

# Лекция 4

---

# Работа с наборами данных

- Как хранить и обрабатывать наборы данных?

Массивы

Коллекции

# Массивы

- Ограничивается доступным размером памяти\*
- Размер массива должен быть указан при его создании.
- Массивы могут хранить как ссылочные типы, так и типы значений.

# Массивы

- Массив является **индексированной** коллекцией объектов.
- Одномерный массив объектов объявляется следующим образом.  
`type[] arrayName;`

# Многомерный массив

- Концептуально, многомерный массив с двумя измерениями напоминает сетку (таблицу).
- Многомерный массив с тремя измерениями напоминает куб.

```
type[, ] arrayName;
```

# Массив массивов

- Одним из вариантов многомерного массива является массив массивов. Массив массивов представляет собой **одномерный массив**, в котором **каждый элемент** является **массивом**. Элементы массива не обязаны иметь одинаковый размер.

```
type[][] arrayName;
```

# Массив массивов

Шаг 1: выделяем память под одномерный массив.

```
int[][] jaggedArray = new int[3][];
```

Шаг 2: Для каждого элемента одномерного массива выделяем память под одномерный массив

```
jaggedArray[0] = new int[5];
```

```
jaggedArray[1] = new int[4];
```

```
jaggedArray[2] = new int[2];
```

# Массив объектов

- Создание массива объектов в отличие от создания массива простых типов данных происходит в два этапа.
- Сначала необходимо объявить массив.
- А затем создать объекты для хранения в нем.



# Массив объектов

- Создадим класс

```
class Class1 {int x;}
```

Теперь создадим массив от этого класса

```
Class1[] mas = new Class1[10];
```

```
for(int i = 0; i < 10; ++i)
```

```
{
```

```
    mas[i] = new Class1();
```

```
}
```

# Другие коллекции

- Рассмотрим другие коллекции, часто используемые в программировании:

ArrayList

List

Dictionary

Queue

Stack

# Библиотеки с коллекциями

- Большая часть классов коллекций содержится в пространствах имен **System.Collections**, **System.Collections.Generic** и **System.Collections.Specialized**.
- Также для обеспечения параллельного выполнения задач и многопоточного доступа применяются классы коллекций из пространства имен **System.Collections.Concurrent**

# Основа коллекций

- Основой для создания всех коллекций является реализация интерфейсов **IEnumerator** и **IEnumerable**.

# IEnumerator

- Интерфейс **IEnumerator** представляет **перечислитель**, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле `foreach`.

# IEnumerable

- Интерфейс **IEnumerable** через свой метод `GetEnumerator` предоставляет перечислитель всем классам, реализующим данный интерфейс.
- Поэтому интерфейс `IEnumerable` (`IEnumerable<T>`) является базовым для всех коллекций.

# ArrayList

- Реализует интерфейс IList с помощью массива с **динамическим изменением размера** по требованию.

# Варианты создания

<code>ArrayList()</code>	Инициализирует новый экземпляр класса <code>ArrayList</code> , который является пустым и имеет начальную емкость по умолчанию.
<code>ArrayList(Collection)</code>	Инициализирует новый экземпляр класса <code>ArrayList</code> , который содержит элементы, скопированные из указанной коллекции, и обладает начальной емкостью, равной количеству скопированных элементов.
<code>ArrayList(int)</code>	Инициализирует новый пустой экземпляр класса <code>ArrayList</code> с указанной начальной емкостью.



# Часто используемые свойства

Capacity	Возвращает или задает число элементов, которое может содержать список ArrayList.
Count	Возвращает число элементов, содержащихся в ArrayList.
Item[Int32]	Возвращает или задает элемент по указанному индексу.

# Часто используемые методы

Add(Object)	Добавляет объект в конец очереди ArrayList.
AddRange(ICollection)	Добавляет элементы интерфейса ICollection в конец списка ArrayList.
Clear()	Удаляет все элементы из коллекции ArrayList.
Contains(Object)	Определяет, входит ли элемент в коллекцию ArrayList.
Insert(Int32, Object)	Вставляет элемент в коллекцию ArrayList по указанному индексу.
Remove(Object)	Удаляет первое вхождение указанного объекта из коллекции ArrayList.
RemoveAt(Int32)	Удаляет элемент списка ArrayList с указанным индексом.

