ИНТЕРФЕЙСЫ. IENUMERABLE

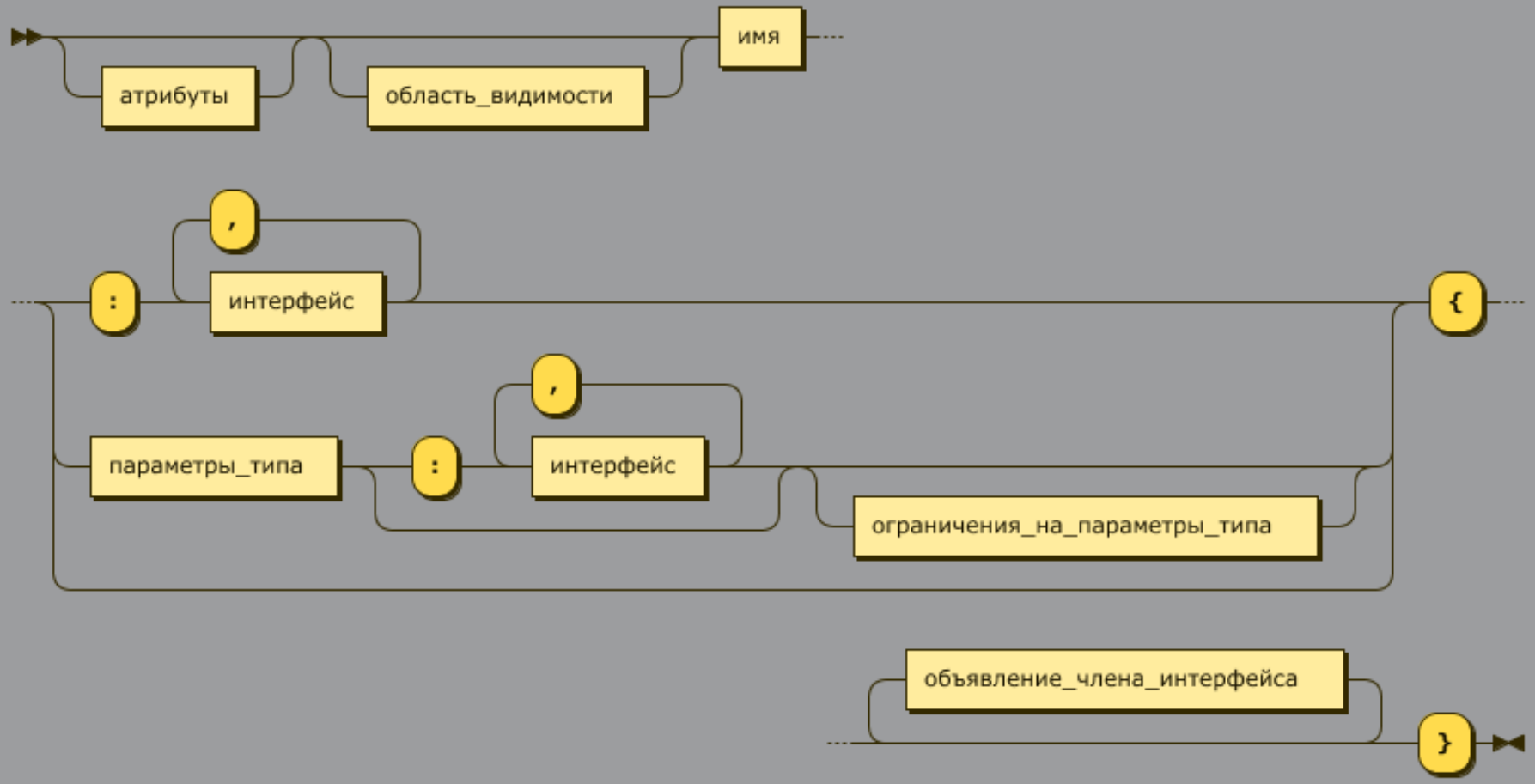
C# - ЛЕКЦИЯ 9

ИНТЕРФЕЙСЫ

- Задают контракт для типов, их реализующих
- Возможна реализация множества интерфейсов одновременно
- Приводимость работает как с наследованием
- Это ссылочные типы
- Экземплры - это экземпляры классов или струтур

```
объявление_интерфейса
    ::= атрибуты? область_видимости? имя
        ((':' интерфейс (',' интерфейс)*)? | параметры_типа (':' интерфейс
        '{' объявление_члена_интерфейса*'}'
```

