<epam>

Коллекции

Что такое коллекции и зачем они нужны?

ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ

Коллекция в программировании — программный объект, содержащий в себе, тем или иным образом, набор значений одного или различных типов, и позволяющий обращаться к этим значениям.

Коллекция позволяет записывать в себя значения и извлекать их. Назначение коллекции — служить хранилищем объектов и обеспечивать доступ к ним. Обычно коллекции используются для хранения групп однотипных объектов, подлежащих стереотипной обработке. Для обращения к конкретному элементу коллекции могут использоваться различные методы, в зависимости от её логической организации. Реализация может допускать выполнение отдельных операций над коллекциями в целом. Наличие операций над коллекциями во многих случаях может существенно упростить программирование.

Какие бывают коллекции?

ВИДЫ КОЛЛЕКЦИЙ

- **Массив**
- **Динамический массив**
- **Список**
- **Хэш-таблица**
- **Словарь**
- **Очередь**
- **Стэк**
- **Обобщенные коллекции**
- **И многие другие...**

Массив

САМАЯ ПРОСТАЯ КОЛЛЕКЦИЯ

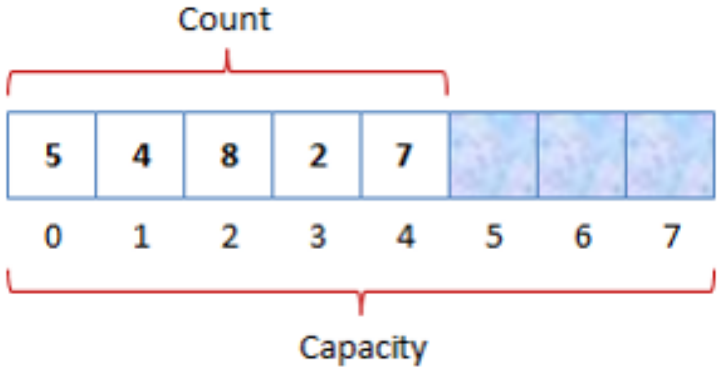
- **Длина неизменяема**
- **Обращение к элементам по индексу**
- **Индексация начинается с 0!!!**
- **Массив может быть любого типа**
- **Массив может быть многомерным**

```
int[] myArray = new int[5];
myArray[0] = 1;
myArray[1] = 2;
myArray[4] = 5;
int first = myArray[1];
int last = myArray[4];
```

Динамический массив

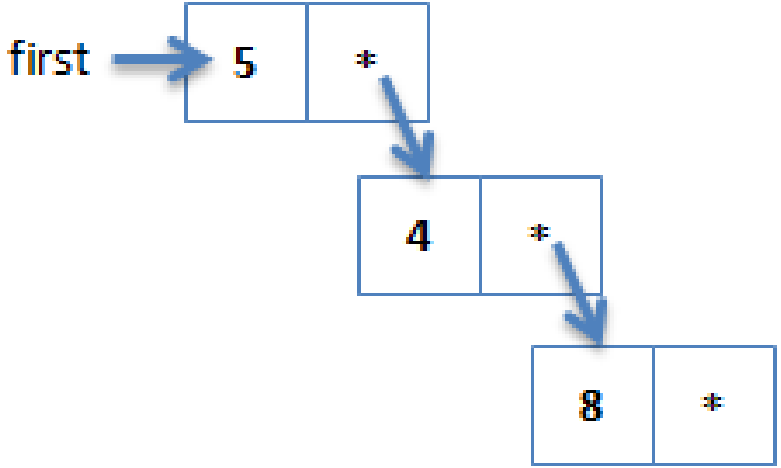
МАССИВ ПЕРЕМЕННОЙ ДЛИНЫ

- **Является надстройкой над обычным массивом**
- **Не пересоздает массив при добавлении элемента, пока не достигнут предел вместимости**



Связный список

- **Элементы связанного списка имеют ссылки на следующий и (если это двусвязный список) предыдущий элемент списка.**
- **Нельзя обращаться к элементам такого списка с помощью индексов.**
- **Переключаться по элементам такой коллекции можно только последовательно вперёд или назад.**
- **Удобное добавление и удаление.**



Хэш-таблица

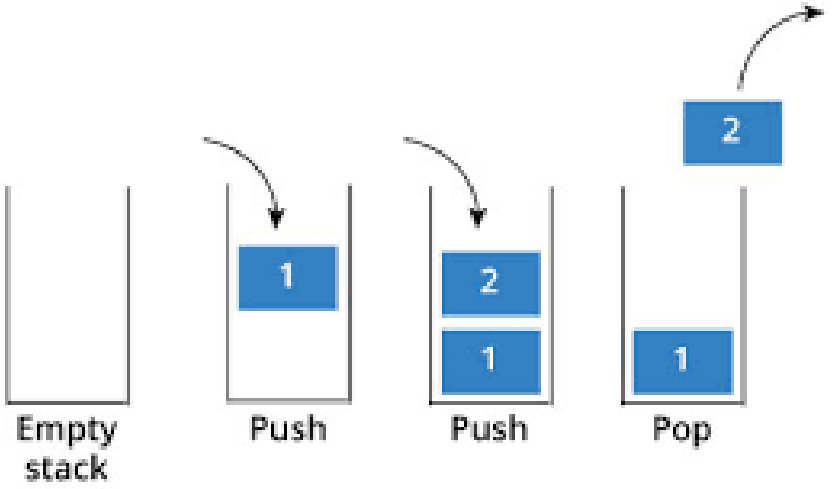
- Данные хранятся в паре «ключ-значение»
- Доступ к данным осуществляется по ключу
- При добавлении элемента вычисляется хэш-код ключа, при доступе по ключу вычисляется хэш-код и поиск данных.

