# СТРУКТУРЫ. ИНДЕКСАТОРЫ, ОПЕРАТОРЫ. ПЕРЕЧИСЛЕНИЯ.

C# - ЛЕКЦИЯ 7

# НА ПРОШЛОМ ЗАНЯТИИ

- Объявление структур
- Конструкторы
- Константы
- Поля
- Методы
- Свойства

# СЕГОДНЯ В ПРОГРАММЕ

- Индексаторы
- Операторы
- Перечисления

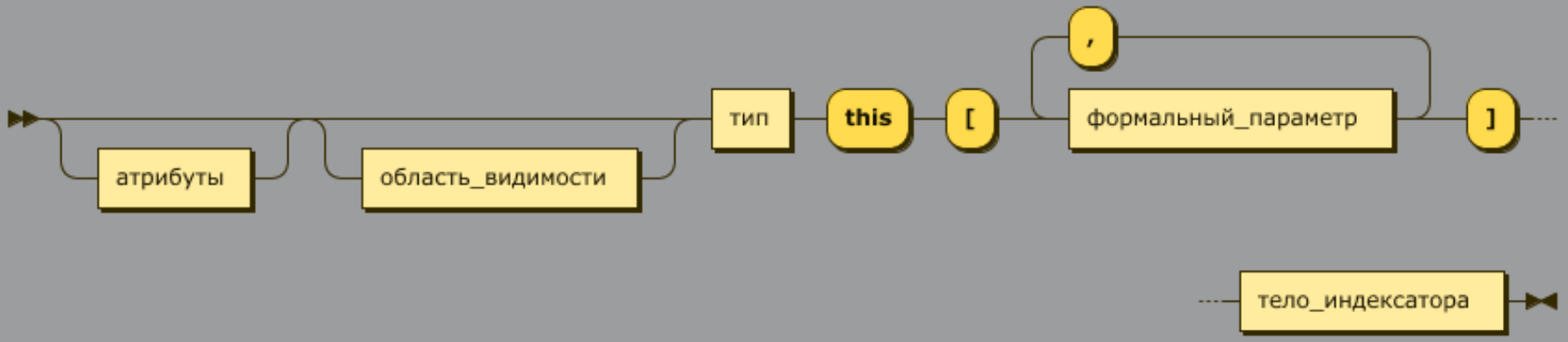
# ИНДЕКСАТОРЫ

# Индексаторы

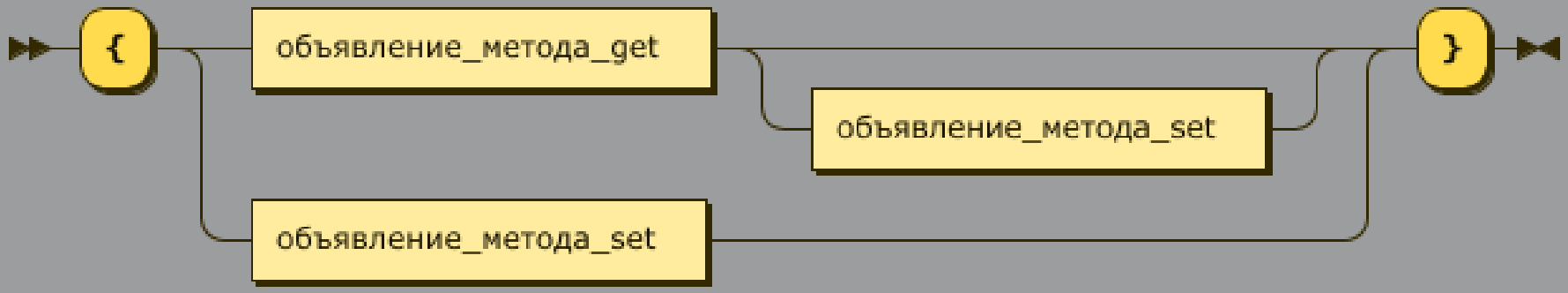
- Позволяют определить операцию [ ] для экземпляров типа

```
объявление_индексатора_структуры
    ::= атрибуты? область_видимости? тип 'this' '[' список_формальных_пара
        тело_индексатора

список_формальных_параметров
    ::= формальный_параметр | формальный_параметр ',' список_формальных_па

тело_индексатора
    ::= '{' (объявление_метода_get объявление_метода_set | объявление_мето
        объявление_метода_set) '}'

объявление_метода_get ::= атрибуты? область_видимости? 'get' тело_метода_g

объявление_метода_set ::= атрибуты? область_видимости? 'set' тело_метода_s
```

