Лабораторная работа №5

«Массивы и коллекции. Интерфейс IEnumerable»

**Цели работы:**

1. Научиться реализации интерфейса IEnumerable и работе с массивами и коллекциями языка C#.

**Задание№1**

Создайте класс MyMatrix, представляющий матрицу m на n.

Создайте конструктор, принимающий число строк и столбцов, заполняющий матрицу случайными числами в диапазоне, который пользователь вводит при запуске программы.

Создайте метод Fill, перезаполняющий матрицу случайными значениями.

Создайте метод ChangeSize, изменяющий число строк и/или столбцов с копированием значений существующей матрицы. Если новая матрица больше существующий, то метод должен дозаполнить новую матрицу случайными числами.

Создайте метод ShowPartialy, принимающий в качестве параметров начальные и конечные значения строк и столбцов, значения матрицы внутри которых нужно вывести на консоль.

Создайте метод Show, выводящий все значения матрицы на консоль.

Создайте индексатор для матрицы вида this[int index1, int index2] с аксессором и мутатором.

**Задание№2**

Создайте класс MyList<T>.

Реализуйте в простейшем приближении возможность использования его экземпляра аналогично экземпляру класса List<T>.

Минимально требуемый интерфейс взаимодействия с экземпляром должен включать метод добавления элемента, индексатор для получения значения элемента по указанному индексу, свойство получения общего количества элементов и поддержку инициализатора коллекции.

При выполнении нельзя использовать коллекции, только массивы.

**Задание№3**

Создайте коллекцию MyDictionary<TKey,TValue>.

Реализуйте в простейшем приближении возможность использования ее экземпляра аналогично экземпляру класса Dictionary<TKey,TValue>.

Минимально требуемый интерфейс взаимодействия с экземпляром должен включать метод добавления элемента, индексатор для получения значения элемента по указанному индексу и свойство только для чтения для получения общего количества элементов.

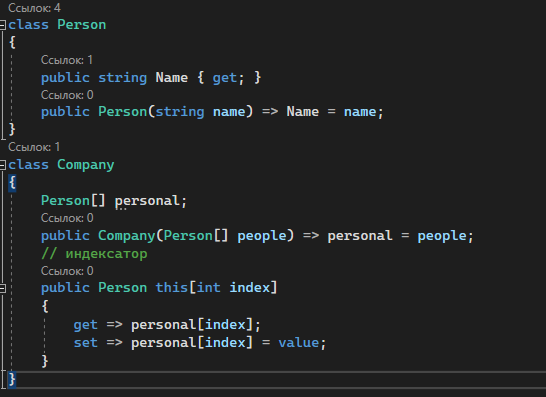
Реализуйте возможность перебора элементов коллекции в цикле foreach. При выполнении нельзя использовать коллекции, только массивы.

**Теоретические сведения**

**Индексаторы**

Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово this, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

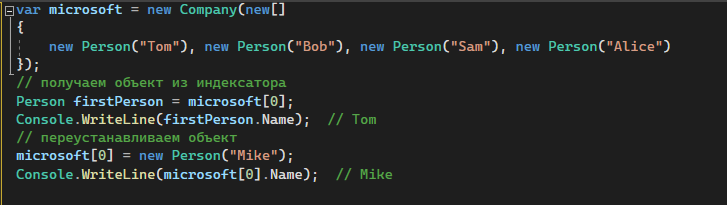


Индексатор в принципе подобен стандартному свойству. Во-первых, для индексатора определяется тип в данном случае тип Person. Тип индексатора определяет, какие объекты будет получать и возвращать индексатор.

Во-вторых, для индексатора определен параметр int index, через который обращаемся к элементам внутри объекта Company. Для возвращения объекта в индексаторе определен блок get.

В блоке set, как и в обычном свойстве, получаем через параметр value переданный объект Person и сохраняем его в массив по индексу.

