ЛЕКЦИЯ 13-14

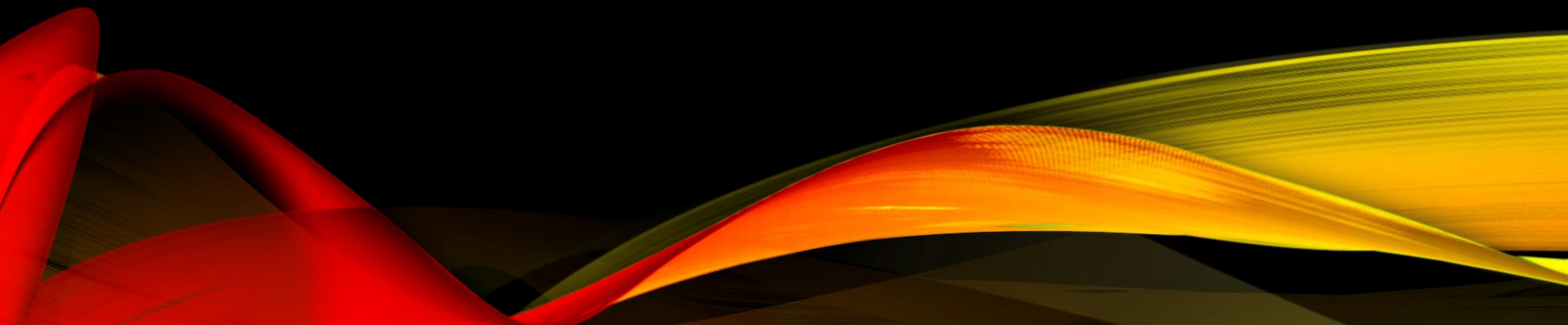
- Модуль 3
- 21.02.2024
- Итераторы и перечислимые коллекции

ЦЕЛИ ЛЕКЦИИ

- Разобраться с механикой работы foreach
- Познакомиться с понятиями итерируемой коллекции и итератора
- Изучить варианты реализации итерируемых / перечислимых коллекций в C#
- Запись https://disk.yandex.ru/d/qd3U_k1TUxJlmw

FOREACH

Механика работы



ИСПОЛЬЗОВАНИЕ ОПЕРАТОРА FOREACH

```
foreach (<тип> имяПеременной in имяКоллекции)
{
    // Тело foreach.
}
```

Элементы коллекции имяКоллекции имеют тип <тип>

```
int[] arr = { 10, 11, 12, 13 };
foreach (int item in arr)
{
    Console.WriteLine($"Item value: {item}");
}
```

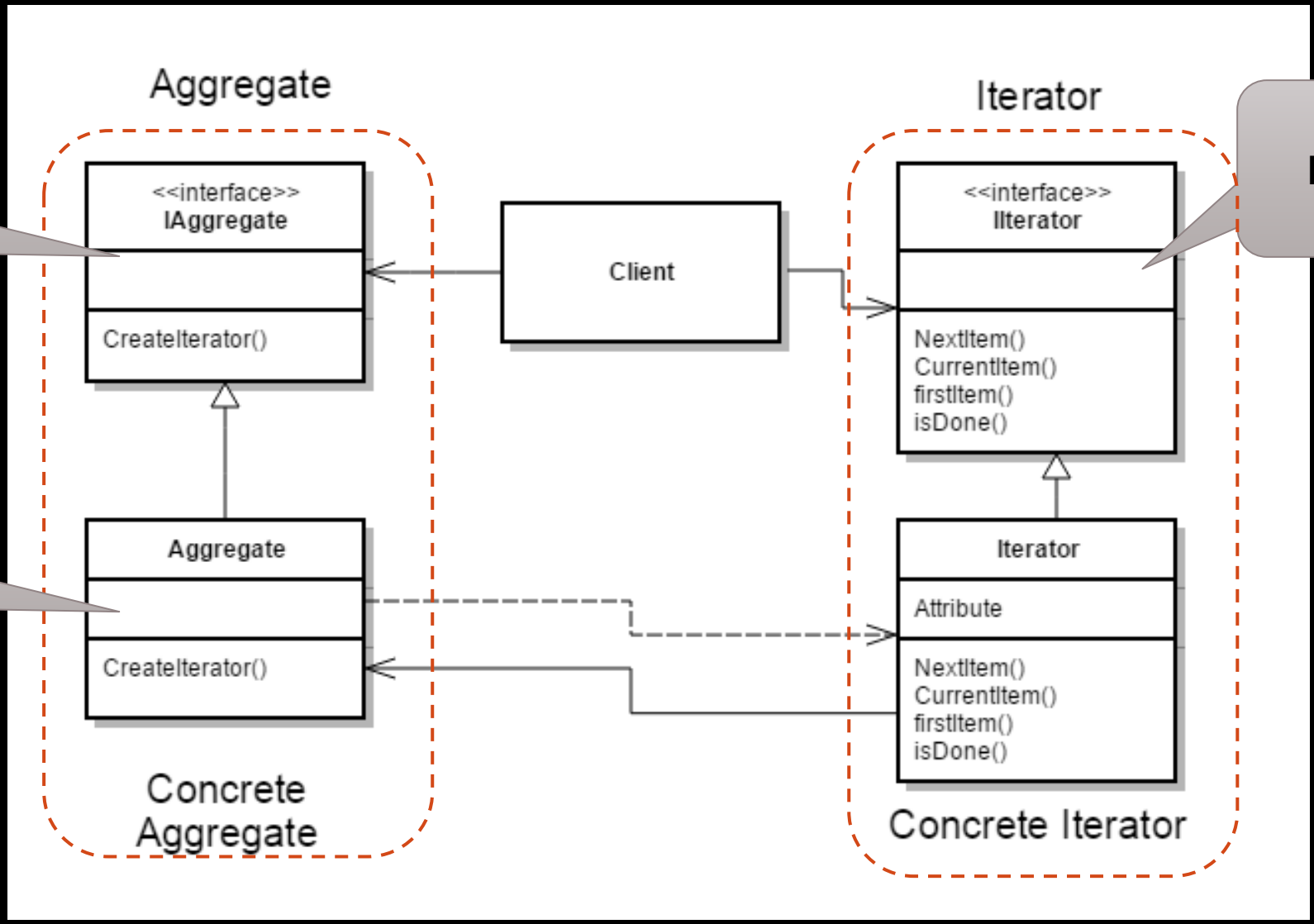
ШАБЛОН ИТЕРАТОР

Шаблон итератор – поведенческий шаблон проектирования, при котором реализуется специальный объект для последовательного доступа к элементам объекта-агрегата, не раскрывая его внутреннего представления клиентскому коду

ШАБЛОН ИТЕРАТОР

IEnumerable

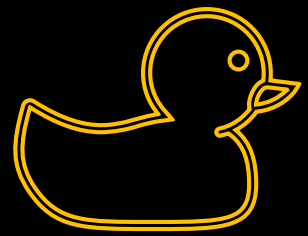
Агрегат (в нашем случае коллекция)