объявление_интерфейса:



```
interface IShape
{
    double GetArea();

    double GetPerimeter();
}
```

объявление_члена_интерфейса

- Нельзя указать объявлять конструкторы, деструктор, поля
- Нельзя указать область видимости (всагда public)
- Нельзя указать реализацию (можно в C# 8)
- Статические члены - обязаны иметь реализацию

```
interface ISomeInterface
{
    double SomeProperty { get; }

    void DoWork();

    string this[int index]
    {
        get;
        set;
    }
}
```

Явная реализация интерфейса

- Что делать если тип реализует интерфейсы с пересекающимися по синтаксису, но отличающиеся по смысу члены?
- В этом случае используют явную реализацию интерфейса
- Синтаксис: перед именем члена указывают имя интерфейса с точкой, а область видимости не указывают

```
public class ScanerAndRecognizer : IScaner, IRecognizer
{
    void IScaner.Run()
    {
        System.Console.WriteLine("IScaner.Run");
    }

    void IRecognizer.Run()
    {
        System.Console.WriteLine("IRecognizer.Run");
    }
}

var obj = new ScanerAndRecognizer();
//obj.Run();  // Ошибка компиляции

IScaner c = obj;
c.Run();  // Вызов IScaner.Run

IRecognizer s = obj;
s.Run(); // Вызов IRecognizer.Run
```

ИЗВЕСТНЫЕ ИНТЕРФЕЙСЫ

```
public interface ICloneable
{
    object Clone();
}
```

```
public interface IEquatable<T>
{
    bool Equals(T? other);
}
```

```
public interface IComparable
{
    int CompareTo(object? obj);
}
```

```
public interface IComparable<in T>
{
    int CompareTo(T? obj);
}
```

```
interface IConvertible
{
    TypeCode GetTypeCode();
    bool ToBoolean(IFormatProvider? provider);
    byte ToByte(IFormatProvider? provider);
    char ToChar(IFormatProvider? provider);
    DateTime ToDateTime(IFormatProvider? provider);
    decimal ToDecimal(IFormatProvider? provider);
    double ToDouble(IFormatProvider? provider);
    short ToInt16(IFormatProvider? provider);
    int ToInt32(IFormatProvider? provider);
    long ToInt64(IFormatProvider? provider);
    sbyte ToSByte(IFormatProvider? provider);
    float ToSingle(IFormatProvider? provider);
    string ToString(IFormatProvider? provider);
    object ToType(Type conversionType, IFormatProvider? provider);
    ushort ToUInt16(IFormatProvider? provider);
    uint ToUInt32(IFormatProvider? provider);
    ulong ToUInt64(IFormatProvider? provider);
}
```

```
public interface IFormatProvider
{
    object? GetFormat(Type? formatType);
}

// IFormatProvider реализован классом CultureInfo
CultureInfo culture;

culture = CultureInfo.CurrentCulture;
culture = CultureInfo.CurrentUICulture;
culture = CultureInfo.InvariantCulture;
culture = CultureInfo.CreateSpecificCulture("de-CA"); //German (Canada)
culture = new CultureInfo("fr-FR");
```

```
string dateString = "11.03.2021";
var culture = new CultureInfo("en-US");
DateTime dt = DateTime.Parse(dateString, culture);
Console.WriteLine("{0:dd MMMM yyyy}", dt);  //  03 November 2021

culture = new CultureInfo("ru-RU");
DateTime dt = DateTime.Parse(dateString, culture);
Console.WriteLine("{0:dd MMMM yyyy}", dt);  //  11 March 2021
```

```
public interface IFormattable
{
    string ToString(string? format, IFormatProvider? formatProvider);
}
```

```
interface IDisposable
{
    void Dispose();
}
```

