

# ЛЕКЦИЯ 6

- 20.09.2022
- Перегрузка методов

# ЦЕЛИ ЛЕКЦИИ

- Познакомится с особенностями передачи параметров по ссылке в методы на языке C#
- Познакомимся с перегрузкой методов C#
- Познакомится с локальными функциями в C#
- Поговорим о статических классах и статических членах классов



Это изображение, автор: Неизвестный автор, лицензия: CC BY-NC

# МОДИФИКАТОРЫ ПАРАМЕТРОВ ДЛЯ ПЕРЕДАЧИ ПО ССЫЛКЕ

out  
in  
ref



# МОДИФИКАТОР REF

Параметр, отмеченный **ref**, указывает, что операции выполняются непосредственно с аргументом (не с его копией)

```
class Program
{
    static void MoveF(ref double fVal, double c) => fVal += c;

    static void Main(string[] args)
    {
        double x = 1;
        double y = 2 * x;

        // Вычисляем 2*x + 3
        MoveF(ref y, 3);
    }
}
```

При вызове у  
аргумента  
сохраняется **ref**

Значение аргумента  
может (**но не обязательно**)  
быть изменено в теле  
метода

# МОДИФИКАТОР REF: ОШИБКИ КОМПИЛЯЦИИ

```
class Program
{
    static void MoveF(ref double fVal, double c) => fVal += c;

    static void Main(string[] args)
    {
        double x = 1;
        double y;
        MoveF(ref y, 3);
    }
}
```

Инициализация  
отсутствует

Передать с модификатором **ref** можно поле или проинициализированную локальную переменную

Ошибка компиляции



# МОДИФИКАТОР IN

Параметр, отмеченный **in**, указывает, что в операциях участвует непосредственно аргумент (не его копия), но **изменить значение аргумента в методе нельзя**

```
class Program
{
    static void MoveF(ref double fVal, in double c) => fVal += c;

    static void Main(string[] args)
    {
        double x = 1;
        double y = 2*x;
        double k = 3;

        MoveF(ref y, k); // MoveF(ref y, in k);
    }
}
```

При вызове у аргумента **in** может отсутствовать или присутствовать, если передано поле или проинициализированная локальная переменная

# МОДИФИКАТОР IN: ОШИБКИ КОМПИЛЯЦИИ

```
class Program  
{
```

```
    static void MoveF(ref double fVal, in double c) => fVal += c;
```

```
    static void Main(string[] args)  
    {  
        double x = 1;  
        double y = 2*x;  
  
        MoveF(ref y, in 3);  
    }
```

ДЛЯ КОНСТАНТ И ВЫЗОВОВ  
МЕТОДОВ **in** НЕ УКАЗЫВАЮТ

```
    static void Main(string[] args)  
    {  
        double x = 1;  
        double y = 2*x;  
        double k;  
  
        MoveF(ref y, in k);  
    }
```

не  
проинициализирована

ошибка компиляции

# ЗАГОЛОВКИ С ОПЦИОНАЛЬНЫМИ ПАРАМЕТРАМИ

```
// Строковый литерал - константа.  
static void Meth1(string str = "default") { }  
// Math.PI - const.  
static void Meth2(double pi = Math.PI) { }  
// null - константа.  
static void Meth3(string str = null) { }  
// Модификатор in допускает передачу констант.  
static void Meth4(in string result = "DEBUG") { }  
// Аргументам по умолчанию нельзя передавать значения при создании.  
static void Meth5(DateTime dt = new()) { }  
// Использование значения struct по умолчанию.  
static void Meth6(BigInteger num = default) { }
```



# МОДИФИКАТОР OUT

Параметр, отмеченный **out**, указывает, что в операциях участвует непосредственно аргумент (не его копия), и **аргумент в методе должен получить значение (обязательно!)**

```
static void MoveF(ref double fVal, in double c, out int fInt)
{
    fVal += c;
    fInt = (int)fVal;
}
```

```
static void Main(string[] args)
{
    double x = 1;
    double y = 2*x;
    double k = 3;
    int intF;
```

```
    MoveF(ref y, in k, out intF);
```

```
}
```