IEnumerator

СПОСОБЫ СОЗДАНИЯ ИТЕРИРУЕМЫХ КОЛЛЕКЦИЙ

- Интерфейсы **IEnumerator / IEnumerable**
- Обобщённые интерфейсы **IEnumerator<T> / IEnumerable<T>**
- Конструкция, в которой интерфейсы не применяются (утиная типизация)



ИТЕРАТОРЫ

Итераторы .NET однонаправленны и используются только для чтения

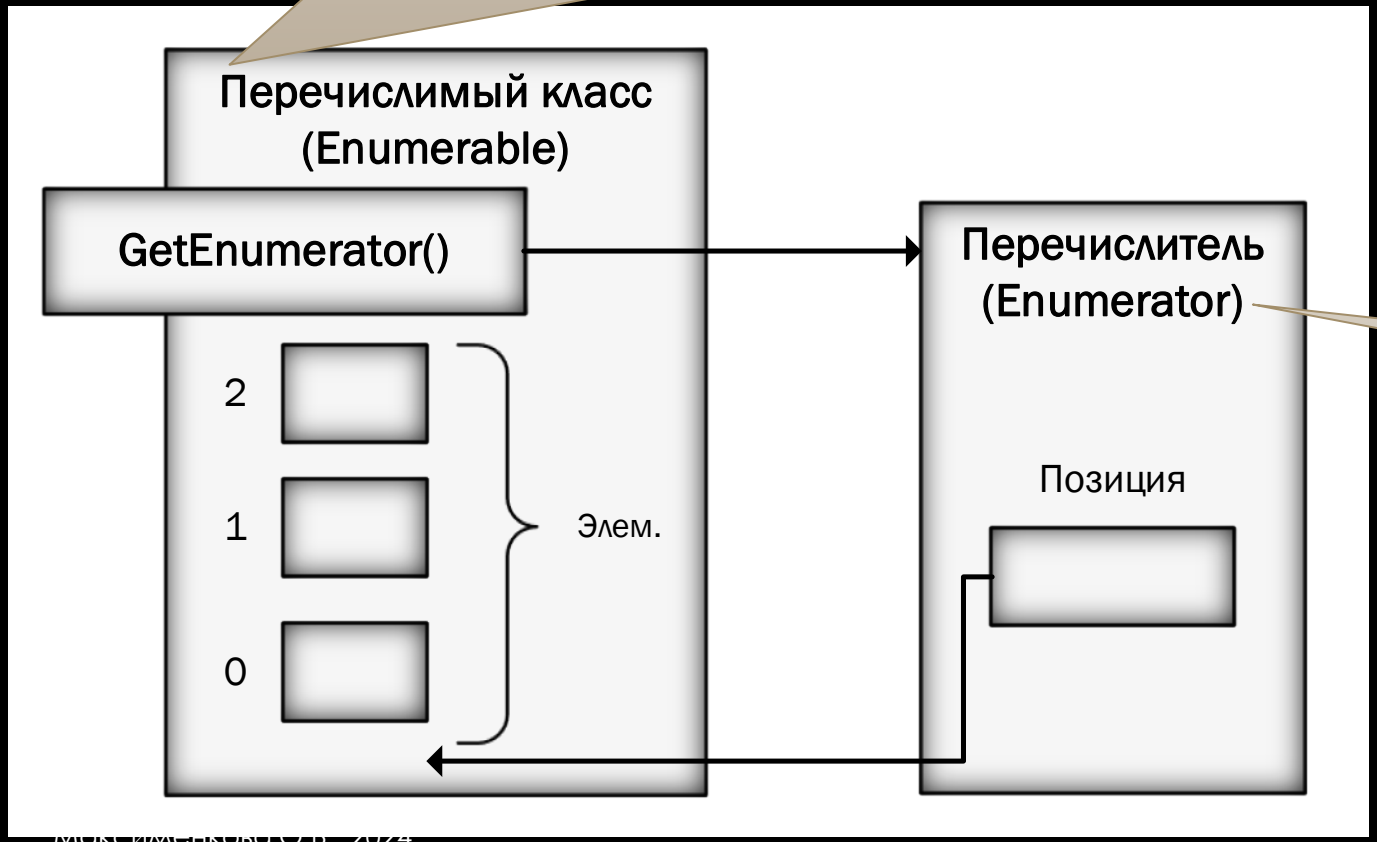
Чтобы получить экземпляр итератора, вызывается метод `GetEnumerator()` интерфейса `IEnumerable` (каждый раз возвращается новый экземпляр итератора)

Члены интерфейса IEnumerator:

- `object Current { get; }` — возвращает текущий элемент агрегата
- `bool MoveNext()` — осуществляет переход к следующему элементу агрегата, false возвращает, если достигнут конец последовательности
- `void Reset()` — устанавливает итератор в начало агрегата

ИТЕРИРУЕМАЯ КОЛЛЕКЦИЯ И ИТЕРАТОР

Перечислимый класс (итерируемая коллекция) – класс, реализующий метод **`GetEnumerator()`**, который возвращает **перечислитель** для своих элементов



`GetEnumerator()` возвращает экземпляр итератора/перечислителя **(`Enumerator`)**

Перечислитель (итератор) – объект, который может вернуть каждый элемент коллекции, по порядку

```
class Iterator
{
    private DemoCollection? collection;
    private int currentIndex;
    internal Iterator(DemoCollection? collection = null)
    {
        if (collection is null) { throw new NullReferenceException(); }
        this.collection = collection;
        currentIndex = -1;
    }
    public int Current
    {
        get
        {
            if(currentIndex == -1 || currentIndex == collection!.Count)
            {
                throw new IndexOutOfRangeException();
            }
            return collection[currentIndex];
        }
    }
    public bool MoveNext()
    {
        if(currentIndex != collection!.Count) { currentIndex++; }
        return currentIndex < collection.Count;
    }
 }
```

```
class DemoCollection  {
    int[] data = { 5, 6, 7 };
    public int this[int index] { get => data[index]; }
    public int Count => data.Length;
    public Iterator GetEnumerator() { return new Iterator(this); }
}
```

```
DemoCollection dc = new DemoCollection();
foreach(var item in dc)
{
    Console.WriteLine($"{item} ");
}
```