КОНСТРУКЦИЯ USING

USING

Используется для повышения удобства использования объектов, реализующих интерфейс `System.IDisposable`

IDisposable

```
Stream stream = File.OpenRead("todos.csv");
try
{
    //  работа с потоком
}
finally
{
    if (stream != null)
        stream.Dispose();
}
```

СИНТАКСИС `using`

```
using (<тип_переменной> <имя_переменной> = <выражение>)
{
}
```

где выражение должно возвращать тип, приводимый к IDisposable

ПРИМЕР

```
using (Stream stream = File.OpenRead("todos.csv"))
{
    //  работа с потоком
}
```

НЕСКОЛЬКО USING ПОДРЯД

```
using (Stream stream = File.OpenRead("todos.csv"))
using (StreamReader reader = new StreamReader(stream))
{
    //  работа с потоком и читателем
}
```

IEnumerable

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

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

```
public interface IEnumerator
{
    object? Current { get; }

    bool MoveNext();

    void Reset();
}

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

```
var array = new[] { 2, 4, 7 };
var enumerator = array.GetEnumerator();

while (enumerator.MoveNext())
{
    Console.WriteLine(enumerator.Current);
}
```

Помним про IDisposable

```
var array = new[] { 2, 4, 7 };
var enumerator = array.GetEnumerator();

try
{
    while (enumerator.MoveNext())
    {
        Console.WriteLine(enumerator.Current);
    }
}
finally
{
    (enumerator as IDisposable)?.Dispose();
}
```

Цикл foreach

```
var array = new[] { 2, 4, 7 };

foreach (var item in array)
{
    Console.WriteLine(item);
}
```

```
class Product
{
    public string Sku { get; set; }

    public string Title { get; set; }

    public decimal Price { get; set; }

    //  ...

    public override int GetHashCode()
    {
        return Sku.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        return (obj as Product)?.Sku == Sku;
    }
}
```

```
class Basket
{
    private readonly Dictionary<Product, int> items = new Dictionary<Product, int>();

    public void Add(Product product)
    {
        if (items.ContainsKey(product))
            items[product]++;
        else
            items[product] = 1;
    }

    public void Remove(Product product)
    {
        if (items.ContainsKey(product))
        {
            items[product]--;

            if (items[product] == 0)
               items.Remove(product);
        }
    }

    public int Count { get { return items.Count; } }

    //  ...
```

```
// ...

    public decimal Price
    {
        get
        {
            decimal result = 0;

            foreach (var item in items)
            {
                result += item.Key.Price * item.Value;
            }

            return result;
        }
    }

    public IEnumerator<BasketItem> GetEnumerator()
    {
        return new BasketEnumerator(items);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

```
class BasketEnumerator : IEnumerator<BasketItem>
{
    private Dictionary<Product, int> items;

    private IEnumerator<KeyValuePair<Product, int>> enumerator;

    public BasketEnumerator(Dictionary<Product, int> items)
    {
        this.items = items;
    }

    public bool MoveNext()
    {
        if (enumerator == null)
            enumerator = items.GetEnumerator();

        return enumerator.MoveNext();
    }

    //  ...
```

```
//  ...

    public BasketItem Current
    {
        get
        {
            return new BasketItem
            {
                Product = enumerator.Current.Key,
                Count = enumerator.Current.Value
            };
        }
    }

    object IEnumerator.Current { get { return Current; } }

    public void Reset()
    {
        enumerator?.Reset();
    }

    public void Dispose()
    {
        enumerator?.Dispose();
    }
}
```

```
class PositiveOddEnumerator : IEnumerator<int>
{
    public int Current { get; private set; }

    object IEnumerator.Current { get { return Current; } }

    public void Dispose()
    {
    }

    public bool MoveNext()
    {
        if (Current == 0)
            Current = 1;
        else
            Current += 2;

        return Current > 0;
    }

    public void Reset()
    {
        Current = 0;
    }
}
```

ICollection

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }

    void CopyTo(Array array, int index);
}
```

IList

```
public interface IList : ICollection, IEnumerable
{
    object this[int index] { get; set; }

    bool IsFixedSize { get; }

    bool IsReadOnly { get; }

    int Add(object value);

    void Clear();

    bool Contains(object value);

    int IndexOf(object value);

    void Insert(int index, object value);

    void Remove(object value);

    void RemoveAt(int index);
}
```

YIELD

100 - это много или мало?