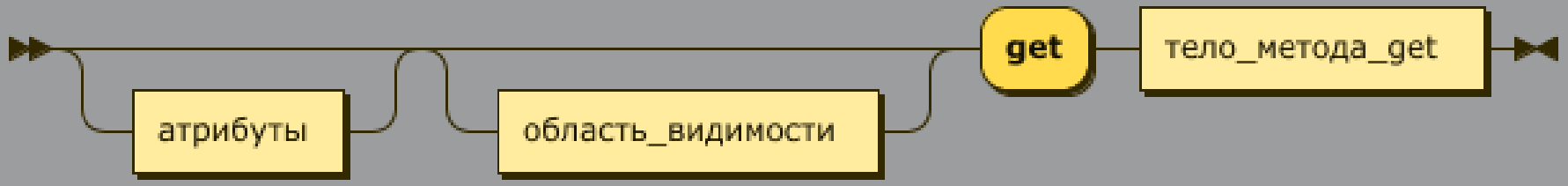
# объявление_индексатора_структуры:



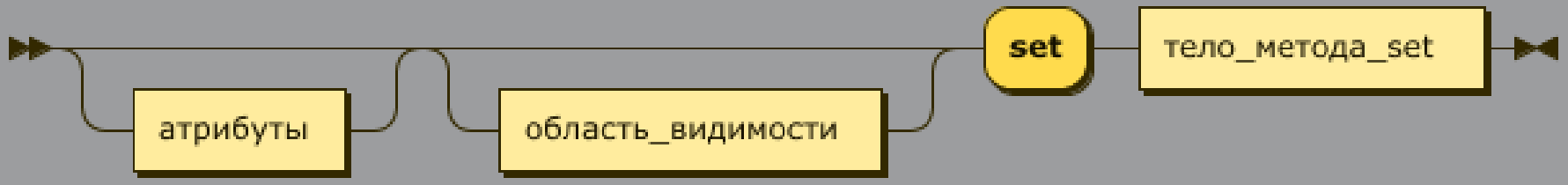
# тело_индексатора:



## объявление_метода_get:




## объявление_метода_set:



## тело_метода_get

- Соответствует телу метода, принимающего параметры индексатора и возвращающего значение типа данного индексатора
- Область видимости может только сужать область видимости индексатора

## тело_метода_set

- Соответствует телу метода, принимающего параметр с именем value и типом данного свойства, а также параметры индексатора и не возвращающего значение
- Область видимости может только сужать область видимости свойства

```
struct Point
{
    public int X { get; }

    public int Y { get; }

    public int this[int dim]
    {
        get
        {
            int result;

            if (dim == 0)
                result = X;
            else if (dim == 1)
                result = Y;
            else
                throw new ArgumentOutOfRangeException("Invalid dimension");

            return result;
        }
    }
}
```

```
class ListOfFloats
{
    private const int DefaultCapacity = 4;

    private float[] items = new float[DefaultCapacity];

    public int this[int index]
    {
        get { return items[index]; }
        set { items[index] = value; }
    }

    //  ...
}
```