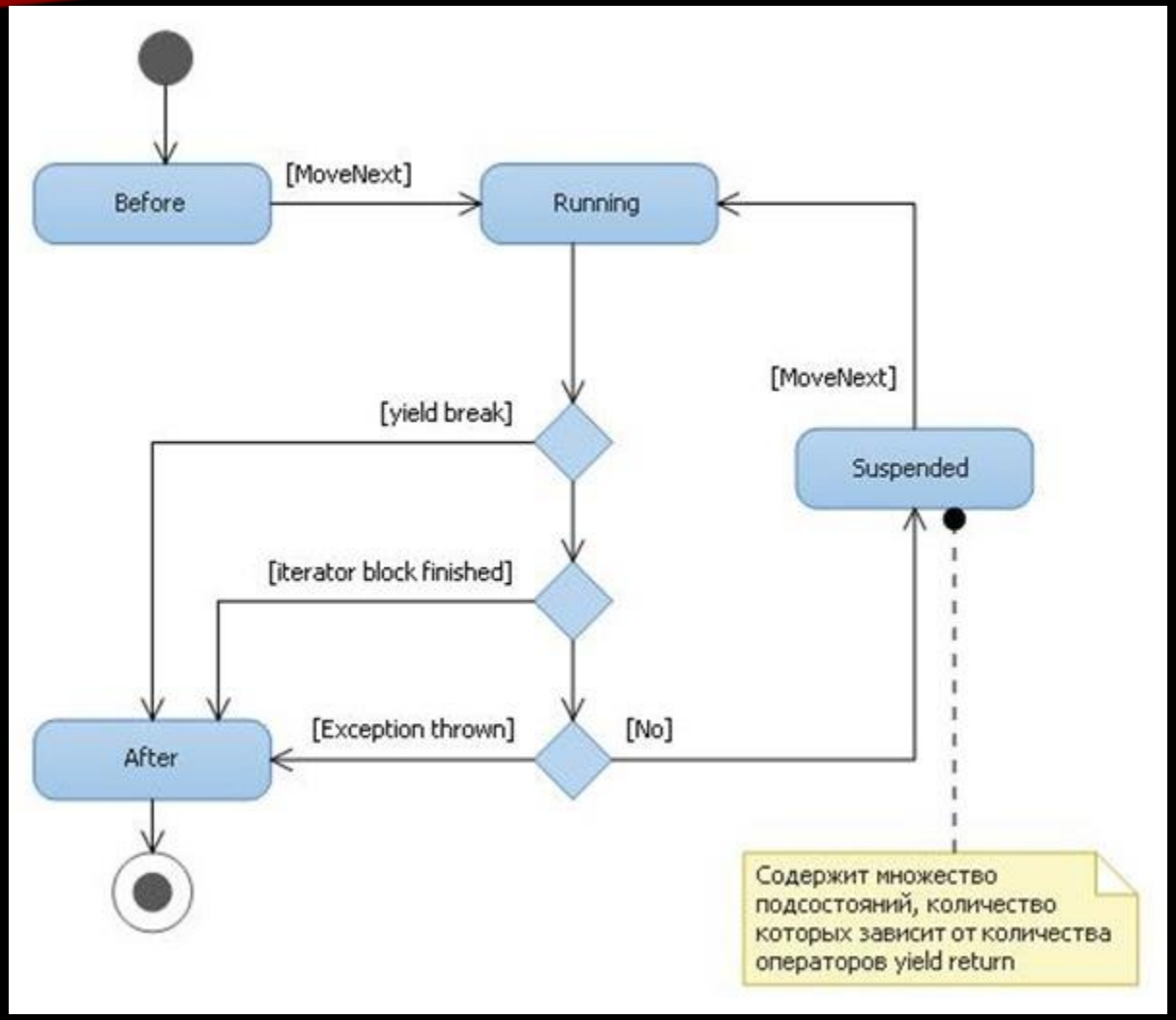
СОВЕРШЕНСТВУЕМ КОД НА ОСНОВЕ ITERATOR BLOCK

```
class DemoCollection
{
    int[] data = { 5, 6, 7 };
    public int Count => data.Length;
    // Using Iterator Block.
    public IEnumerator<int> GetEnumerator()
    {
        for (int i = 0;i < data.Length;i++)
        {
            yield return data[i];
        }
    }
}
```

yield return применяется для предоставления следующего значения итератора

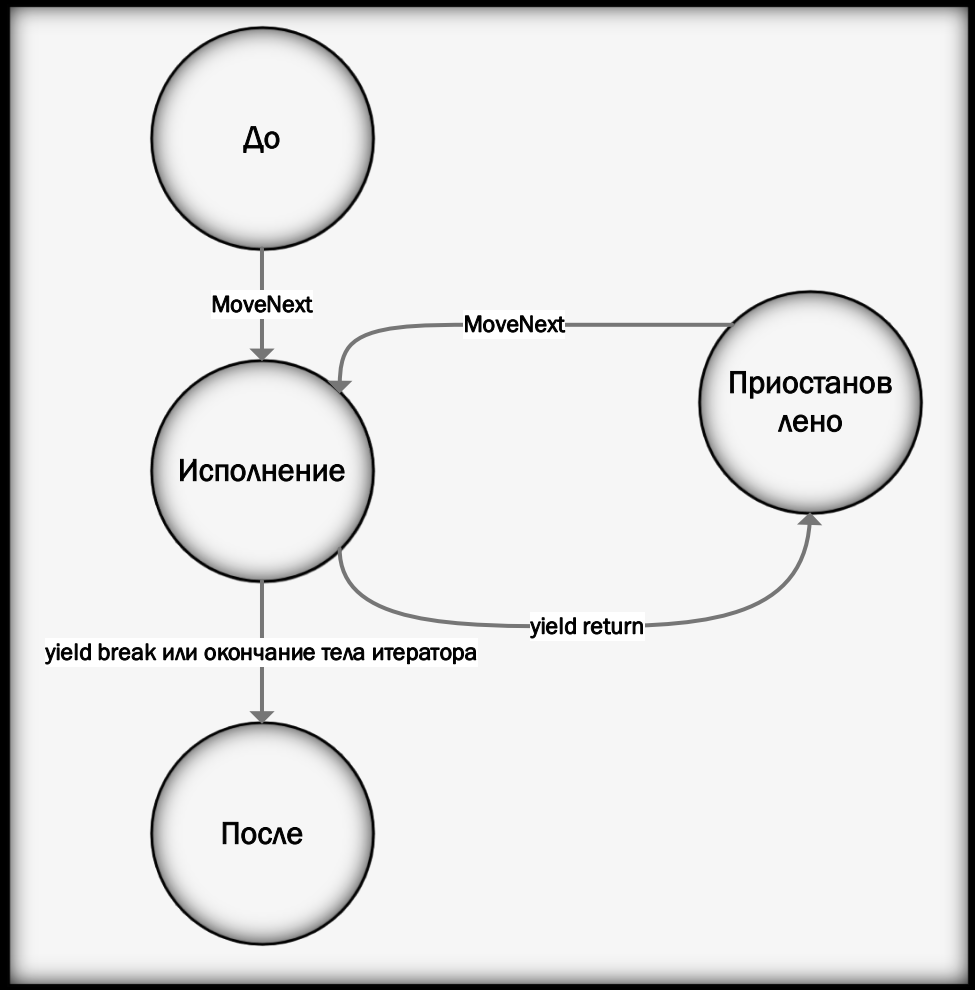
```
DemoCollection dc = new DemoCollection();
foreach(var item in dc)
{
    Console.WriteLine($"{item} ");
}
```

КОНЕЧНЫЙ АВТОМАТ ВНУТРИ MOVENEXT()



[Рисунок из статьи Programming stuff: Итераторы в языке C#. Часть 2 (sergeyteplyakov.blogspot.com)]