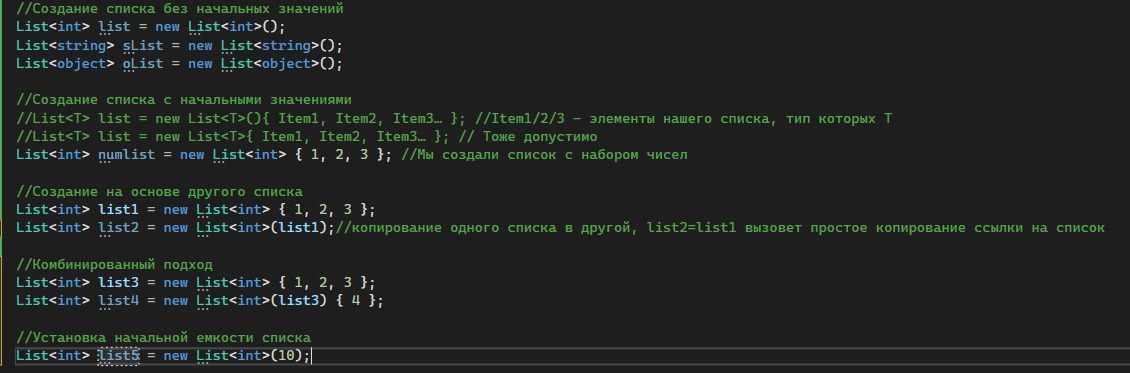
После этого мы можем работать с объектом Company как с набором объектов Person:



**List<T>**

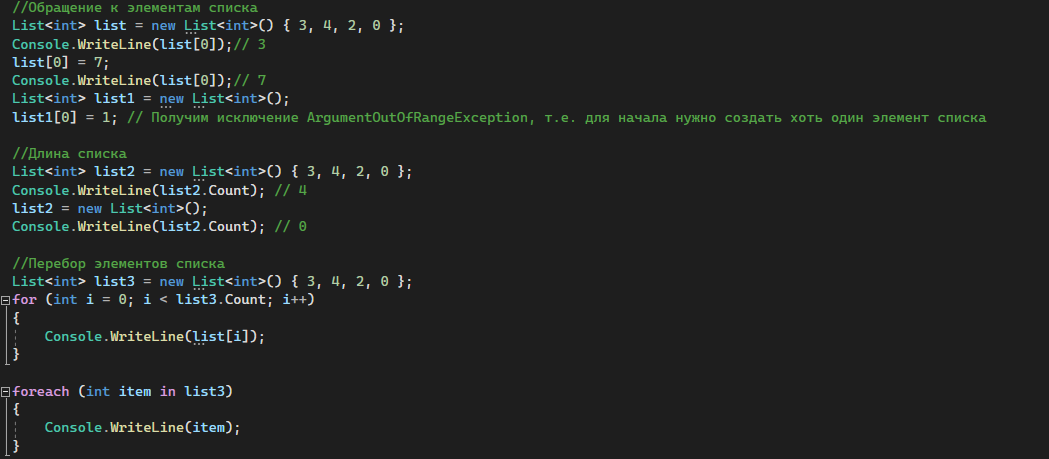
List<T> — класс из пространства имен System.Collections.Generic, список однотипных объектов. В отличие от массива, предоставляет набор методов, облегчающих работу, таких как добавление новых элементов (это удобно, когда неизвестно заранее сколько будет элементов).

Есть несколько вариантов создания списка:

****

Внутри списка находится массив, который при заполнении определенной емкости динамически расширяется, чтобы снизить издержки на выделение памяти при добавлении элементов. На низком уровне при проверке расширения проверяется, если в массиве (который внутри списка) нет элементов, то его размер выставляется равным \_defaultCapacity (переменная, которая имеет начальное значение = 4), в противном случае считается из «количество элементов» \* 2. Т. е. всякий раз, когда мы достигаем определенной границы массива, List автоматически расширяет его в 2 раза, однако ограничение списка — это Array.MaxArrayLength – 2146435071 элементов. Для повышения производительности, если известен конечный размер списка, можно изначально задать его размер, что избавит от дополнительных выделений памяти.