# Пример

```
ArrayList array = new ArrayList();  
array.Add("Hello");  
array.Add('l');  
array.Add(1);
```

```
Console.WriteLine("  Count: {0}", array.Count);  
Console.WriteLine("  Capacity: {0}", array.Capacity);  
Console.Write("  Values:");  
for(int i = 0; i < array.Count; ++i)  
{  
    Console.Write("  {0}", array[i]);  
}
```

# List

- Представляет **строго типизированный** список объектов, доступных по индексу.
- Класс **List<T>** является универсальным эквивалентом класса `ArrayList`. Он реализует универсальный интерфейс **IList<T>** с помощью массива, размер которого динамически увеличивается по мере необходимости.

# Что лучше

- Делая выбор между классами `List<T>` и `ArrayList`, предлагающими сходные функциональные возможности, следует помнить, что класс `List<T>` в большинстве случаев **обрабатывается быстрее** и является **потокобезопасным**. Если в качестве типа `T` класса `List<T>` используется **ссылочный** тип, оба класса действуют **идентичным** образом.

# Варианты создания

`List<T>()`

Инициализирует новый экземпляр класса `List<T>`, который является пустым и имеет начальную емкость по умолчанию.

`List<T>(IEnumerable<T>)`

Инициализирует новый экземпляр `List<T>`, который содержит элементы, скопированные из указанной коллекции, и имеет емкость, достаточную для размещения всех скопированных элементов.

`List<T>(int)`

Инициализирует новый пустой экземпляр класса `List<T>` с указанной начальной емкостью.

# Часто используемые свойства

Capacity	Возвращает или задает общее число элементов, которые может вместить внутренняя структура данных без изменения размера.
Count	Получает число элементов, содержащихся в интерфейсе List<T>.
Item[Int32]	Возвращает или задает элемент по указанному индексу.

# Часто используемые методы

Add(T)	Добавляет объект в конец очереди List<T>.
AddRange(IEnumerable<T>)	Добавляет элементы указанной коллекции в конец списка List<T>.
Clear()	Удаляет все элементы из коллекции List<T>.
Contains(T)	Определяет, входит ли элемент в коллекцию List<T>.
Insert(Int32, T)	Вставляет элемент в коллекцию List<T> по указанному индексу.
Remove(T)	Удаляет первое вхождение указанного объекта из коллекции List<T>.
RemoveAt(Int32)	Удаляет элемент списка List<T> с указанным индексом.



# Пример List

```
List<string> cars = new List<string>();  
cars.Add("BMW");  
cars.Add("Mercedes");  
cars.Add("Ford Mustang");  
cars.Add("Corvette");  
cars.Add("Jaguar");  
  
for (int i = 0; i < cars.Count; ++i)  
{  
    Console.WriteLine("  {0}", cars[i]);  
}  
Console.ReadKey();
```

# Queue

- Представляет коллекцию объектов, основанную на принципе "**первым вошёл — первым вышел**". (FIFO)
- Добавление элементов происходит в конец списка. Извлечение из начала списка.

# Варианты создания

`Queue<T>()`

Инициализирует новый экземпляр класса `Queue<T>`, который является пустым и имеет начальную емкость по умолчанию.

`Queue<T>(IEnumerable<T>)`

Инициализирует новый экземпляр `Queue<T>`, который содержит элементы, скопированные из указанной коллекции, и имеет емкость, достаточную для размещения всех скопированных элементов.

`Queue<T>(int)`

Инициализирует новый пустой экземпляр класса `Queue<T>` с указанной начальной емкостью.

# Часто используемые свойства

Count

Получает число элементов, содержащихся в интерфейсе `Queue<T>`.