ИТЕРАТОР, КАК КОНЕЧНЫЙ АВТОМАТ



ПРИМЕР. КЛАСС С БЛОКОМ ИТЕРАТОРА

```
class Numbers {
    public uint Numb { get; set; }
    public Numbers(uint n) => Numb = n;
    public IEnumerator<uint> GetEnumerator() => IterMethod();

    public IEnumerator<uint> IterMethod() {
        uint newX = Numb;
        do  {
            uint d = newX % 10;
            yield return d;
            newX = newX / 10;
        }
        while (newX != 0);
    }
}
```

```
Numbers n = new Numbers(7096);
foreach(var i in n)
{
    Console.WriteLine(i);
}
```

```
Вывод:
6
9
0
7
```

ПРИМЕР. КЛАСС КОЛЛЕКЦИЯ С ИТЕРАТОРОМ

```
class Digits  {
    public uint Numb { get; set; }
    public Digits(uint n) => Numb = n;
    public IEnumerator<uint> GetEnumerator() => IterMethod().GetEnumerator();

    public IEnumerable<uint> IterMethod()  {
        List<uint> list = new List<uint>();
        uint newX = Numb;
        do  {
            uint d = newX % 10;
            list.Add(d);
            newX = newX / 10;
        }
        while (newX != 0);
        for (int i = list.Count - 1; i != -1; i--)
            yield return list[i];
    }
}
```

```
Digits d = new Digits(100500);
foreach(var i in d)
{
    Console.WriteLine(i);
}
```

Вывод:

```
1
0
0
5
0
0
```

БЛОК С ИТЕРАТОРОМ ВОЗВРАЩАЕТ IENUMERATOR<T> ИЛИ IENUMERABLE<T>

```
// Итератор / перечислитель (enumerator):
public IEnumerator<string> IteratorMethod()
{  …..
   yield return … ;
}
```

```
// Итерируемая коллекция / перечисляемое (enumerable):
public IEnumerable<string> IteratorMethod()
{  …..
   yield return … ;
}
```

ИТЕРАТОРЫ И YIELD RETURN (1)

возврат обобщенного перечислителя

```
public IEnumerator<string> BlackAndWhite() // версия 1
{
    yield return "black";   // yield return
    yield return "gray";    // yield return
    yield return "white";   // yield return
}
```

ИТЕРАТОРЫ И YIELD RETURN (2)

```
  возврат обобщенного перечислителя
        ↓
public IEnumerator<string> BlackAndWhite() // версия 2
{
  string[] TheColors = { "black", "gray", "white" };
  for (int i = 0; i < TheColors.Length; i++)
        yield return TheColors[i]; // yield return
}
```

ИСПОЛЬЗОВАНИЕ ИТЕРАТОРА ДЛЯ СОЗДАНИЯ ПЕРЕЧИСЛИТЕЛЯ

```
class ColorsCollection
{
    // Возвращает итератор.
    public IEnumerator<string> GetEnumerator() => BlackAndWhite();

    public IEnumerator<string> BlackAndWhite()
    {
        yield return "black";
        yield return "gray";
        yield return "white";
    }
}
```

```
ColorsCollection mc = new ColorsCollection();
foreach (string shade in mc) // используем объект класса
Console.WriteLine(shade);
```

Вывод:
black
gray
white

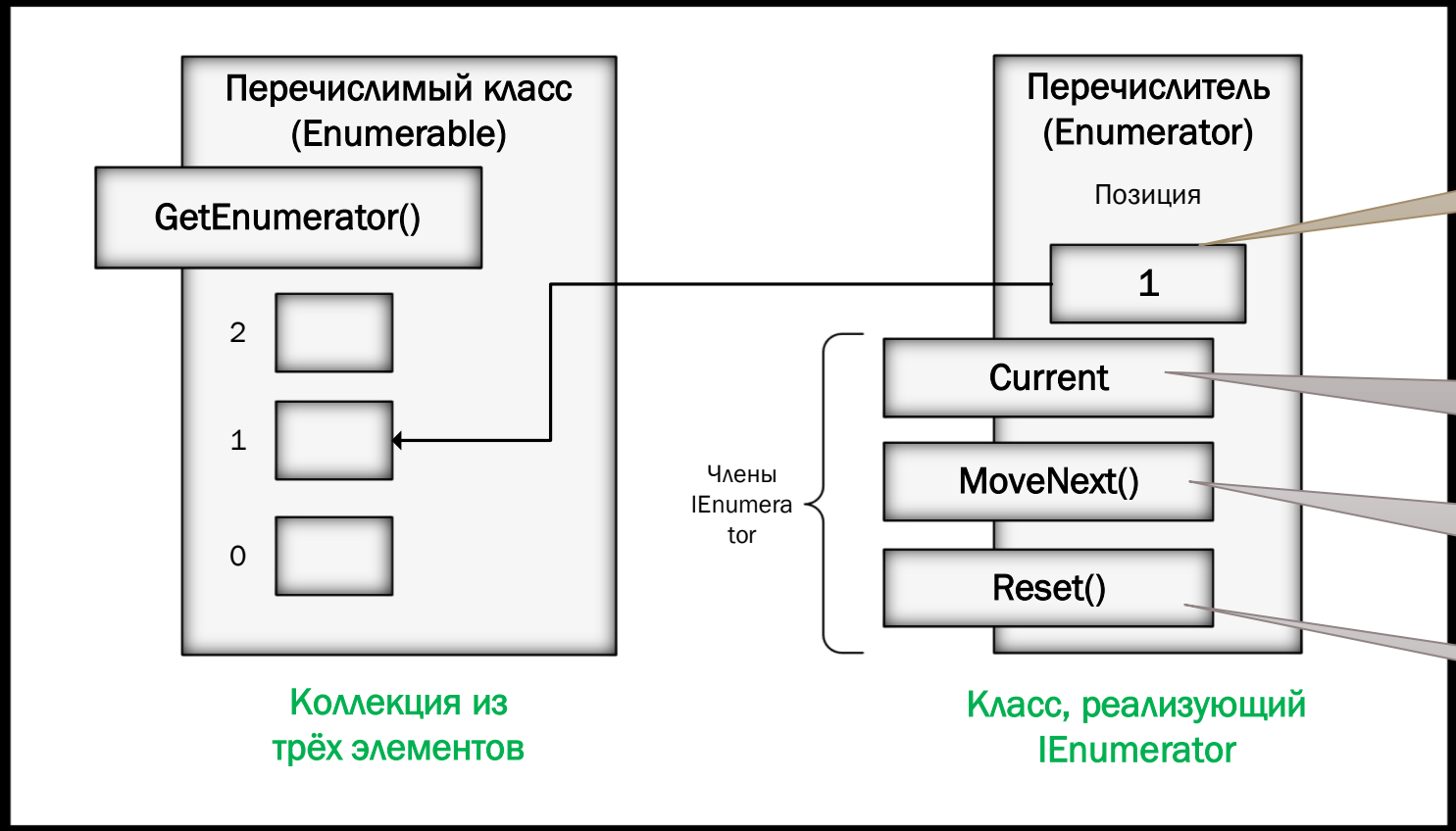
ИСПОЛЬЗОВАНИЕ ИТЕРАТОРА ДЛЯ СОЗДАНИЯ ИТЕРИРУЕМОЙ КОЛЛЕКЦИИ

```
class ColorsCollection  {
    public IEnumerator<string> GetEnumerator() {
        IEnumerable<string> myEnumerable = BlackAndWhite();
        return myEnumerable.GetEnumerator();
    }
    public IEnumerable<string> BlackAndWhite() {
        yield return "black";
        yield return "gray";
        yield return "white";
    }
}
```

```
ColorsCollection mc = new ColorsCollection();
foreach (string shade in mc) // используем объект класса
Console.Write("{0} ", shade);
// используем метод-итератор класса:
foreach (string shade in mc.BlackAndWhite())
    Console.Write("{0} ", shade);
```

