

Collections

- Collections
 - Array
 - Control flow
 - for
 - foreach, IEnumerable
 - yield
 - Рекомендации
 - Generic collections
 - Interfaces
 - List<T>
 - Dictionary<TKey,TValue>
 - SortedList vs SortedDictionary
 - HashSet
 - Равенство TBD
 - IEquatable<T>
 - IEqualityComparer<T>
 - IComparable<T>

Array

- нумерация с нуля

```
int[] array = new int[3];
int[] array2 = new int[3] { 1, 3, 9 }; // Инициализатор
int[] array3 = new int[] { 1, 3, 9 };
int[] array4 = new[] { 1, 3, 9 };
int[] array5 = { 1, 3, 9 };

array[0] = 1;
array[1] = 2;
array[2] = 3;
array[3] = 3; // throw IndexOutOfRangeException (проверка обычно выполняется 1 раз
за цикл и не влияет на производительность)
```

Многомерные массивы:

```
int[] mdarray = new int[] { 0, 1, 2, 3 };  
int[,] mdarray2 = { { 0, 1, 2 }, { 2, 1, 0 } };  
int[,,] mdarray3 = new int[2, 3, 2];
```

jagged array:

```
int[][] jagged = new int[2][];  
jagged[0] = new int[3] { 1, 2, 3 };  
jagged[1] = new int[3] { 7, 7, 7 };
```

```
int[][] jagged = new int[2][];  
jagged[0] = new int[3] { 1, 2, 3 };  
jagged[1] = new int[3] { 7, 7, 7 };  
  
foreach(int[] row in jagged)  
{  
    foreach(int value in row)  
    {  
        Console.Write($"{value} ");  
    }  
    Console.WriteLine();  
}  
for (int i = 0; i < jagged.Length; i++)  
{  
    for (int k = 0; k < jagged[i].Length; k++)  
    {  
        Console.Write($"{jagged[i][k]} ");  
    }  
    Console.WriteLine();  
}
```

Control flow

Циклы:

- `for`
- `foreach`
- `do`
- `while`

Внутри циклов:

- `continue` - следующая итерация цикла
- `break` - выйти из цикла

Условные операторы:

- `if { } else { }`
- `switch`
- ternary operator

for

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Value of i: {0}", i);
}

int i = 0;

for(;;)
{
    if (i < 10)
    {
        Console.WriteLine("Value of i: {0}", i);
        i++;
    }
    else
        break;
}
```

```
int i = 0;

while (true)
{
    Console.WriteLine("Value of i: {0}", i);
    i++;
    if (i > 10)
        break;
}

i = 0;

do
{
    Console.WriteLine("Value of i: {0}", i);
    i++;
} while (i < 10);
```

foreach, IEnumerable

```
foreach (int x in src)
{
    // Do something with x.
}

public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }

    void Dispose();
    bool MoveNext();
    void Reset(); // Для совместимости с COM, для нового цикла enumerator создается заново
}
```



```

foreach (int x in src)
{
    // Do something with x.
}

var e = src.GetEnumerator();
while (e.MoveNext()) // вначале Current кидает ошибку, необходимо выполнить
MoveNext
{
    var x = (int)e.Current; // without the cast if src was an IEnumerable<T>
    // Do something with x.
}

using (var e = src.GetEnumerator())
{
    while (e.MoveNext())
    {
        int x = e.Current;
        // Do something with x.
    }
}

```

```
public class BoxEnumerator : IEnumerator<Box>
{
    private BoxCollection _collection;
    private int curIndex;
    private Box current;

    public BoxEnumerator(BoxCollection collection)
    {
        _collection = collection;
        curIndex = -1;
        current = default(Box);
    }

    public bool MoveNext()
    {
        //Avoids going beyond the end of the collection.
        if (++curIndex >= _collection.Count)
            return false;

        current = _collection[curIndex];
        return true;
    }
}
```

```
public void Reset() { curIndex = -1; }  
  
void IDisposable.Dispose() { }  
  
public Box Current { get { return current; } }  
  
object IEnumerator.Current { get { return Current; } }  
}
```

- всегда явно использовать `foreach`
- обычно енуератор считается валидным пока коллекция не изменялась
- обычно при удалении, добавлении, элементов методы `MoveNext` or `Reset` должны кидать `InvalidOperationException`
- в дефолтной ситуации перебор коллекции непотокобезопасен и надо делать собственную синхронизацию