```
class Sequences
{
    public IEnumerable<int> GetNaturalNumbers()
    {
        var result = new int[100];

        for (int i = 0; i < result.Length; i++)
            result[i] = i + 1;

        return result;
    }
}
```

Сделаем гибче

```
class Sequences
{
    public IEnumerable<int> GetNaturalNumbers(int count)
    {
        var result = new int[count];

        for (int i = 0; i < result.Length; i++)
            result[i] = i + 1;

        return result;
    }
}
```

Логика на уровень выше

```
class Sequences
{
    public IEnumerable<int> GetOddNaturalNumbers(int count)
    {
        var numbers = GetNaturalNumbers(count);
        var result = new List<int>();

        foreach (int number in numbers)
        {
            if (number % 2 == 1)
                numbers.Add(number);
        }

        return result;
    }
}
```

Логика на уровень выше

```
class Sequences
{
    public IEnumerable<int> GetOddNaturalNumbers(int count)   //  допустим
    {
        var numbers = GetNaturalNumbers(count); //  массив из 100 чисел
        var result = new List<int>();

        foreach (int number in numbers)
        {
            if (number % 2 == 1)
                numbers.Add(number);
        }

        return result;  //  в списке - массив из 64 чисел
    }
}
```

Попробуем с IEnumerable

```
class NaturalNumbers : IEnumerable<int>
{
    public IEnumerator<int> GetEnumerator()
    {
        return new NaturalNumbersEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

```
class NaturalNumbersEnumerator : IEnumerator<int>
{
    private int current = 0;

    public int Current
    {
        get
        {
            if (current == 0)
                throw new InvalidOperationException("Iteration not started yet");

            return current;
        }
        private set { current = value; }
    }

    object IEnumerator.Current { get { return Current; } }

    public bool MoveNext()
    {
        Current++;

        return true;
    }

    public void Reset()
    {
        Current = 0;
    }

    public void Dispose()
    {
    }
}
```

```
class OddNaturalNumbers : IEnumerable<int>
{
    public IEnumerator<int> GetEnumerator()
    {
        return new OddNaturalNumbersEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

```
class OddNaturalNumbersEnumerator : IEnumerator<int>
{
    private readonly NaturalNumbersEnumerator naturalNumbers = new NaturalNumbersEnumerator();

    public int Current
    {
        get { return naturalNumbers.Current; }
    }

    object IEnumerator.Current { get { return Current; } }

    public bool MoveNext()
    {
        do
            naturalNumbers.MoveNext();
        while (naturalNumbers.Current % 2 != 1);

        return true;
    }

    public void Reset()
    {
        naturalNumbers.Reset();
    }

    public void Dispose()
    {
    }
}
```

Yield

- Это ключевое слово языка
- Используется в комбинации с return и break в методах с возвращаемым значением типа IEnumerable
- Упрощает реализацию IEnumerable

Yield

```
IEnumerable<Something> GetSomethings()
{
    //  ...
    //  some code
    //  ...

    yield return someValue;

    //  ...
    //  some code
    //  ...

    yield return someValue;

    if (someCondition)
        yield break;

    //  ...
    //  some code
    //  ...

    yield return someValue;
}
```

```
var somethings = GetSomethings();
var enumerator = somethings.GetEnumerator();

enumerator.MoveNext();  // true
Something something = enumerator.Current;

enumerator.MoveNext();  // true
something = enumerator.Current;

enumerator.MoveNext(); //  false
```

```
class Sequences
{
    public IEnumerable<int> GetNaturalNumbers()
    {
        int i = 1;

        while (true)
        {
            yield return i;

            i++;
        }
    }

    public IEnumerable<int> GetOddNaturalNumbers()
    {
        foreach (var number in GetNaturalNumbers())
        {
            if (number % 2 == 1)
                yield return number;
        }
    }
}
```

ВОПРОСЫ