# Часто используемые методы

Clear()	Удаляет все объекты из Queue<T>.
Contains(T)	Определяет, входит ли элемент в коллекцию Queue<T>.
Dequeue()	Удаляет объект из начала очереди Queue<T> и возвращает его.
Enqueue(T)	Добавляет объект в конец очереди Queue<T>.
Peek()	Возвращает объект, находящийся в начале очереди Queue<T>, но <b>не удаляет его</b> .

# Как она реализована

- Этот класс реализует универсального очередь в виде **циклического массива**. Объекты, хранящиеся в `Queue<T>` **вставляются с одной стороны**, и **извлекаются с другой**.

# Пример

```
Queue<string> numbers = new Queue<string>();  
numbers.Enqueue("one");  
numbers.Enqueue("two");  
numbers.Enqueue("three");  
numbers.Enqueue("four");  
numbers.Enqueue("five");
```

```
Console.WriteLine(numbers.Peek());  
Console.WriteLine(numbers.Dequeue());  
Console.WriteLine(numbers.Dequeue());  
Console.WriteLine(numbers.Peek());  
Console.WriteLine(numbers.Peek());
```

# Stack

- Представляет коллекцию переменного размера экземпляров одинакового заданного типа, обслуживаемую по принципу "**последним пришел - первым вышел**" (LIFO).
- Это означает, что новый элемент вставляется в начало и извлекается из начала.



# Варианты создания

`Stack<T>()`

Инициализирует новый экземпляр класса `Stack<T>`, который является пустым и имеет начальную емкость по умолчанию.

`Stack<T>(IEnumerable<T>)`

Инициализирует новый экземпляр `Stack<T>`, который содержит элементы, скопированные из указанной коллекции, и имеет емкость, достаточную для размещения всех скопированных элементов.

`Stack<T>(int)`

Инициализирует новый экземпляр `Stack<T>` класс, который является пустым и обладает указанной начальной емкостью или начальной емкостью по умолчанию, какое значение больше.

# Часто используемые свойства

Count

Получает число элементов, содержащихся в интерфейсе `Stack<T>`.

# Часто используемые методы

Clear()	Удаляет все объекты из Stack<T>.
Contains(T)	Определяет, входит ли элемент в коллекцию Stack<T>.
Peek()	Возвращает объект в верхней части Stack<T> без его удаления.
Pop()	Удаляет и возвращает объект в верхней части Stack<T>.
Push(T)	Вставляет объект как верхний элемент стека Stack<T>.

# Пример

```
Stack<string> numbs = new Stack<string>();  
numbs.Push("one");  
numbs.Push("two");  
numbs.Push("three");  
numbs.Push("four");  
numbs.Push("five");
```

```
Console.WriteLine(numbs.Peek());  
Console.WriteLine(numbs.Pop());  
Console.WriteLine(numbs.Pop());  
Console.WriteLine(numbs.Peek());  
Console.WriteLine(numbs.Peek());
```

# Что-когда используется?

- Очереди и стеки полезны, когда требуется временное хранилище для данных;
- Очередь `Queue<T>` используют, когда необходимо получить доступ к данным в том же порядке, в котором их сохраняют.
- Стек `Stack<T>` используют, когда требуется доступ к данным в **обратном** порядке.

# Dictionary

- Ассоциативная коллекция. Представляет собой набор пар **ключ-значение**.

# Варианты создания (не все!)

`Dictionary<TKey, TValue>()`

Инициализирует новый пустой экземпляр класса `Dictionary<TKey, TValue>`, имеющий начальную емкость по умолчанию и использующий функцию сравнения по умолчанию, проверяющую равенство для данного типа ключа.

`Dictionary<TKey, TValue>(Int32)`

Инициализирует новый пустой экземпляр класса `Dictionary<TKey, TValue>`, имеющий заданную начальную емкость и использующий функцию сравнения по умолчанию, проверяющую равенство для данного типа ключа.

# Часто используемые свойства

Count	Возвращает число пар "ключ-значение", содержащихся в словаре Dictionary<TKey, TValue>.
Item[TKey]	Возвращает или задает значение, связанное с указанным ключом.
Keys	Возвращает коллекцию, содержащую ключи из словаря Dictionary<TKey, TValue>.
Values	Возвращает коллекцию, содержащую значения из словаря Dictionary<TKey, TValue>.