```
public class List<T> : IEnumerable<T>
{
    // Итератор класса List<T>, Это структура, причем изменяемая!!!
    public struct Enumerator : IEnumerator<T>, IDisposable
    { }

    public List<T>.Enumerator GetEnumerator() { return new Enumerator(this); }

    // Явная реализация интерфейса
    IEnumerator<T> IEnumerator<T>.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

```
var list = new List<int> {1, 2, 3};

// Вызываем List<T>.Enumerator GetEnumerator
foreach(var i in list)
{}

// Вызываем IEnumerable<T> GetEnumerator
foreach(var i in (IEnumerable<int>)list)
{}

// Пример интересного поведения связанного с изменяемыми структурами
var x = new { Items = new List<int> { 1, 2, 3 }.GetEnumerator() };
while (x.Items.MoveNext())
{
    Console.WriteLine(x.Items.Current);
}
```

yield

- итераторный блок `yield` можно использовать только в методе, возвращающем `IEnumerable`, `IEnumerator` или обобщенные эквиваленты
- внутри итераторного блока запрещены обычные `return`
- создает **автомат состояний**, который фактически генерит enumerator поверх метода
- при `yield return` приостанавливает выполнение кода, фактически завершая метод `MoveNext()` + `Current` и отдавая управление коду, использующему итератор

```
static IEnumerable<int> CreateEnumerable()
{
    for (int i = 0; i < 3; i++)
    {
        yield return i;
        if (DateTime.Now >= limit)
            yield break;
    }
    yield return -1;
}

foreach(int i in CreateEnumerable()) { .... }
```

```
for (DateTime day = timetable.StartDate; day <= timetable.EndDate; day =  
day.AddDays(1))  
{  
  
foreach (DateTime day in timetable.DateRange)  
{  
  
public IEnumerable<DateTime> DateRange  
{  
    get  
    {  
        for (DateTime day = StartDate; day <= EndDate; day = day.AddDays(1))  
        {  
            yield return day;  
        }  
    }  
}
```



```
static IEnumerable<int> CountWithTimeLimit(DateTime limit)
{
    try
    {
        for (int i = 1; i <= 100; i++)
        {
            if (DateTime.Now >= limit)
                yield break;
            yield return i;
        }
    }
    finally
    {
        Console.WriteLine("Stopping!");
    }
}
```

Ограничения (Lippert [blog](#))

- Оператор `yield return` не разрешено использовать внутри блока `try` при наличии любых блоков `catch`
- не допускается применять оператор `yield return` или `yield break` в блоке `finally`
- не допускаются `out/ref` параметры, в анонимных методах

Рекомендации

- Используйте `foreach` везде, где удобно
- Если вдруг будете реализовывать еnumератор - делайте это проще, без структур и полагания на duck typing
- Используйте итераторный блок, если это делает код красивее и проще

Generic collections

Interfaces

Интерфейсы, имеющие отношение к коллекциям

- **IEnumerable<T>** - доступ к enumerator, для последовательного доступа к элементам через foreach
- **IEnumerator<T>**
- **ICollection<T>** - общие свойства и методы для всех коллекций (CopyTo, Add, Remove, Contains, свойство Count)
- **IList<T>** - последовательный список
- **IDictionary<TKey, TValue>** - интерфейс для коллекции, хранящей объекты в виде пар ключ-значение

System.Collections.Generic

Главные коллекции:

- `List<T>` - последовательный список
- `Dictionary<TKey, TValue>` - набор уникальных пар "ключ-значение"
- `HashSet<T>` - высокопроизводительный список
- `SortedSet<T>` - отсортированная коллекция
- `SortedList<TKey, TValue>` - коллекция, хранящая пары "ключ-значение", отсортированные по ключу
- `SortedDictionary<TKey, TValue>` отличия в реализации, использовании памяти, скорости выполнения отдельных методов
- `Stack<T>` LIFO
- `Queue<T>` - очередь объектов FIFO
- `LinkedList<T>` - двухсвязанный список