Вывод:
black gray white black gray white

ИТЕРАТОР ДЛЯ ИТЕРИРУЕМОЙ КОЛЛЕКЦИИ



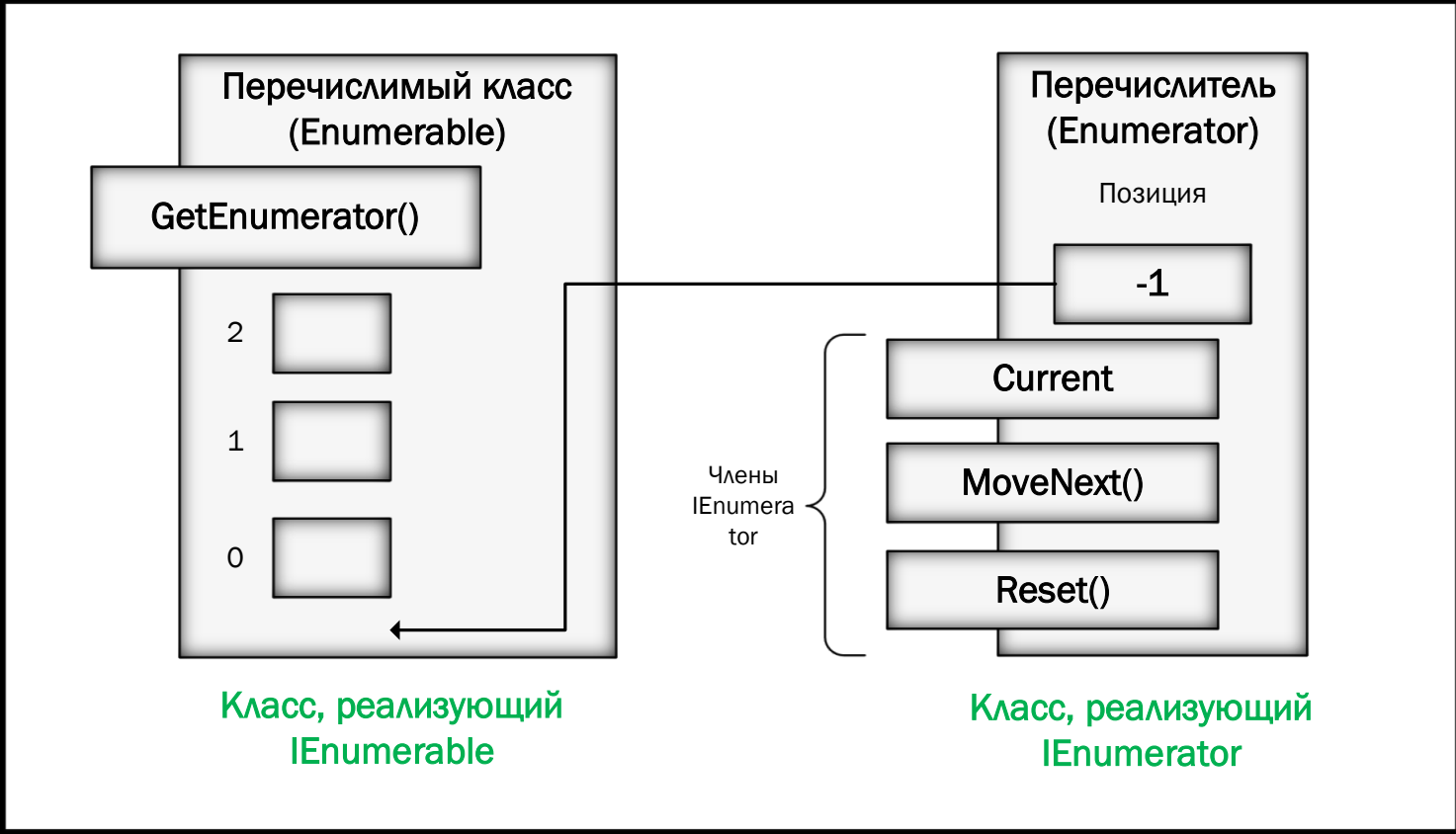
Позиция текущего элемента

Свойство `Current` возвращает значение элемента с текущей позиции

Передвигает позицию на следующий элемент последовательности

Сбрасывает позицию на исходную конфигурацию

ИНТЕРФЕЙС IENUMERABLE



СПОСОБ ОПРЕДЕЛЕНИЯ ИТЕРИРУЕМОГО КЛАССА

Класс, реализующий IEnumerable, коллекция

```
using System.Collections;
Реализация интерфейса IEnumerable
                  ↓
class MyClass : IEnumerable
{
      public IEnumerator GetEnumerator { ... }
...                   ↑
}       возврат объекта типа IEnumerator
```

```
using System.Collections;

class MyEnumerator: IEnumerator
{     возвращает ссылку на объект
          ↓
  public object Current { get; }     // текущий элемент
  public bool MoveNext() { ... }     // перемещение к следующему элементу
  public void Reset() { ... }              // сброс (возврат к началу)
  ...
}
```

Класс, реализующий IEnumerator, итератор

ПРИМЕР

Enumerator / Итератор

```
class ColorEnumerator : IEnumerator
{
    string[] _colNames;
    int _pos = -1;
    public ColorEnumerator(string[] otherNames)
    { // Конструктор.
        _colNames = new string[otherNames.Length];
        for (int i = 0; i < otherNames.Length; i++)
            _colNames[i] = otherNames[i];
    }
    public object Current { get => _colNames[_pos]; }   // Current.
    public bool MoveNext()
    {   // MoveNext().
        if (_pos < _colNames.Length - 1) { _pos++; return true; }
        else return false;
    }
    public void Reset() => _pos = -1;  // Reset ().
} // End of class ColorEnumerator.
```

```
class MyColors : IEnumerable
{
    string[] _colors = { "Red", "Yellow", "Blue" };
    // GetEnumerator()
    public IEnumerator GetEnumerator() => new ColorEnumerator(_colors);
}  // End of class MyColors.
```

```
MyColors mc = new MyColors();
foreach (string color in mc)
    Console.WriteLine(color);
```