При вызове у аргумента сохраняется **out**

При вызове у аргумента **out** нет обязательств по инициализированности

# МОДИФИКАТОР OUT: ОШИБКИ КОМПИЛЯЦИИ

```
class Program
{
    static void MoveF(ref double fVal, in double c, out int fInt) =>
        fVal += c + fInt;

    static void Main(string[] args)
    {
        double x = 1;
        double y = 2*x;
        double k = 3;
        int intF;

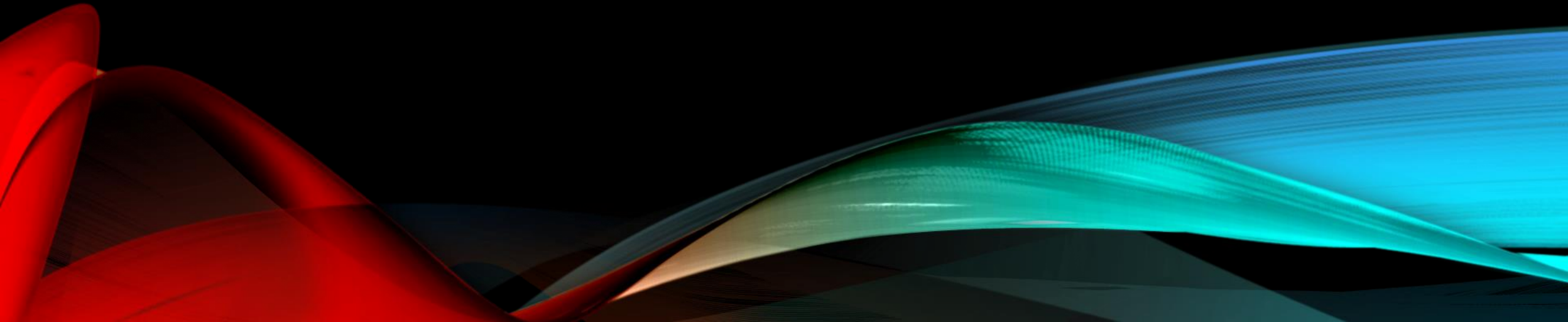
        MoveF(ref y, in k, out intF);
    }
}
```

Переменная **fInt** обязана получить значение в методе, но не получает. Ошибка компиляции

**out** не обязывает аргумент быть инициализированным, ошибка компиляции

# ПЕРЕДАЕМ ПРОИЗВОЛЬНОЕ КОЛИЧЕСТВО ЭЛЕМЕНТОВ

Модификатор params



# МОДИФИКАТОР PARAMS

**params** – модификатор параметров метода, позволяющий передать переменное число аргументов

```
static string MultStr(int n = 0, params string[] str)
{
    string output = "";

    for(int i = 0; i < n; i++)
    {
        output += str[i];
    }
    return output;
}
```

Тип параметра  
обязательно  
одномерный массив

Передаём  
массив

Передаём  
список

Ничего не  
передаём

```
string[] test = { "Bar", "Foo", "13" };
Console.WriteLine(MultStr(2, test));
Console.WriteLine(MultStr(3, "a", "bc", "def", "ghij", "klmno"));
Console.WriteLine(MultStr());
```

# ПАРАМЕТР ИСПОЛЬЗОВАНИЯ PARAMS

```
using System;

int[] arr = { 1, 2, 3, 4, 5 };
int sum3 = Sum(3, 4, 5);    // Массив формируется неявно.
int sum5 = Sum(arr);        // Массив явно передаётся в метод.
Console.WriteLine($"sum3 = {sum3}; sum5 = {sum5}");

static int Sum(params int[] elems) {
    int sum = 0;
    foreach (int elem in elems) {
        sum += elem;
    }
    return sum;
}
```

**Результат выполнения:**  
sum3 = 12; sum5 = 15

# ПОРЯДОК УКАЗАНИЯ ПАРАМЕТРОВ В ЗАГОЛОВКЕ МЕТОДА

Обязательные  
параметры:

Опциональные  
параметры:

Массив с params:

( int x, decimal y, ... int op1 = 17, double op2 = 36, ... params int[] intVals )



# СРАВНЕНИЕ МОДИФИКАТОРОВ ПАРАМЕТРОВ

Модификатор	Нужен при вызове?	Что может быть аргументом	Особенности
<b>&lt;пусто&gt;</b>	—	Всё кроме неинициализированных переменных.	Передача по значению.
<b>ref</b>	да	Поля и инициализированные локальные переменные.	Передача по ссылке, допустимы изменения.
<b>out</b>	да	Поля и любые локальные переменные.	Передача по ссылке, требуется инициализация.
<b>in</b>	не всегда, иногда недопустим	Всё кроме неинициализированных переменных.	Передача по ссылке (для инициализированных переменных и полей) и по значению для всего остального, изменения запрещены.
<b>params</b>	нет	Одномерный массив или последовательность значений, приводимых к указанному типу.	Передача по значению. Позволяет явно передать массив или сформировать его из набора аргументов, перечисленных через запятую.