**Работа со списком**



### Методы списка: Добавление в список

* void Add(T item) — добавляет новый элемент с типом T в конец списка.
* void AddRange(IEnumerable<T> collection) — добавляет в список коллекцию или массив в конец списка.
* void Insert(int index, T item) — вставляет элемент в коллекцию по указанному индексу. При выходе за границы коллекции (0 > index или Count-1 < index) будет сформировано исключение System.ArgumentOutOfRangeException.
* void InsertRange(int index, IEnumerable<T> collection) — аналогично методу выше, только вставляет список элементов по указанному индексу. Также при выходе за границы будет сформировано исключение System.ArgumentOutOfRangeException.

### Методы списка: Поиск и проверка элемента

* int BinarySearch(T item) — бинарный поиск элемента в списке, возвращает индекс элемента. Работает корректно, если список отсортирован. Имеет еще два варианта реализации с дополнительными параметрами для корректного сравнения элементов или указания начальной точки поиска.
* bool Contains(T item) — проверка наличия элемента в списке.
* bool Exists(Predicate<T> match) — проверяет наличие в списке хотя бы одного элемента удовлетворяющего условиям делегата match.
* T Find(Predicate<T> match) — возвращает первый элемент, удовлетворяющий условиям делегата match.
* List<T> FindAll(Predicate<T> match) — аналогичен Find, только возвращает все элементы.
* int FindIndex(Predicate<T> match) — возвращает первый индекс элемента, удовлетворяющего условиям делегата match. Имеет еще два варианта реализации с указанием позиции поиска и количества элементов.
* int FindLastIndex(Predicate<T> match) — аналогично FindIndex, только возвращает индекс последнего элемента, удовлетворяющего условиям делегата match. Также имеет еще два варианта реализации с указанием позиции поиска и количества элементов.
* List<T> GetRange(int index, int count) — этот метод получает подсписок из списка от указанного индекса с указанным количеством элементов.
* int IndexOf(T item) — возвращает первый индекс элемента, если он найден в списке либо -1. Также имеет еще два реализации с указанием позиции поиска и количества элементов.
* int LastIndexOf(T item) — возвращает последний индекс элемента, если он найден в списке либо -1. Также имеет еще два реализации с указанием позиции поиска и количества элементов.

### Методы списка: Удаление из списка

* bool Remove(T item) — удаляет первое вхождение указанного элемента из списка. Возвращает true, если элемент был удален или false, если нет.
* int RemoveAll(Predicate<T> match) — удаляет все элементы, удовлетворяющие условиям делегата match. Возвращает количество удаленных элементов.
* void RemoveAt(int index) — удаляет элемент списка с указанным индексом. При выходе индекса за границы списка формирует исключение System.ArgumentOutOfRangeException.
* void RemoveRange(int index, int count) — удаляет элементы массива с указанного индекса в указанном количестве. Если index меньше 0 или значение параметра count меньше 0, то будет сформировано исключение System.ArgumentOutOfRangeException. Если параметры index и count не указывают допустимый диапазон элементов в списке, будет сформировано исключение System.ArgumentException.
* void Clear() — удаляет все элементы из списка.

### Методы списка: Сортировка списка

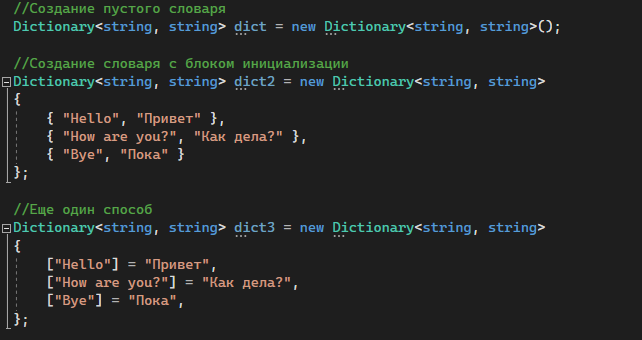
* void Sort() — сортирует элементы. В основе лежит алгоритм быстрой сортировки с ограничением глубины рекурсии до 32, при достижении которого используется сортировка кучей.
* void Reverse() — изменяет порядок элементов во всем списке на обратный. Имеет перегруженную версию, в которой указывается начальный индекс и количество элементов.

