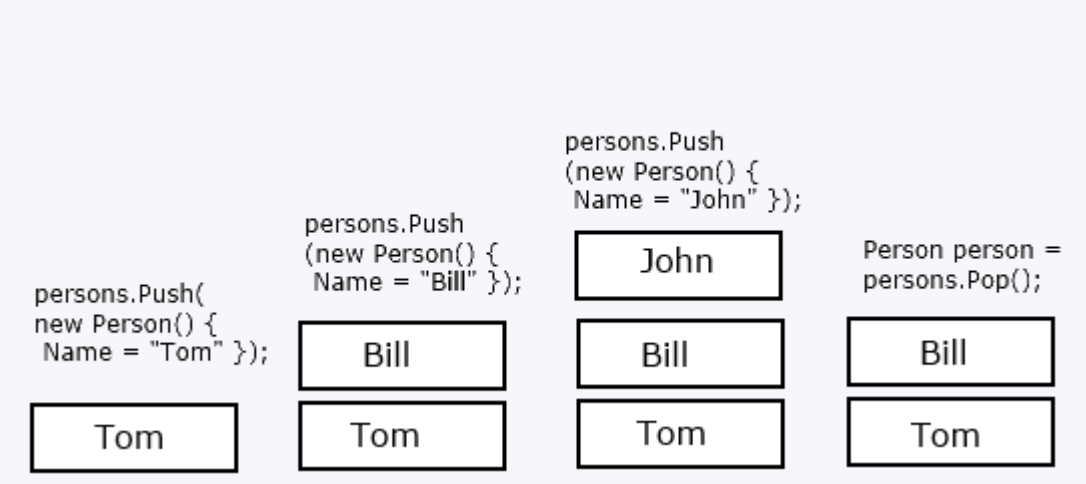
В классе Stack можно выделить два основных метода, которые позволяют управлять элементами:

- **Push**: добавляет элемент в стек на первое место
- **Pop**: извлекает и возвращает первый элемент из стека
- **Peek**: просто возвращает первый элемент из стека без его удаления

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

            numbers.Push(3); // в стеке 3
            numbers.Push(5); // в стеке 5, 3
            numbers.Push(8); // в стеке 8, 5, 3
```

```
            // так как вверху стека будет находиться число 8, то оно и извлекается
            int stackElement = numbers.Pop(); // в стеке 5, 3
            Console.WriteLine(stackElement);
```



Коллекция Dictionary<T, V>

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Каждый такой объект является объектом класса **KeyValuePair<TKey, TValue>**. Благодаря свойствам `Key` и `Value`, которые есть у данного класса, мы можем получить ключ и значение элемента в словаре

```
Dictionary<int, string> countries = new Dictionary<int, string>(5);
countries.Add(1, "Russia");
countries.Add(3, "Great Britain");
countries.Add(2, "USA");
countries.Add(4, "France");
countries.Add(5, "China");

foreach (KeyValuePair<int, string> keyValue in countries)
{
    Console.WriteLine(keyValue.Key + " - " + keyValue.Value);
}

// получение элемента по ключу
string country = countries[4];
// изменение объекта
countries[4] = "Spain";
// удаление по ключу
countries.Remove(2);
```

Класс словарей также, как и другие коллекции, предоставляет методы Add и Remove для добавления и удаления элементов. Только в случае словарей в метод Add передаются два параметра: ключ и значение. А метод Remove удаляет не по индексу, а по ключу

Инициализация словарей

```
Dictionary<string, string> countries = new Dictionary<string, string>
{
    {"Франция", "Париж"},
    {"Германия", "Берлин"},
    {"Великобритания", "Лондон"}
};

foreach(var pair in countries)
    Console.WriteLine("{0} - {1}", pair.Key, pair.Value);
```

Класс ObservableCollection

Кроме стандартных классов коллекций типа списков, очередей, словарей, стеков .NET также предоставляет специальный класс ObservableCollection. Он по функциональности поход на список List за тем исключением, что позволяет известить внешние объекты о том, что коллекция была изменена

```
class Program
    {
        static void Main(string[] args)
        {
            ObservableCollection<User> users = new ObservableCollection<User>
            {
                new User { Name = "Bill"},
                new User { Name = "Tom"},
                new User { Name = "Alice"}
            };

            users.CollectionChanged += Users_CollectionChanged;

            users.Add(new User { Name = "Bob" });
            users.RemoveAt(1);
            users[0] = new User{ Name = "Anders" };

            foreach(User user in users)
            {
                Console.WriteLine(user.Name);
            }