Вывод:
Red
Yellow
Blue

Enumerable / Итерируемая коллекция

https://replit.com/@olgamaksimenkova/ColorsCollection

ИТЕРИРУЕМАЯ КОЛЛЕКЦИЯ БЕЗ ИНТЕРФЕЙСА

```
class ColorEnumerator : IEnumerator
{
    string[] _colNames;
    int _pos = -1;
    public ColorEnumerator(string[] otherNames)
    { // Конструктор.
        _colNames = new string[otherNames.Length];
        for (int i = 0; i < otherNames.Length; i++)
            _colNames[i] = otherNames[i];
    }
    public object Current { get => _colNames[_pos]; }
    public bool MoveNext()
    {   // MoveNext().
        if (_pos < _colNames.Length - 1) { _pos++; return true; }
        else return false;
    }
    public void Reset() => _pos = -1;  // Reset ().
} // End of class ColorEnumerator.
```

```
class ColorEnumerator
{
    string[] _colNames;
    int _pos = -1;
    public ColorEnumerator(string[] otherNames)
    { // Конструктор.
        _colNames = new string[otherNames.Length];
        for (int i = 0; i < otherNames.Length; i++)
            _colNames[i] = otherNames[i];
    }
    public string Current { get => _colNames[_pos]; }
    public bool MoveNext()
    {   // MoveNext().
        if (_pos < _colNames.Length - 1) { _pos++; return true; }
        else return false;
    }
} // End of class ColorEnumerator.
```

```
class MyColors : IEnumerable
{
    string[] _colors = { "Red", "Yellow", "Blue" };
    // GetEnumerator()
    public IEnumerator GetEnumerator() => new ColorEnumerator(_colors);
}  // End of class MyColors.
```

```
class MyColors
{
    string[] _colors = { "Red", "Yellow", "Blue" };
    // GetEnumerator()
    public ColorEnumerator GetEnumerator() => new ColorEnumerator(_colors);
}  // End of class MyColors.
```

ОБЩИЕ ПАТТЕРНЫ

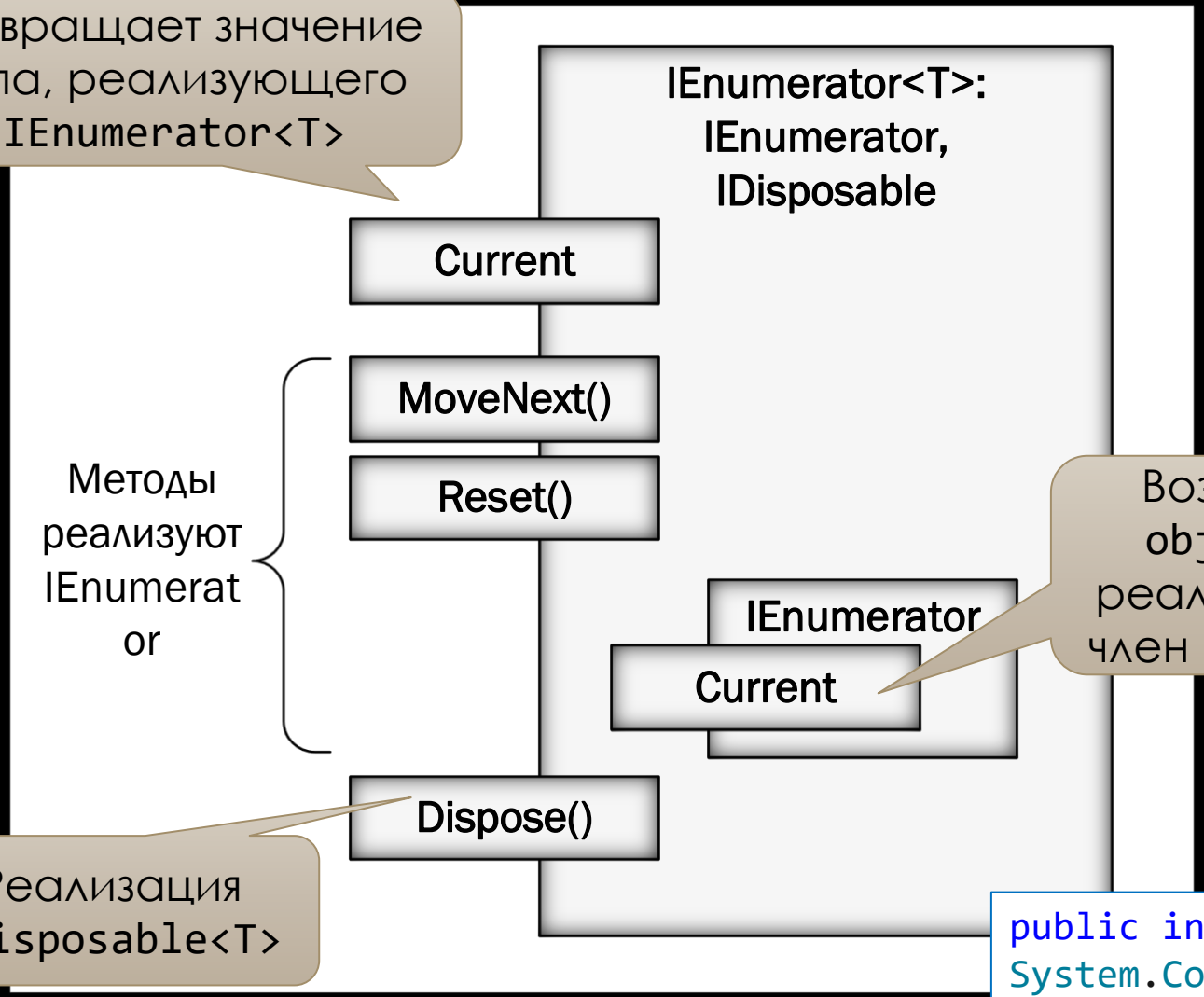
Коллекция с итератором

```
// шаблон проектирования итератора
class MyClass      { // утиная типизация
  public IEnumerator<string>  GetEnumerator()
  {
        return IteratorMethod();
  }
  public IEnumerator<string> IteratorMethod()
  {
        yield return … ;
  }
}
void Main()        {
  MyClass mc = new MyClass();
  foreach (string s in mc)
  …..
  foreach (string s in mc.IteratorMethod())
  …..

}
```

Итерируемая коллекция с итератором

```
// шаблон проектирования итератора
class MyClass      { // утиная типизация
  public IEnumerator<string>  GetEnumerator()
  {
        return IteratorMethod().GetEnumerator();
  }
  public IEnumerable<string> IteratorMethod()
  {
        yield return … ;
  }
}
void Main()        {
  MyClass mc = new MyClass();
  foreach (string s in mc)
  …..
  foreach (string s in mc.IteratorMethod())
  …..
}
```

РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА IENUMERATOR<T>



- Обязательно для обобщенной версии (часть `IEnumerator<T>`…)
- Опционально для необобщенной версии (не часть `IEnumerator`…)

```
public interface IEnumerator
```

```
public interface IEnumerable
```

```
public interface IEnumerable<out T> :
System.Collections.IEnumerable
```

```
public interface IEnumerator<out T> : IDisposable,
System.Collections.IEnumerator
```