# ОШИБОЧНЫЕ ЗАГОЛОВКИ С ОПЦИОНАЛЬНЫМИ ПАРАМЕТРАМИ


```
// DateTime.Now - значение этапа выполнения.  
static void Meth1(DateTime dt = DateTime.Now) { }  
  
// MyClass - не struct или enum.  
static void Meth2(MyClass mc = new MyClass()) { }  
  
// params не может иметь значения по умолчанию.  
static void Meth3(params int[] data = null) { }  
  
// Параметры с модификаторами ref и out не могут быть опциональными.  
static void Meth4(ref string result = "Источник") { }  
  
// Аргументам по умолчанию нельзя передавать значения при создании.  
static void Meth5(DateTime dt = new DateTime(2021, 9, 19)) { }
```

# ПЕРЕГРУЗКА МЕТОДОВ



# ВОПРОС

```
static void SwapDouble(ref double x, ref double y)
{
    x += y;
    y = x - y;
    x -= y;
}
```



```
double tmp;
tmp = x;
x = y;
y = tmp;
```

```
double x = 2.7;
double y = 3.13334;
```

```
Console.WriteLine($"x = {x} y = {y}");
SwapDouble(ref x, ref y);
Console.WriteLine($"x = {x} y = {y}");
```

```
x = 2,7 y = 3,13334
x = 3,13334 y = 2,69999999999999997
```

1. Почему такой вывод?
2. Как исправить?
3. Можно ли обменять целочисленные переменные в этом методе?

# РАЗБИРАЕМСЯ С ПЕРЕДАЧЕЙ ССЫЛОЧНОГО ПАРАМЕТРА

```
static void SwapDouble(ref double x, ref double y)
{
    double tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
int x = 0, y = 1;
Console.WriteLine($"x = {x} y = {y}");
SwapDouble(ref x, ref y);
Console.WriteLine($"x = {x} y = {y}");
```

тип **ref int** не  
приводим к  
**ref double**

**Решение:**

```
static void SwapInt(ref int x, ref int y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
int x = 0, y = 1;

Console.WriteLine($"x = {x} y = {y}");
SwapInt(ref x, ref y);
Console.WriteLine($"x = {x} y = {y}");
```

# ИСПОЛЬЗУЕМ ПЕРЕГРУЗКУ МЕТОДА

```
static void Swap(ref double x, ref double y)
{
    double tmp;
    tmp = x;
    x = y;
    y = tmp;
}

static void Swap(ref int x, ref int y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Имя одно

Число или типы параметров разные

```
double xD = 2.7, yD = 3.13334;
int xI = 0, yI = 1;
```

```
Console.WriteLine($"x = {xD} y = {yD}");
Console.WriteLine($"x = {xI} y = {yI}");
```

```
Swap(ref xD, ref yD);
Swap(ref xI, ref yI);
```

```
Console.WriteLine($"x = {xD} y = {yD}");
Console.WriteLine($"x = {xI} y = {yI}");
```



# СИГНАТУРА

- **Сигнатура** [signature] **подпрограммы** – информация, достаточная для правильной организации вызова подпрограммы
- В сигнатуру входит соглашение о вызове и описание всех формальных параметров (включая возвращаемое значение в случае функции)



Это важно, потому что для методов  
это не так

# СИГНАТУРА МЕТОДА

*Заголовок метода*



**Сигнатура метода – это идентификатор +**

- количество параметров
- типы параметров и их порядок следования
- модификаторы параметров (in, out, ref)

В отличие от заголовка, интересующего программиста, сигнатура в первую очередь нужна компилятору для различения методов

# ПЕРЕГРУЗКА И ПОЛИМОРФИЗМ

- **Перегрузка** [subroutine overloading] – наличие в рамках одного пространства имён разных подпрограмм с одинаковыми именами, если их списки параметров различаются
- Перегрузка метода – это проявление **статического полиморфизма**
  - на этапе компиляции программы будет выбрана наиболее подходящая версия, и именно она уйдёт скомпилированную сборку



# НЕКОРРЕКТНАЯ ПЕРЕГРУЗКА