Иванов	данные
Петров	данные
Сидоров	данные

Стэк

LAST IN – FIRST OUT (LIFO)

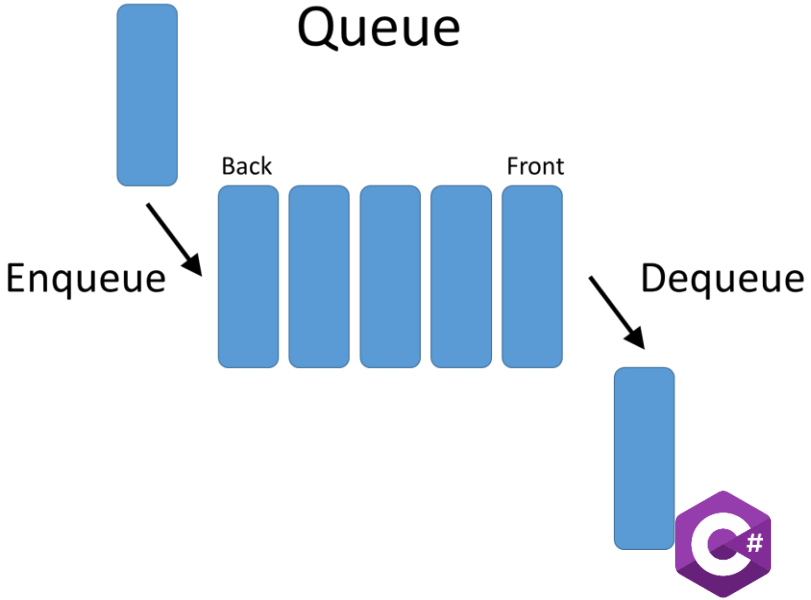
- Динамическая коллекция
- Работает по принципу «последний вошел – первый вышел»
- Главные операции: добавить элемент на вершину и взять элемент с вершины.
- Используется лишь в алгоритмических целях.



Очередь

FIRST IN – FIRST OUT (FIFO)

- Динамическая коллекция
- Работает по принципу «первый вошел – первый вышел»
- Главные операции: добавить элемент в конец очереди и взять элемент из начала очереди.
- Используется лишь в алгоритмических целях.



Обобщенные коллекции

КОЛЛЕКЦИИ, ХРАНЯЩИЕ ЭЛЕМЕНТЫ ЛЮБОГО ТИПА

- List<T>
- Stack<T>
- Queue<T>
- Dictionary<TKey, TValue>
- …

Обобщенный список List<T>

САМАЯ ЧАСТО ИСПОЛЬЗУЕМАЯ КОЛЛЕКЦИЯ

Основные методы:

- Add
- AddRange
- Clear
- Contains
- Find
- Insert
- Remove
- Sort
- ...

```
var testNames = new List<string>();
testNames.Add("jasey lane");
testNames.Add("bruno mars");
testNames.Add("john lennon");
testNames.Add("just name");

// Iterate through the list.
foreach (var testName in testNames)
{
    Console.Write(testName + " ");
}
```

Словарь Dictionary<TKey, TValue>

ПОЛЕЗНАЯ КОЛЛЕКЦИЯ ДЛЯ БЫСТРОГО ПОИСКА ЭЛЕМЕНТА ПО КЛЮЧУ

Основные методы:

- Add
- ContainsKey
- ContainsValue
- Remove

```
IDictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1,"One");
dict.Add(2,"Two");
dict.Add(3,"Three");

IDictionary<int, string> dict = new Dictionary<int, string>()
{
    {1,"One"},
    {2, "Two"},
    {3,"Three"}
};
```

Пара ключ-значение KeyValuePair<TKey, TValue>

- Используется в коллекциях
- Не является коллекцией

```
List<KeyValuePair<int, string>> myList = new List<KeyValuePair<int, string>>();
myList.Add( item: new KeyValuePair<int, string>(1, "Hello"));
myList.Add( item: new KeyValuePair<int, string>(2, "World"));
KeyValuePair<int, string> item = myList[0];
int key = item.Key;
string value = item.Value;
```