# ОПЕРАТОРЫ

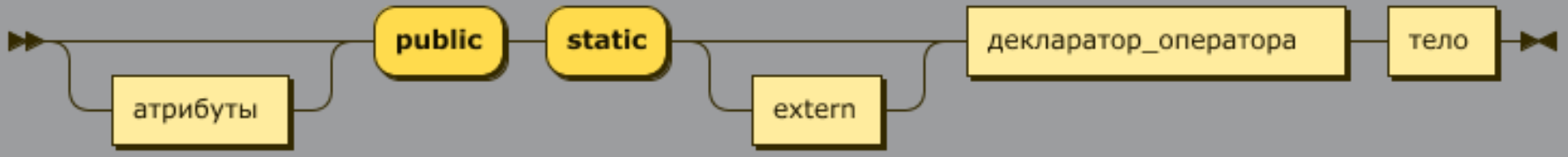
# ОПЕРАТОРЫ

- Унарные
- Бинарные
- Приведения типа

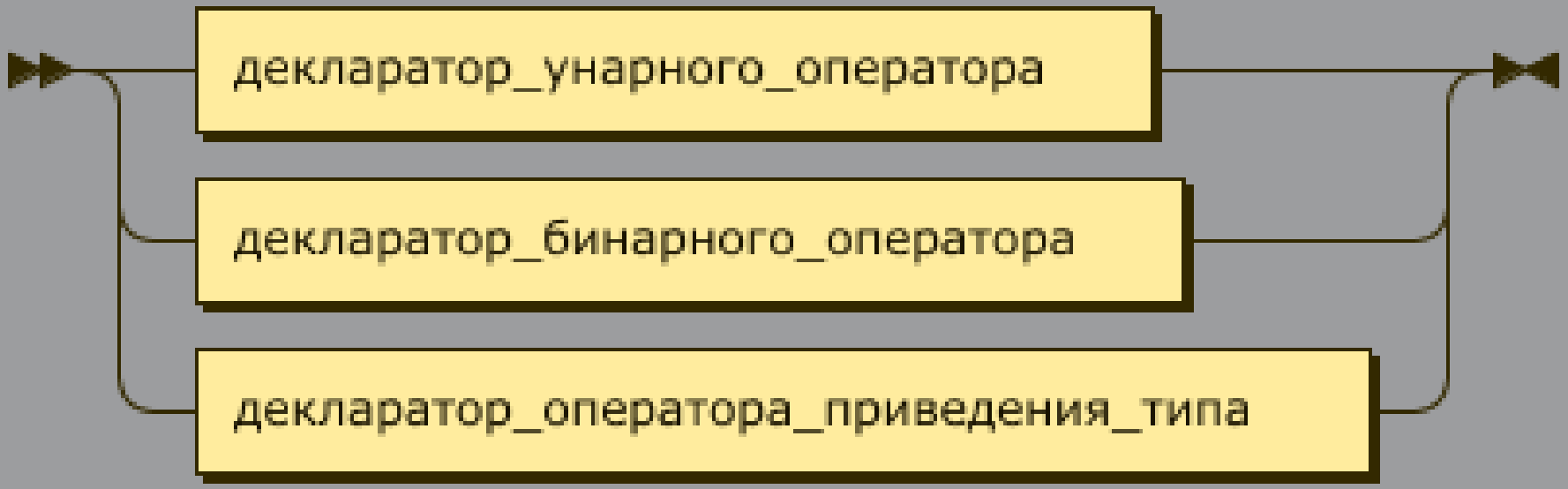
# СИНТАКСИС

```
объявление_оператора
    ::= атрибуты? 'public' 'static' extern? декларатор_оператора тело

декларатор_оператора
    ::= декларатор_унарного_оператора
      | декларатор_бинарного_оператора
      | декларатор_оператора_приведения_типа
```

## объявление_оператора:



декларатор_оператора:

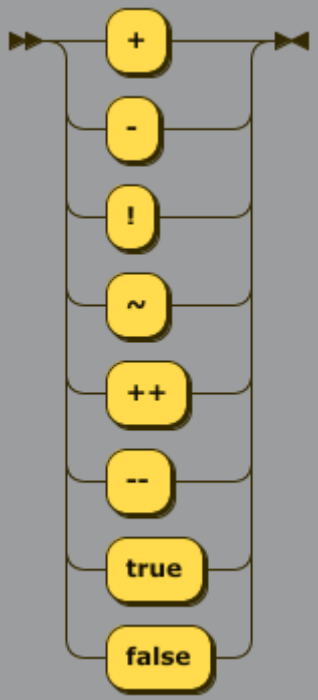


```
декларатор_унарного_оператора ::= тип 'operator' унарный_оператор '(' тип
```

```
унарный_оператор ::= '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false
```

декларатор_унарного_оператора:

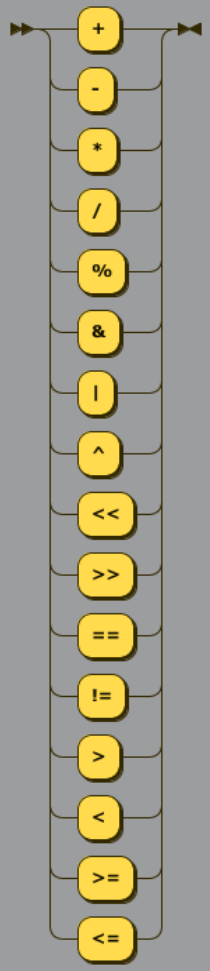
унарный_оператор:



```
декларатор_бинарного_оператора
    ::= тип 'operator' бинарный_оператор '(' тип имя ',' тип имя ')'

бинарный_оператор
    ::= '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<' | '>>'
        | '==' | '!=' | '>' | '<' | '>=' | '<='
```

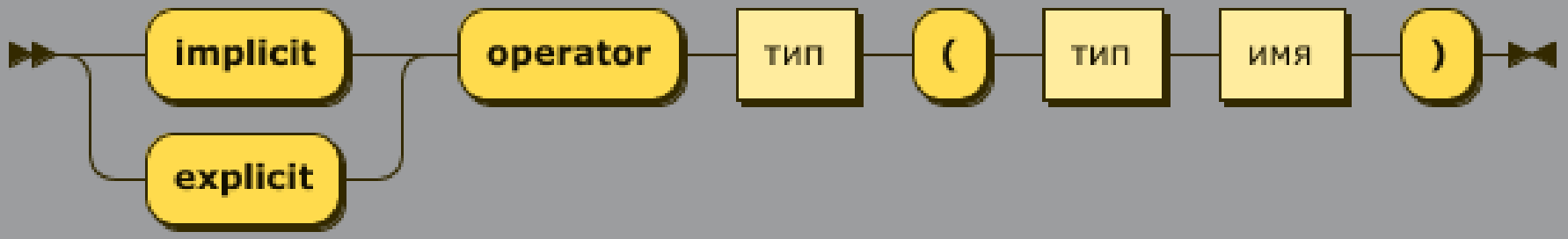
декларатор_бинарного_оператора:

# бинарный_оператор:



```
декларатор_оператора_приведения_типа
                        ::= 'implicit' 'operator' тип '(' тип имя ')' |
                            'explicit' 'operator' тип '(' тип имя ')'
```

декларатор_оператора_приведения_типа:



# ПРИМЕРЫ

## УНАРНЫЙ ОПЕРАТОР

```
struct MyDecimal
{
    public int Integer { get; }

    public int Fractional { get; }

    public MyDecimal(int integer, int fractional)
    {
        Integer = integer;
        Fractional = fractional;
    }

    public static MyDecimal operator -(MyDecimal number)
    {
        return new MyDecimal(-number.Integer, number.Fractional);
    }
}
```