```
static void Swap (out double x, ref double y)
{
    double tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
static void Swap (ref double x, ref double y)
{
    double tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
static void Swap (ref double x, ref double y)
{
    double tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
static void Swap (double x, double y)
{
    double tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

# НЕ ПЕРЕГРУЗКА, А ОШИБКА

```
static void Swap(int x, int y)
{
    x += y;
    y = x - y;
    x -= y;
}
```

Реализация в теле методов разная, а сигнатура совпадает

Два метода с одинаковой сигнатурой, описанные в одном типе приводят к ошибке компиляции

```
static void Swap(int x, int y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Если во втором или первом методе были ссылочные типы, то проблема бы исчезла, сигнатуры - разные

НЕМНОГО ОТДОХНЁМ





# ПРОВЕРЯЕМ СЕБЯ 1

В классе описан метод:

```
static long AddValues(int a, ref int b) => a + b;
```

Пять программистов получили тестовое задание добавить в тот же класс одну перегруженную версию этого метода. Все программисты предложили решение, но в некоторых случаях код перестал компилироваться.

Укажите все перегрузки, в которых программисты ошиблись:

- 1) `static long AddValues(int a, int b) => a + b;`
- 2) `static long AddValues(int a, out int b) { b = 1; return a + b; }`
- 3) `static long AddValues(long h, long m) => h + m;`
- 4) `static long AddValues(float f, float g) => (long)(f + g);`
- 5) `static long AddValues(int c, int d, int e) => c + d + e;`

# ПРОВЕРЯЕМ СЕБЯ 2

В классе описан метод с заголовком:

```
static void Swap(int x, double y)
```

Программистам в качестве тестового задания предложили составить один заголовок метода, но с отличающейся от указанного сигнатурой. Все программисты предложили решение, но некоторые решения оказались ошибочными и содержали заголовки с такой же сигнатурой.

Укажите все ошибочные заголовки:

- 1) `int Add(int z, double g)`
- 2) `static void Swap(int x, double y)`
- 3) `static int Swap(int x, double y)`
- 4) `static void Swap(ref int x, double y)`
- 5) `static void Swap(double x, double y)`

# ЛОКАЛЬНЫЕ ФУНКЦИИ



# ЛОКАЛЬНАЯ ПОДПРОГРАММА

- **Локальная (вложенная) подпрограмма** [nested subroutine] – подпрограмма, объявление которой находится внутри объявления другой подпрограммы, которая становится её областью видимости
- **Локальная функция** – (в С#) это метод, полностью вложенный в другой метод
  - Вызов локальной функции может осуществить только содержащий ее код

метод

конструктор

аксессор

аксессор события

финализатор

анонимный метод

лямбда-выражение

локальная функция (другая)

# ПРИМЕР. СТАТИЧЕСКАЯ ЛОКАЛЬНАЯ ФУНКЦИЯ

У локальных функций не может быть  
модификатора доступа

```
static void Main(string[] args)
{
    string str = "Default value"; // Лок. переменная с инициализацией.
    int x;                       // Лок. переменная без инициализации.

    Console.WriteLine(str);      // Так можно.
    // Console.WriteLine(x);     // А так нельзя, переменная не инициализирована.
    Console.WriteLine(StrValConcat(str, x = 15));
    Console.WriteLine(x);

    static string StrValConcat(string s, int a)
    {
        string str = "In method StrValConcat::";
        return str + s + a;
    }
}
```

Теперь **StrValConcat()** –  
локальная функция для  
**Main()**

**StrValConcat()** – не имеет  
доступа к локальным  
переменным **Main()**

# ПРИМЕР. НЕСТАТИЧЕСКАЯ ЛОКАЛЬНАЯ ФУНКЦИЯ

```
static void Main(string[] args)
{
    string str = "Default value"; // Лок. переменная с инициализацией.
    int x;                        // Лок. переменная без инициализации.