Таблица сложности операций

КАК ПОДОБРАТЬ НУЖНУЮ КОЛЛЕКЦИЮ

	add to end	remove from end	insert at middle	remove from middle	Random Access	In-order Access	Search for specific element	**Notes**
Array	O(n)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)	Most efficient use of memory; use in cases where data size is fixed.
List<T>	best case O(1); worst case O(n)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)	Implementation is optimized for speed. In many cases, List will be the best choice.
Collection<T>	best case O(1); worst case O(n)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)	List is a better choice, unless publicly exposed as API.
LinkedList<T>	O(1)	O(1)	O(1)	O(1)	O(n)	O(1)	O(n)	Many operations are fast, but watch out for cache coherency.
Stack<T>	best case O(1); worst case O(n)	O(1)	N/A	N/A	N/A	N/A	N/A	Shouldn't be selected for performance reasons, but algorithmic ones.
Queue<T>	best case O(1); worst case O(n)	O(1)	N/A	N/A	N/A	N/A	N/A	Shouldn't be selected for performance reasons, but algorithmic ones.
Dictionary<K,T>	best case O(1); worst case O(n)	O(1)	best case O(1); worst case O(n)	O(1)	O(1)*	O(1)*	O(1)	Although in-order access time is constant time, it is usually slower than other structures due to the over-head of looking up the key.

Сортировка коллекций

ПРИМЕРЫ СОРТИРОВОК

```
int[] myArray = { 1, 6, 3, 8, 5, 2 };
Array.Sort(myArray);

List<int> myList = new List<int> { 1, 4, 6, 3, 2, 9, 7 };
myList.Sort();
```

Методы Equals и GetHashCode

```
public class ImaginaryNumber : IEquatable<ImaginaryNumber>
{
    public double RealNumber { get; set; }
    public double ImaginaryUnit { get; set; }

    public override bool Equals(object obj)
    {
        return Equals(obj as ImaginaryNumber);
    }

    public bool Equals(ImaginaryNumber other)
    {
        return other != null && RealNumber == other.RealNumber
               && ImaginaryUnit == other.ImaginaryUnit;
    }

    public override int GetHashCode()
    {
        var hashCode = 352033288;
        hashCode = hashCode * -1521134295 + RealNumber.GetHashCode();
        hashCode = hashCode * -1521134295+ImaginaryUnit.GetHashCode();
        return hashCode;
    }
}
```

Библиотека LINQ (Language Integrated Query)

Основные операции с коллекциями:

- Фильтрация
- Проекция
- Объединение
- Упорядочивание
- Группировка
- Агрегация
- Выборка
- ...

Фильтрация

ФИЛЬТРАЦИЯ С ПОМОЩЬЮ МЕТОДА WHERE

С использованием библиотеки Linq:

```
var source = new int[] { -1, -2, 3, 4, -5, -6, 0 };
var nonNegative = source.Where(num => num >= 0);
```

Использование стандартных методов:

```
var source = new int[] { -1, -2, 3, 4, -5, -6, 0 };
var nonNegative = new List<int>();
foreach(var num in source)
{
    if (num >= 0)
    {
        nonNegative.Add(num);
    }
}
```

Выборка

ВЫБОРКА С ПОМОЩЬЮ МЕТОДА SELECT

Проецирование каждого элемента последовательности в новую форму:

```
ссылка: 1
class MyClass
{
    ссылка: 1
    public int Number { get; set; }
}

ссылок: 0
static void Main(string[] args)
{
    var source = new int[] { -1, -2, 3, 4, -5, -6, 0 };
    var objects = source.Select(num => new MyClass
    {
        Number = num
    });
}
```

Упорядочивание

УПОРЯДОЧИВАНИЕ С ПОМОЩЬЮ МЕТОДОВ ORDERBY И THENBY

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  "Chrysler", "Dodge", "BMW",
    "Ferrari",  "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

var auto :IOrderedEnumerable<string> = cars.OrderBy(s :string => s.Length).ThenBy(s :string=>s);
auto = cars.OrderByDescending(s :string => s);
```

Методы OrderBy и ThenBy упорядочивают коллекцию на основе метода, который выделяет сравнимые (IComparable) ключи для каждого элемента. В данном случае методы передаются с помощью лямбда-выражений.

Цепочки методов

ИЗ МЕТОДОВ БИБЛИОТЕКИ LINQ МОЖНО СОСТАВЛЯТЬ ЦЕПОЧКИ

Методы библиотеки реализованы как методы расширяющие тип IEnumerable<T> и возвращают также тип IEnumerable<T>, а значит выходное значение методы можно использовать как входное для следующего метода обработчика в цепочке. Пример на рисунке.

```
ссылок: 4
class User
{
    ссылок: 5
    public string Name { get; set; }
    ссылок: 5
    public int Age { get; set; }

}

ссылок: 0
static void Main(string[] args)
{
    var source = new User[]
    {
        new User { Name = "Name1", Age = 10 },
        new User { Name = "Name2", Age = 10 },
        new User { Name = "Name3", Age = 10 }
    };
    string[] names = source
        .Where(u => u.Age > 0)
        .OrderBy(u => u.Name)
        .ThenBy(u => u.Age)
        .Select(u => u.Name)
        .ToArray();
}
```