List<T>

Аналог Vector в C++

- Random access - constant $O(1)$
- Insertion or removal of elements at the end - amortized constant $O(1)$
- Insertion or removal of elements - linear in the distance to the end of the vector $O(n)$

```
public class List: IList<T>, ICollection<T>, IEnumerable<T>

// Некоторые методы
void Add(T item);                // добавление нового элемента
void AddRange(ICollection collection); // добавление коллекции
int IndexOf(T item);            // находит индекс первого вхождения
void Insert(int index, T item);  // вставляет элемент на позицию index
bool Remove(T item);            // удаляет элемент из списка (return true,
если успешно)
void RemoveAt(int index);        // удаление элемента по указанному индексу
index
void Sort();                     // сортировка
int BinarySearch(T item);        // бинарный поиск (возвращает индекс, если
нашел). Список должен быть отсортирован
```

```
List<int> list = new List<int>() { 1, 22, 3, 4, 5 };

list.Add(6);
list.AddRange(new int[] { 3, 2 }); // Можно добавлять коллекцию
list.RemoveAt(1);                  // Удаляем второй элемент
list.Insert(0, 33);                 // вставляем на первое место
list[2] = 2;                       // Можно по индексу обращаться

foreach (int i in numbers)
{
    Console.WriteLine(i);
}
```

Dictionary<TKey,TValue>

```
public class Dictionary<TKey, TValue>: IDictionary<TKey, TValue>,  
    ICollection<KeyValuePair<TKey, TValue>>,  
    IEnumerable<KeyValuePair<TKey, TValue>>,  
    IEnumerable, IDictionary, ICollection
```

- Hashtable
- Dictionary is $O(1)$ and in worst case $O(\log(N))$
- По факту содержит внутри 3 коллекции (пары, **Keys**, **Values**)
- Содержит перечисление **KeyValuePair<TKey, TValue>**

Properties:

- **Comparer** - `IEqualityComparer<T>` объект сравнивающий ключи на равенство
- **Count** - количество пар
- **Item[TKey]** - Get / set для значения по ключу
- **Keys** - коллекция ключей
- **Values** - коллекция значений

Methods:

```
void Add(TKey, TValue);  
void Clear();  
bool ContainsKey(TKey key);           // Есть ли ключ в коллекции  
bool ContainsValue(TValue value);     // Есть ли значение  
bool Remove(TKey key);                // удалить пару  
bool TryGetValue(TKey key, out TValue value); // попробовать получить значение по  
ключу
```



```
Dictionary<string, string> openWith = new Dictionary<string, string>();

openWith.Add("txt", "notepad.exe");
openWith.Add("bmp", "paint.exe");
openWith.Add("dib", "paint.exe");
openWith.Add("rtf", "wordpad.exe");

try
{
    openWith.Add("txt", "winword.exe");
}
catch (ArgumentException)
{
    Console.WriteLine("An element with Key = \"txt\" already exists.");
}
```

Итерирование:

```
foreach(KeyValuePair<string, string> item in openWith)
{
    foo(item.Key);
    bar(item.Value);
}

foreach(var item in openWith.Keys)
{
    foo(item);
}

foreach(var item in openWith.Values)
{
    foo(item);
}
```

SortedList vs SortedDictionary

- И то и другое binary search tree with $O(\log n)$ retrieval
- Похожие объектные модели
- Разница в использовании памяти, скорости вставки и удаления:
 - `SortedList<TKey, TValue>` использует меньше памяти
 - `SortedDictionary<TKey, TValue>` - быстрее вставка и удаление для несортированных данных - $O(\log n)$, вместо $O(n)$ у `SortedList<TKey, TValue>`
 - `SortedList<TKey, TValue>` быстрее, если вставка идет одним куском из сортированных данных

SOF

HashSet

<http://theburningmonk.com/2011/03/hashset-vs-list-vs-dictionary/>

Типичное применение:

- Добавляем данные нечасто
- Нужен $O(1)$ доступ к элементам

```
var sqlErrorCodes = new HashSet<int>({});
```

```
if (sqlErrorCode.Contains(someCode))  
{  
}
```

Равенство

TBD

<https://msdn.microsoft.com/ru-ru/library/ms173147%28v=vs.90%29.aspx?f=255&MSPPError=-2147217396>

<https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/statements-expressions-operators/how-to-define-value-equality-for-a-type>

<https://habrahabr.ru/post/188038/>

Equals

GetHashCode

IEquatable<T>

TBD

<https://docs.microsoft.com/ru-ru/dotnet/api/system.ieuatable-1?view=netframework-4.7.1>

`IEqualityComparer<T>`

TBD

[https://msdn.microsoft.com/en-us/library/ms132151\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms132151(v=vs.110).aspx)

IComparable<T>

TBD

<https://msdn.microsoft.com/en-us/library/4d7sx9hd.aspx>