    Console.WriteLine(str);      // Так можно.
    // Console.WriteLine(x);     // А так нельзя, переменная не инициализирована.
    Console.WriteLine(StrValConcat(str, x = 15));
    Console.WriteLine(x);
}
```

```
string StrValConcat(string s, int a)
{
    string str = "In method StrValConcat::";
    return str + s + a + x;
}
```

Теперь **StrValConcat()** –  
локальная функция для  
**Main()**

**StrValConcat()** – имеет  
**доступ** к локальным  
переменным **Main()**



# НЕМНОГО О КОНСТАНТАХ И ДРУГИХ СТАТИЧЕСКИХ ЧЛЕНАХ КЛАССОВ



# СТАТИЧЕСКИЙ КЛАСС

- Нельзя создавать экземпляры статического класса
- Содержит только статические методы
- Позволяет обращаться к своим членам через имя класса
- Не содержит экземплярных конструкторов
- Не может быть унаследован

```
static class Methods  
{  
  
}
```

# ПРИМЕР СТАТИЧЕСКОГО КЛАССА

```
static class Methods  
{  
    public static double Calc(double x) => x;  
    public static double Calc(double x, double y) => x + y;  
}
```

```
Console.WriteLine(Methods.Calc(3));  
Console.WriteLine(Methods.Calc(3,7));
```

# КОНСТАНТА

- **Константа** [constant] – именованный элемент данных, значение которого задаётся при объявлении и затем не может изменяться. Имя константы не может появляться в левой части оператора присваивания
- В С# **константы** — это постоянные значения, которые известны во время компиляции и не изменяются во время выполнения программы

**const**

встроенный тип

идентификатор

= значение;

Или System.Object

# МОДИФИКАТОР CONST В C#

```
class Program
```

```
{
```

```
// Константы доступны по имени типа.
```

```
const int ten = 10;
```

```
static void Main()
```

```
{
```

```
const double PI = 3.1415; // локальная константа.
```

```
System.Console.WriteLine($"PI * ten = {PI * ten}");
```

```
}
```

```
}
```

КОНСТАНТА ВИДНА ВСЕМ  
МЕТОДАМ В КЛАССЕ  
**Program**

КОНСТАНТА ЛОКАЛЬНА ДЛЯ **Main()**  
И ДОСТУПНА В НЁМ И ЛОКАЛЬНЫХ  
ФУНКЦИЙ **Main()**

Модификатор **const** используется в C# для констант уровня компиляции и требует обязательной инициализации константы в момент определения

# СТАТИЧЕСКИЙ КОНСТРУКТОР

```
using System;
class MyClass {
    // num инициализируется перед вызовом статического конструктора.
    static int num = 42;
    static MyClass() { // Статический конструктор: static <Имя Типа>.
        Console.WriteLine($"The answer is... {num}!");
        Console.WriteLine($"Starting time: {DateTime.Now}");
    }
}
```

**Статический конструктор** – специальный функциональный член класса, вызывающийся автоматически при первом обращении к типу сразу после инициализации всех статических полей



# МОДИФИКАТОРЫ ДОСТУПА

**Модификаторы доступа** определяют, можно ли обращаться к определённому типу/члену типа за его пределами

**Для констант можно использовать:**

- **private** – закрытый (доступ только внутри типа)
- **public** – открытый (доступ без ограничений)
- **protected** – защищенный (доступ только для наследников)
- **internal** – внутренний (доступ внутри той же сборки)
- **protected internal** – внутри сборки ИЛИ для всех наследников
- **private protected** – только для наследников внутри той же сборки (C# 7.2)

По умолчанию члены пространств имён *неявно* имеют модификатор **internal**, а все члены классов – **private**

# СТАТИЧЕСКИЕ ПОЛЯ

- **Статическое поле** – это такое поле класса, которое доступно для обращения даже при отсутствии объектов классов
  - Существует только одна копия статического поля
  - В статических полях иногда сохраняют данные, общие для всех объектов типа

`static`

тип

идентификатор

[ = значение];

Статические локальные переменные в языке не  
поддерживаются

# ДОСТУПНОСТЬ ЧЛЕНОВ ВНЕ КЛАССА

Допускает вызов метода за пределами **MyClass**

```
class MyClass {  
    static long num = 13;  
    public static void Print(int u) {  
        long prod = u * num;  
        System.Console.WriteLine("u * numb = " + prod);  
    }  
}  
  
class Program {  
    static void Main() {  
        MyClass.Print(3);    // Обращение к Print по имени типа.  
    }  
}
```

По умолчанию **private**, обратится за пределами **MyClass**

# ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Материалы лекций по С# Подбельского В.В., Дударева В.А
- Richter, J. CLR via C#. Fourth edition
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/in-parameter-modifier>
- <https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/ref>
- <https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/params>
- <https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/out-parameter-modifier>
- <http://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/how-to-define-constants>