КЛАСС, РЕАЛИЗУЮЩИЙ ИНТЕРФЕЙС IENUMERATOR<T>

```
using System.Collections;
using System.Collections.Generic;

class MyGenEnumerator: IEnumerator< T > {
  public T Current { get; }      // IEnumerator<T>--Current
        явная реализация
            ↓
  object IEnumerator.Current { get { ... } } // IEnumerator--Current
  public bool MoveNext() { ... }              // IEnumerator--MoveNext
  public void Reset() { ... }          // IEnumerator—Reset

  public void Dispose() { ... }       // IDisposable--Dispose
...
}
```

ПРИМЕР. ВОЗВРАТ ИТЕРАТОРА ИЗ МЕТОДА

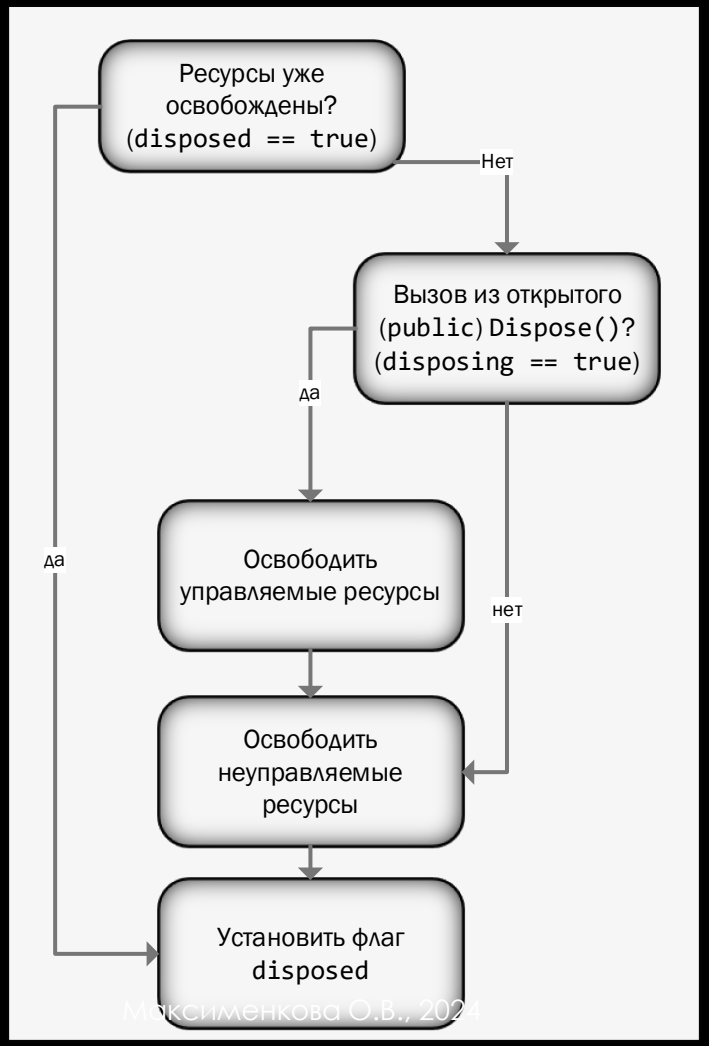
```
using System.IO;

string sourceDir = @"..\..\..";

var dirs = Directory.EnumerateDirectories(sourceDir);
var files = Directory.EnumerateFiles(sourceDir);

foreach(var dir in dirs)
{
    Console.WriteLine(dir.ToUpper());
}
foreach (var file in files)
{
    Console.WriteLine(file);
}
```

Итерируемая коллекция папок

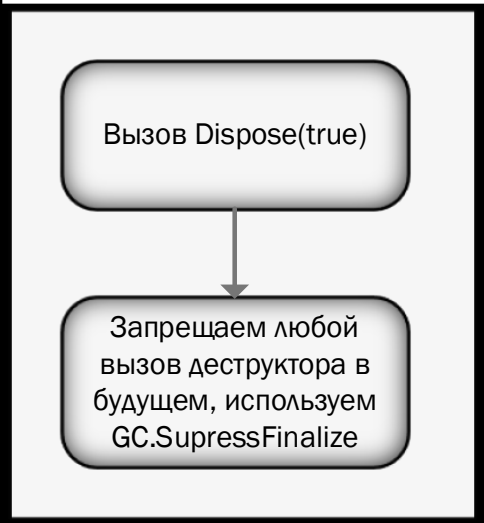
Итерируемая коллекция файлов

СТАНДАРТНЫЙ ПАТТЕРН DISPOSE

`protected virtual void Dispose(bool disposing)`



`public void Dispose()`



`Destructor`

МЕТОД DISPOSE( )

```
class MyClass          {
    bool disposed = false;              // флаг статуса освобождения
////////////////////////////////////////////////////
    public void Dispose()    {
        if (!disposed)       {      // проверка флага
          // вызвать Dispose для управляемых ресурсов….
          // освободить неуправляемые ресурсы….
        }
        disposed = true;             // установка флага статуса
        GC.SuppressFinalize(this);   // чтобы GC не вызывал Finalize
    }
////////////////////////////////////////////////////
    ~MyClass() // деструктор
    {
        if (!disposed)       {       // проверка флага
         // освободить неуправляемые ресурсы
          ...
    }
}
```

СТАНДАРТНЫЙ ПАТТЕРН DISPOSE

```
class MyClass: IDisposable  {
    bool disposed = false; // флаг статуса освобождения
    public void Dispose()  { // открытый метод Dispose
        Dispose( true );
        GC.SuppressFinalize(this);
    }
    ~MyClass()  {     // деструктор
        Dispose(false);
    }

  protected virtual void Dispose(bool disposing)  {        // защищенный!
      if (!disposed)  {
          if (disposing)  {
           // Освободить управляемые ресурсы
          }
       // Освободить неуправляемые ресурсы
      }
      disposed = true;
  }
}
```

МНОЖЕСТВЕННЫЕ ИТЕРАТОРЫ

```
using System;
using System.Collections.Generic;
```

```
class ColorCollection  {
    string[] Colors ={"Red", "Orange", "Yellow",
                "Green", "Blue", "Purple"};
    public IEnumerable<string> Forward() { // перечислимое 1
        for (int i = 0; i < Colors.Length; i++)
            yield return Colors[i];
    }
    public IEnumerable<string> Reverse() {// перечислимое 2

        for (int i = Colors.Length - 1; i >= 0; i--)
            yield return Colors[i];
    }
}
```

```
static void Main() {
    ColorCollection cc = new ColorCollection();
    foreach (string color in cc.Forward())
        Console.Write("{0} ", color);
    Console.WriteLine();
    foreach (string color in cc.Reverse())
        Console.Write("{0} ", color);

    Console.WriteLine();

    IEnumerable<string> ieable = cc.Reverse();
    IEnumerator<string> ieator = ieable.GetEnumerator();
    while (ieator.MoveNext())
        Console.Write("{0} ", ieator.Current);
    Console.WriteLine();
}
```