# ПРИМЕРЫ

## БИНАРНЫЙ ОПЕРАТОР

```
struct Fraction
{
    int num, den;
    public Fraction(int num, int den)
    {
        this.num = num;
        this.den = den;
    }

    public static Fraction operator +(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.den + b.num * a.den,
           a.den * b.den);
    }

    public static Fraction operator *(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.num, a.den * b.den);
    }

    public static implicit operator double(Fraction f)
    {
        return (double)f.num / f.den;
    }
}
```

```
class Test
{
    static void Main()
    {
        Fraction a = new Fraction(1, 2);
        Fraction b = new Fraction(3, 7);
        Fraction c = new Fraction(2, 3);
        Console.WriteLine((double)(a * b + c));
    }
}
// Вывод:
// 0.880952380952381
```

# ПРИМЕРЫ

## IMPLICIT OPERATOR

```
struct Digit
{
    public Digit(double d) { val = d; }
    public double val;
    // ...other members

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        return d.val;
    }
    //  User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        return new Digit(d);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

# ПРИМЕРЫ

## EXPLICIT OPERATOR

```
struct Celsius
{
    public Celsius(float temp)
    {
        Degrees = temp;
    }

    public float Degrees { get; }

    public static explicit operator Fahrenheit(Celsius c)
    {
        return new Fahrenheit((9.0f / 5.0f) * c.Degrees + 32);
    }
}
```

```
struct Fahrenheit
{
    public Fahrenheit(float temp)
    {
        Degrees = temp;
    }

    public float Degrees { get; }

    public static explicit operator Celsius(Fahrenheit fahr)
    {
        return new Celsius((5.0f / 9.0f) * (fahr.Degrees - 32));
    }
}
```

```
class MainClass
{
    static void Main()
    {
        Fahrenheit fahr = new Fahrenheit(100.0f);
        Console.Write($"{fahr.Degrees} Fahrenheit");
        Celsius c = (Celsius)fahr;

        Console.Write($" = {c.Degrees} Celsius");
        Fahrenheit fahr2 = (Fahrenheit)c;
        Console.WriteLine($" = {fahr2.Degrees} Fahrenheit");
    }
}
// Вывод:
// 100 Fahrenheit = 37.77778 Celsius = 100 Fahrenheit
```

# ОГРАНИЧЕНИЯ НА УНАРНЫЕ ОПЕРАТОРЫ

Пусть унарный оператор объявлен для типа T. Тогда

- +, −, !, ~ принимают 1 параметр типа T или T? и возвр любой тип
- ++, −− принимают на вход 1 параметр типа T или T? и возвр тип T или T? или производный от него

## ОГРАНИЧЕНИЯ НА УНАРНЫЕ ОПЕРАТОРЫ

Пусть унарный оператор объявлен для типа T. Тогда

- `true` и `false` принимают 1 параметр типа T или T? и возвр bool
- `true` и `false` должны быть объявлены оба

# ОГРАНИЧЕНИЯ НА БИНАРНЫЕ ОПЕРАТОРЫ

Пусть бинарный оператор объявлен для типа T. Тогда

- Все кроме `<<` и `>>` принимают 2 параметра, хотя бы один из которых является `T` или `T?` и возвр любой тип
- `<<` и `>>` принимают 2 параметра, первый из которых имеет тип `T` или `T?`, а второй - `int` или `int?`
- `==` и `!=`, `>` и `<`, `>=` и `<=` должны быть объявлены попарно

# ОГРАНИЧЕНИЯ НА ОПЕРАТОРЫ ПРИВЕДЕНИЯ ТИПА

Путь исходный тип - S, а целевой - T. Если они nullable, то S0 и T0 - их истинные типы, иначе S0 и T0 - это S и T. Тогда

- S0 и T0 - разные типы
- оператор объявлен или в S0, или в T0
- ни S0, ни T0 не являются интерфейсом
- S не приводим к T и T не приводим к S

# READONLY STRUCT

readonly struct

Запрещает менять состояние экземпляра после его создания

# REF STRUCT

ref struct

Запрещает экземпляру структуры попасть из стека в кучу

# Как?

- Нельзя использовать в качетсве элементов массивов
- Нельзя использовать в качетсве типа члена экземпляра класса и не-ref структуры
- Нельзя реализовыват интерфейсы
- Нельзя упаковывать в виде Object или VelueType
- Есть ограничения при использовании в асинхронных методах
- Нельзя использовать в итераторах

# СТРУКТУРА ИЛИ КЛАСС?

## Рекомендации Microsoft

## Выбирайте структуру, если

- Экземпляры небольшие
- Экземпляры обычно короткоживущие
- Экземпляры часто вложены в другие объекты

Не выбирайте структуру, если тип не обладает хотя бы одной из характеристик:

- Логически представляет собой одно значение (как int, double)
- Экземпляр занимает не больше 16 байт
- Значение неизменяемо
- Не понадобится частая упаковка/распаковка

# СОВЕТЫ ПО ПРОЕКТИРОВАНИЮ СТРУКТУР

# Не создавайте изменяемые структуры