# Часто используемые методы

Add(TKey, TValue)	Добавляет указанные ключ и значение в словарь.
Clear()	Удаляет все ключи и значения из словаря Dictionary<TKey, TValue>.
ContainsKey(TKey)	Определяет, содержится ли указанный ключ в словаре Dictionary<TKey, TValue>.
ContainsValue(TValue)	Определяет, содержит ли коллекция Dictionary<TKey, TValue> указанное значение.
Remove(TKey)	Удаляет значение с указанным ключом из Dictionary<TKey, TValue>.

# Словарь

- Dictionary<TKey, TValue> Универсальный класс предоставляющий сопоставление из набора ключей для набора значений.
- **Каждый ключ** в Dictionary<TKey, TValue> должно быть **уникальным**

# Пример

```
Dictionary<string, string> dict = new  
Dictionary<string, string>();  
dict.Add("txt", "notepad.exe");  
dict.Add("bmp", "paint.exe");  
dict.Add("dib", "paint.exe");  
dict.Add("rtf", "wordpad.exe");
```

```
foreach (KeyValuePair<string, string> kvp in dict)  
{  
    Console.WriteLine("Key = {0}, Value = {1}",  
kvp.Key, kvp.Value);  
}
```

# foreach

- Оператор `foreach` **повторяет** группу вложенных операторов для каждого элемента массива или коллекции объектов, реализующих интерфейс `System.Collections.IEnumerable` или `System.Collections.Generic.IEnumerable<T>`.

# foreach

```
foreach(<тип элемента> <имя элемента> in  
      <имя коллекции>)
```

- Нельзя использовать, если требуется изменять размер коллекции (добавлять или удалять из нее элементы)!

# Пример

```
foreach (var kvp in dict)
{
    Console.WriteLine("Key = {0}, Value = {1}",
kvp.Key, kvp.Value);
}
```

```
foreach(var car in cars)
{
    Console.Write(" {0}", car);
}
```

# Индексаторы

- Индексаторы позволяют индексировать экземпляры класса или структуры точно так же, **как и массивы**.
- Индексаторы напоминают **свойства** за исключением того, что их методы доступа **принимают параметры**.

# Пример

- Создадим класс, в нем массив (нам же нужно будет откуда-то брать элементы) и индексатор

```
class Class5
{
    int[] arr;

    0 references
    public int this[int i]
    {
        set {arr[i] = value;}
        get {return arr[i];}
    }
}
```



# Пример 2

- Индексатор может принимать более одного параметра

```
class Class5
{
    int[,] arr2D;

    0 references
    public int this[int i, int j]
    {
        set { arr2D[i, j] = value; }
        get { return arr2D[i, j]; }
    }
}
```

# Как сделать не получится

- Но реализовать индексатор типа «массив массивов» нельзя

```
class Class5
{
    int[][] arrArr;

    0 references
    public int this[int i][int j]
    {
        set { arrArr[i][j] = value; }
        get { return arrArr[i][j]; }
    }
}
```

# Пример

- Не забудем про конструктор (к слову, зачем он нужен)

```
class Class5
{
    0 references
    public Class5(int a, int b, int c)
    {
        arr = new int[a];
        arr2D = new int[b, c];
    }
}
```

# Как использовать

- Поработаем с одномерным индексатором

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Class5 cl = new Class5(10, 10, 10);

        for (int i = 0; i < 10; ++i )
        {
            cl[i] = i * i;
        }

        for (int i = 0; i < 10; ++i)
        {
            Console.Write("{0} ", cl[i]);
        }
    }
}
```

# Как использовать

## ■ И двумерный

```
for (int i = 0; i < 10; ++i)
{
    for (int j = 0; j < 10; ++j)
    {
        cl[i, j] = i * j;
    }
}

Console.WriteLine();
for (int i = 0; i < 10; ++i)
{
    for (int j = 0; j < 10; ++j)
    {
        Console.Write("{0, 2} ", cl[i]);
    }
    Console.WriteLine();
}

Console.ReadKey();
}
}
```