МНОЖЕСТВЕННЫЕ ИТЕРАТОРЫ (1)

```
class ColorsCollection : IEnumerable<string>  {
    bool ColorFlag = true;
    public ColorsCollection(bool flag)  =>  ColorFlag = flag;   // конструктор
    IEnumerator<string> BlackAndWhite   {// свойство

        get   {
            yield return "black";
            yield return "gray";
            yield return "white";
        }
    }
    IEnumerator<string> Colors    {  // свойство

        get   {
            string[] theColors = { "blue", "red", "yellow" };
            for (int i = 0; i < theColors.Length; i++)
                yield return theColors[i];
        }
    }
```

МНОЖЕСТВЕННЫЕ ИТЕРАТОРЫ (2)

```
public IEnumerator<string> GetEnumerator() {
        return ColorFlag ?
                          Colors // возврат перечислителя Colors
                        : BlackAndWhite;  // возврат перечислителя BlackAndWhite
    }

    System.Collections.IEnumerator
            System.Collections.IEnumerable.GetEnumerator() {
        return ColorFlag ?
                          Colors    // возврат перечислителя Colors
                        : BlackAndWhite; // возврат перечислителя BlackAndWhite
    }
}
```

МНОЖЕСТВЕННЫЕ ИТЕРАТОРЫ (3)

```
class Program
{
    static void Main()
    {
        ColorsCollection mc1 = new ColorsCollection(true); // вызов конструктора
        foreach (string s in mc1)
            Console.Write("{0} ", s);
        Console.WriteLine();
        ColorsCollection mc2 = new ColorsCollection(false); // вызов конструктора
        foreach (string s in mc2)
            Console.Write("{0} ", s);
        Console.WriteLine();
    }
}
```

```
Результат работы программы:
blue red yellow
black gray white
```

https://replit.com/@olgamaksimenkova/MultiNumerableCollection

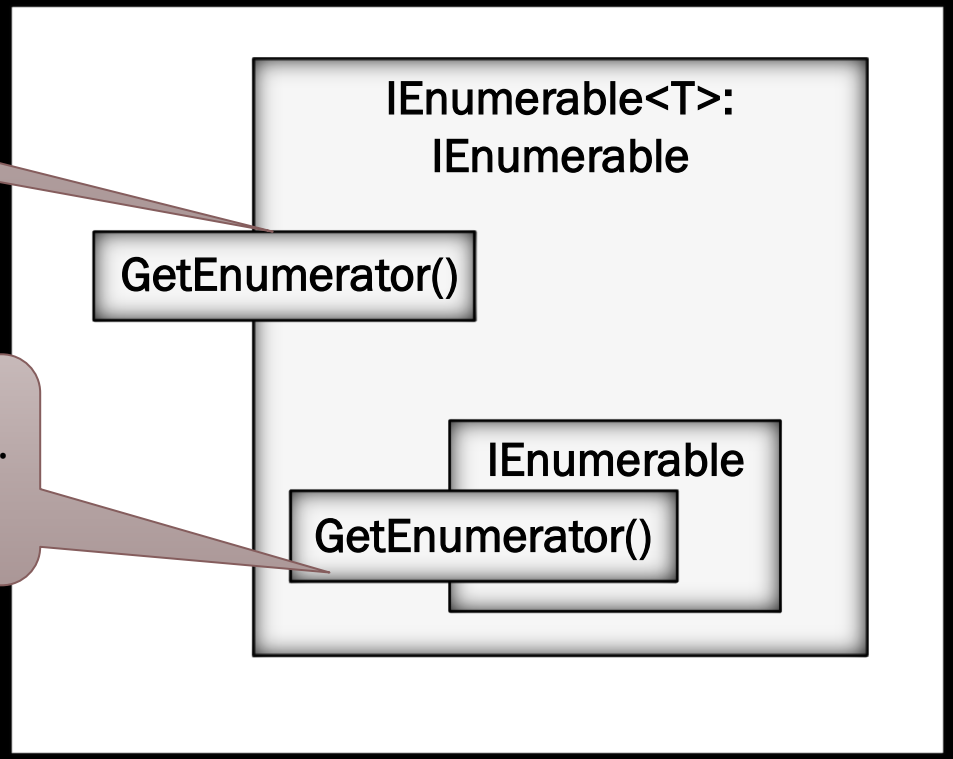
ВЛОЖЕННЫЕ ИТЕРАТОРЫ

```
public IEnumerable<uint> IterMethodEvenIndexOnly (IEnumerable<uint> seq) {
            int i = 0;
            try {
                foreach (var item in seq) {
                    if (i++ % 2 == 0)
                        yield return item;
                    // yield break; // завершает перечисление
                }
            }
            // catch (Exception ex) { }          // нельзя использовать try-catch!
            finally {                       // использовать try-finally можно!
                // в finally нельзя использовать yield!
            }
}
```

РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА IENUMERABLE<T>

Реализует `IEnumerable<T>` и возвращает тип `IEnumerator<T>`

Реализует `IEnumerable` и возвращает тип `IEnumerator`. Реализация явная, как член `IEnumerable`



КЛАСС, РЕАЛИЗУЮЩИЙ ИНТЕРФЕЙС IENUMERABLE<T>

```
using System.Collections;
using System.Collections.Generic;

class MyGenEnumerable: IEnumerable<T>
{
     public IEnumerator<T> GetEnumerator() { ... } // IEnumerable<T>
        явная реализация
                      ↓
     IEnumerator IEnumerable.GetEnumerator() { ... } // IEnumerable
...
}
```

```
public interface IEnumerable<out T> : System.Collections.IEnumerable
```

МОДЕЛИРОВАНИЕ ОПЕРАТОРА FOREACH

using System;
using System.Collections;

```
public static void Main() {
    int[] arr1 = { 10, 11, 12, 13 };

    // получаем итератор System.Collections;
    IEnumerator ie = arr1.GetEnumerator();
    // IEnumerator<int> ie = (arr1 as IEnumerable<int>).GetEnumerator();
    // ...Generic;

    while (ie.MoveNext()) { // к следующему элементу
        int item = (int)ie.Current; // получаем текущий элемент
        Console.WriteLine($"Item value: { item }");
    }
}
```

Результат работы:

10
11
12
13

ССЫЛКИ

- Утипизация в C# (https://habr.com/ru/post/41377/)
- Duck typing или «так ли прост старина foreach (http://sergeyteplyakov.blogspot.com/2012/08/duck-typing-foreach.html)
- Паттер Итератор (Programming stuff: Паттерн Итератор (sergeyteplyakov.blogspot.com))
- IEnumerator <T> интерфейс (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.ienumerator-1?view=net-7.0)
- IEnumerable <T> интерфейс (https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.ienumerable-1?view=net-7.0)
- Итераторы (C#) (https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/iterators)
- IEnumerable интерфейс в C# и LSP (https://habr.com/ru/post/257667/)