```
Point p1 = new Point(13, 17);
Point p2 = p1;
p1.X = 10;
```

Убедитесь, что значение по умолчанию является корректным

# Реализуйте IEquatable`t

```
struct Point : IEquatable<point>
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public bool Equals(Point other)
    {
        return X == other.X && Y == other.Y;
    }
`
```

# Переопределите Object.Equals и ToString()

```
struct Point : IEquatable<Point>
{
    // ...
    public override bool Equals(object obj)
    {
        return obj is Point
            && ((Point)obj).X == X
            && ((Point)obj).Y == Y;
    }
}
```

# СТРУКТУРА КАК ДОМЕННЫЙ ТИП

## Примеры: широта и долгота, вес и рост, количество

- Можно использовать double
- Можно реализовать структуры: Latitude, Longitude, Weight, Height, Quantity
- Что выбрать?

## Использовать встроенные типы?

+ Меньше кода
- Выше требования к именованию и наличию документации
- Выше шанс пропустить ломающее изменение в библиотеках: изменение порядка параметров метода

## Реализовывать структуры?

- – Больше кода
- \+ Типы помогают писать код
- \+ Проверки значений вовремя - fail fast
- – Невозможно перепутать широту и долготу, скорость и расстояние и т.д.

# Checklist для структур

- ✓ Поля со значениями - private
- ✓ Операторы == и != для своего типа
- ✓ Арифметические операторы - по необходимости
- ✓ Операторы неявного приведения из числовых типов
- ✓ Equals - для более эффективного сравнения
- ✓ GetHashCode - для использования в качестве ключей в словарях
- ✓ ToString - для удобства распечатки значений

# ПЕРЕЧИСЛЕНИЯ

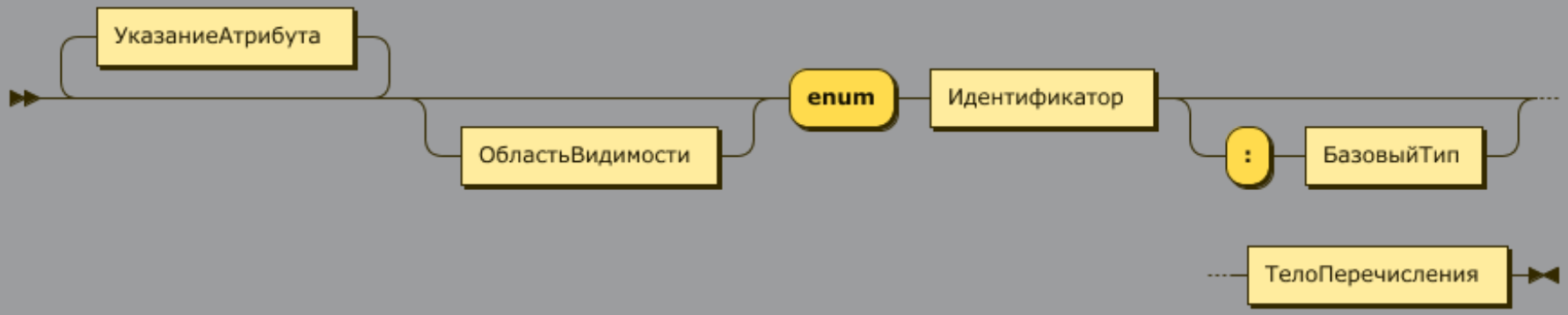
# ПЕРЕЧИСЛЕНИЯ

Перечисление - это набор именованных констант

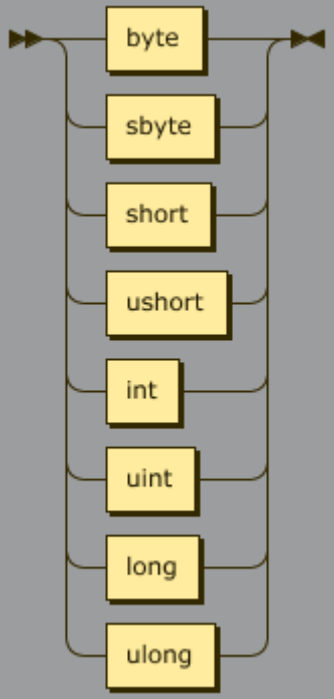
# СИНТАКСИС