            Console.Read();
        }

        private static void Users_CollectionChanged(object sender,
NotifyCollectionChangedEventArgs e)
        {
            switch(e.Action)
            {
                case NotifyCollectionChangedAction.Add: // если добавление
                    User newUser = e.NewItems[0] as User;
                    Console.WriteLine("Добавлен новый объект: {0}", newUser.Name);
                    break;
```

```
                case NotifyCollectionChangedAction.Remove: // если удаление
                    User oldUser = e.OldItems[0] as User;
                    Console.WriteLine("Удален объект: {0}", oldUser.Name);
                    break;
                case NotifyCollectionChangedAction.Replace: // если замена
                    User replacedUser = e.OldItems[0] as User;
                    User replacingUser = e.NewItems[0] as User;
                    Console.WriteLine("Объект {0} заменен объектом {1}",
                                      replacedUser.Name, replacingUser.Name);
                    break;
            }
        }
```

Во-первых, класс ObservableCollection находится в пространстве имен `System.Collections.ObjectModel`, кроме того, также понадобятся ряд объектов из пространства `System.Collections.Specialized`, поэтому в начале подключаем эти пространства имен.

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

В обработчике этого события `Users_CollectionChanged` для получения всей информации о событии используется объект`NotifyCollectionChangedEventArgs e`. Его свойство `Action` позволяет узнать характер изменений. Оно хранит одной из значений из перечисления **NotifyCollectionChangedAction**.

Свойства `NewItems` и `OldItems` позволяют получить соответственно добавленные и удаленные объекты. Таким образом, мы получаем полный контроль над обработкой добавления, удаления и замены объектов в коллекции.

IEnumerable

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

`IEnumerator` GetEnumerator() - возвращает перечислитель, который можно использовать для навигации по коллекции.

IEnumerator

```
public interface IEnumerator
{
```

```
    bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов
    object Current {get;}  // текущий элемент в контейнере
    void Reset(); // перемещение в начало контейнера
}
```

Свойства интерфейса `IEnumerator`:

`Object Current {get;}` – возвращает текущий элемент коллекции.

Методы интерфейса `IEnumerator`:

`Bool MoveNext()` – перемещает перечислитель на следующий элемент коллекции.

`Void Reset()` – возвращает перечислитель на начало коллекции.

```
class Library : IEnumerable
{
    private Book[] books;

    public Library()
    {
        books = new Book[] { new Book("Отцы и дети"), new Book("Война и мир"),
                new Book("Евгений Онегин") };
    }

    public int Length
    {
        get { return books.Length; }
    }

    public Book this[int index]
    {
        get
        {
            return books[index];
        }
        set
        {
            books[index] = value;
        }
    }

    // возвращаем перечислитель
    IEnumerator IEnumerable.GetEnumerator()
    {
        return books.GetEnumerator();
    }
```

```
}
```

Yield

Мы можем не полагаться на реализацию перечислителя в массиве, а создать итератор с помощью ключевого слова **yield**.**Итератор** представляет метод, в котором используется ключевое слово yield для перебора по коллекции или массиву

- Блок, в котором содержится ключевое слово `yield`, расценивается компилятором, как блок итератора.
- Ключевое слово `return` используется для предоставления значения объекту перечислителя.
- Ключевое слово `break` используется для обозначения конца итерации.

```
public static IEnumerable Power()
{
    yield return "Hello world!";
}
```

```
public static IEnumerable Power()
{
    yield break;
}
```

```
IEnumerator IEnumerable.GetEnumerator()
{
    for (int i = 0; i < books.Length; i++)
    {
        yield return books[i];
    }
}
```

Foreach

Циклическая конструкция `foreach` позволяет выполнять навигацию по коллекции, используя реализации интерфейсов `IEnumerable` и `IEnumerator`

```
foreach (var element in myCollection)
{

}
```

`Var` – Локальная переменная с неявным типом имеет строгую типизацию, как если бы тип был задан явно, только тип определяет компилятор

IEnumerable<T>

`IEnumerable`<T> - унаследован от `IEnumerable`

Методы интерфейса `IEnumerable`<T>:

> `IEnumerator`<T> GetEnumerator()-возвращает обобщенный перечислитель, который можно использовать для навигации по коллекции