### Методы списка: Прочее

* List<TOutput> ConvertAll<TOutput>(Converter<T, TOutput> converter) — позволяет сконвертировать все типы элементов текущего списка в другой тип. На входе принимает делегат на метод, преобразующий объект от одного типа к другому.
* void CopyTo(T[] array) — копирует список в массив. Имеет еще 2 варианта реализации с указанием границ и позиций копирования.
* void ForEach(Action<T> action) — это аналог цикла foreach, только в качестве итерации будет выполняться метод, который на входе принимает элемент с типом T. Еще одно отличие такого цикла от foreach состоит в том, что в методе не сработает ни break, ни continue. Аналогом continue при таком подходе можно считать return.
* T[] ToArray() — возвращает массив элементов списка.

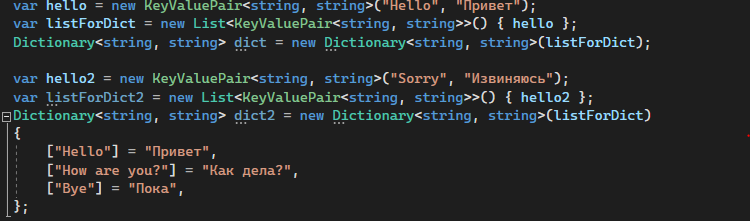
**Dictionary<K, V>**

Dictionary<K, V> — класс пространства имён System.Collections.Generic, который представляет из себя коллекцию ключей и значений, также называемый Словарем. Коллекция типизируется двумя типами: K(key) — тип ключа и V(value) — тип значения. Коллекция позволяет получать значения со скоростью близкой к O(1). Скорость зависит от качества алгоритма хеширования типа, заданного для ключа.



**Dictionary<K, V> : KeyValuePair**

По сути, словарь — это коллекция, т. е. набор элементов, тип элементов — KeyValuePair<TKey, TValue>, где TKey — ключ, а TValue — значение. Данная структура предоставляет свойства Key и Value, с помощью которых можно получить ключ и значение. Один из конструкторов Dictionary<TKey, TValue> принимает на входе список элементов типа KeyValuePair<TKey, TValue>



### Dictionary: **Добавление**

* void Add(TKey key, TValue value) — Добавляет элемент в коллекцию с ключом key и значением value
* bool TryAdd(TKey key, TValue value) — Метод пытается добавить новый элемент в коллекцию. Если ключ в словаре не найден, то метод ничего не сделает и вернёт false.

**Dictionary: Удаление**

* bool Remove(TKey key) — Удаляет элемент по ключу, при успехе возвращает — true
* bool Remove(TKey key, out TValue value) — Аналогично примеру выше, но ещё помещает значение удалённого элемента в выходной параметр value.
* void Clear() — Очищает словарь.

**Dictionary: Получение**

* bool TryGetValue(TKey key, out TValue value) — Пытается получить значение по ключу, при успехе возвращает true и записывает полученное значение в переменную value

**Dictionary: Прочее**

* bool ContainsKey(TKey key) — Проверяет наличие ключа в словаре.
* bool ContainsValue(TValue value) — Проверяет наличие значения в словаре.
* int EnsureCapacity(int capacity) — Обеспечивает возможность хранения указанного количества записей в словаре без дальнейшего увеличения его резервного хранилища. Возвращает текущее количество элементов в словаре.
* void TrimExcess() — Устанавливает ёмкость словаря такой, какой бы она была, если словарь был изначально инициализирован со всеми записями.
* void TrimExcess(int capacity) — Устанавливает ёмкость словаря такой, чтобы в нём помещалось указанное количество записей без дальнейшего увеличения его резервного хранилища. Если capacity меньше текущей ёмкости словаря, то генерирует исключение ArgumentOutOfRangeException.

#### Dictionary: Свойства

* int Count { get; } — Возвращает число элементов в словаре.
* IEqualityComparer<TKey> Comparer { get; } — Возвращает интерфейс IEqualityComparer<T>, используемый для установления равенства ключей словаря.
* Dictionary<TKey, TValue>.KeyCollection Keys { get; } — Коллекция ключей.
* Dictionary<TKey, TValue>.ValueCollection Values { get; } — Коллекция значений.
* TValue this[TKey key] — Индексатор возвращает значение по заданному ключу.