```
ОбъявлениеПеречисления
    ::= УказаниеАтрибута* ОбластьВидимости? 'enum' Идентификатор (':' Базо
        ТелоПеречисления
    ::= byte | sbyte | short | ushort | int | uint | long | ulong
ТелоПеречисления
    ::= '{' СписокЧленовПеречисления '}'
СписокЧленовПеречисления
    ::= ЧленПеречисления | ЧленПеречисления ',' СписокЧленовПеречисления
ЧленПеречисления
    ::= Идентификатор ('=' КонстантноеВыражениеБазовогоТипа)?
```

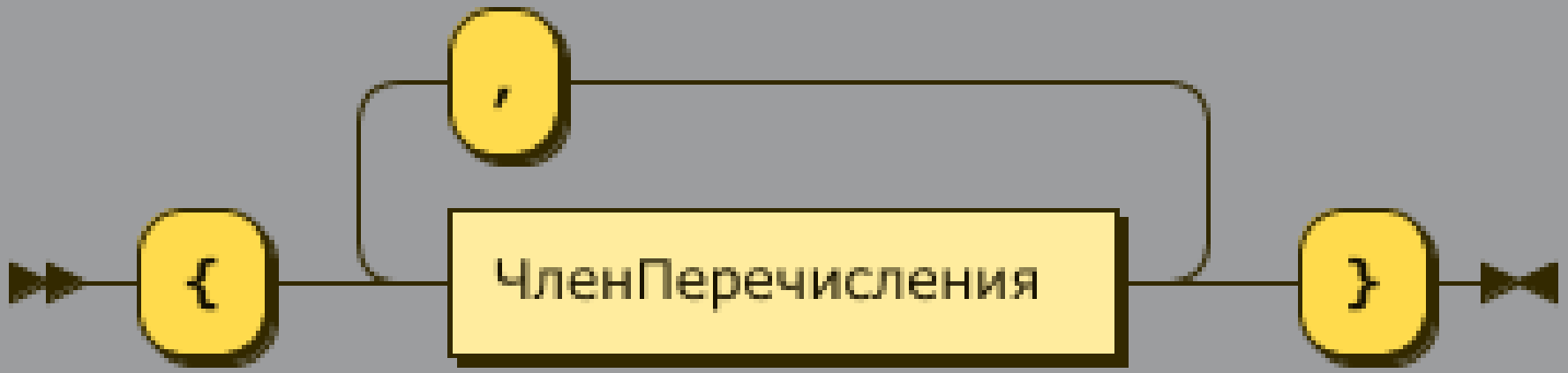
# ОбъявлениеПеречисления:



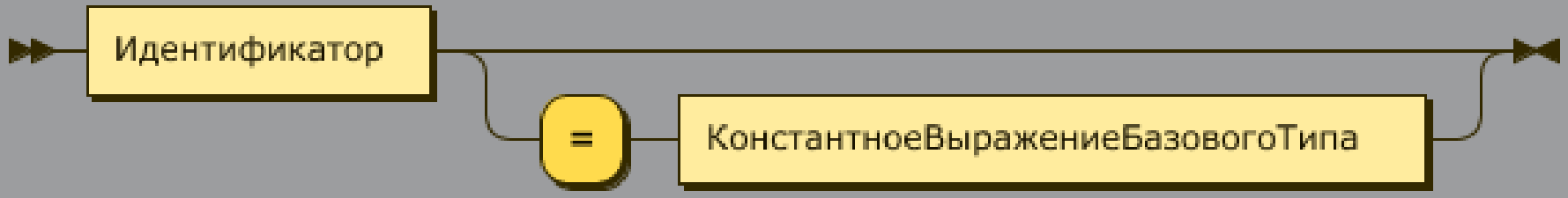
# БазовыйТипПеречисления:



ТелоПеречисления:



ЧленПеречисления:



# БЕЗ БАЗОВОГО ТИПА И ЗНАЧЕНИЙ

```
enum Color
{
    Red,    //  0
    Green,  //  1
    Blue    //  2
}
```

# БЕЗ БАЗОВОГО ТИПА, С ЯВНЫМ УКАЗАНИЕМ ЗНАЧЕНИЙ

```
enum Color
{
    Red = 0,    //  0
    Green = 1,  //  1
    Blue = 2    //  2
}
```

# БЕЗ БАЗОВОГО ТИПА, С ЯВНЫМ УКАЗАНИЕМ ОДНОГО ЗНАЧЕНИЯ

```
enum Color
{
    Red = 17,     //  17
    Green,        //  18
    Blue          //  19
}
```

# БЕЗ БАЗОВОГО ТИПА, С ЯВНЫМ УКАЗАНИЕМ ОДНОГО ЗНАЧЕНИЯ

```
enum Roles
{
    Anonymous,                  //  0
    Operator = 10,              //  10
    Manager,                    //  11
    TopManager,                 //  12
    Administrator = 1000,       //  1000
    SeniorAdministrator,        //  1001
}
```

# С БАЗОВЫМ ТИПОМ, С ЯВНЫМ УКАЗАНИЕМ ОДНОГО ЗНАЧЕНИЯ

```
enum FileMode : ushort
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6
}
```

- Все перечисления неявно наследуют от класса System.Enum
- Класс System.Enum наследует от System.ValueType
- Пользовательским типам запрещено наследовать от System.Enum

## ЧЛЕНЫ System.Enum

- Type GetUnderlyingType(Type enumType)
- TypeCode GetTypeCode()
- bool HasFlag(Enum flag)

## ЧЛЕНЫ System.Enum

- static string GetName(Type enumType, object value)
- static string[] GetNames(Type enumType)
- static bool IsDefined(Type enumType, object value)

## ЧЛЕНЫ System.Enum

- static object Parse(Type enumType, string value)
- static object Parse(Type enumType, string value, bool ignoreCase)
- static bool TryParse<TEnum>(string value, bool ignoreCase, out TEnum result)
- static bool TryParse<TEnum>(string value, out TEnum result)

# ENUM, КОТОРЫЙ МОЖНО ИСПОЛЬЗОВАТЬ КАК ФЛАГИ

```
[Flags]
enum FileAccessMode : ushort
{
    Read = 1
    Write = 2,
    ReadWrite = Read | Write
}
```

# ВОПРОСЫ