

# ЛЕКЦИЯ 5

- 20.09.2022
- Структурное программирование
- Статические методы в C#

# ЦЕЛИ ЛЕКЦИИ

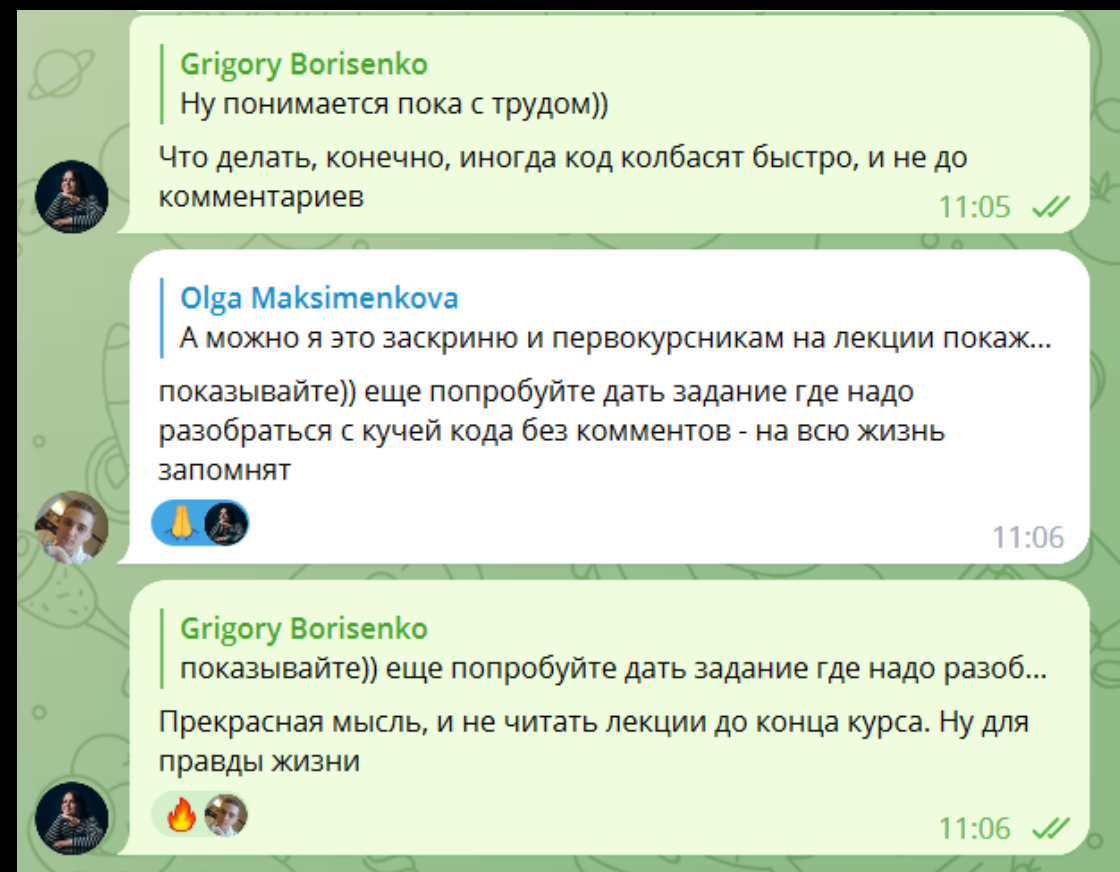
- Повторить, что такое структурное программирование, принципы
- Разобраться с особенностями декомпозиции и структурного проектирования
- Познакомится со статическими методами в С#
- Обсудить особенности передачи параметров в методы



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



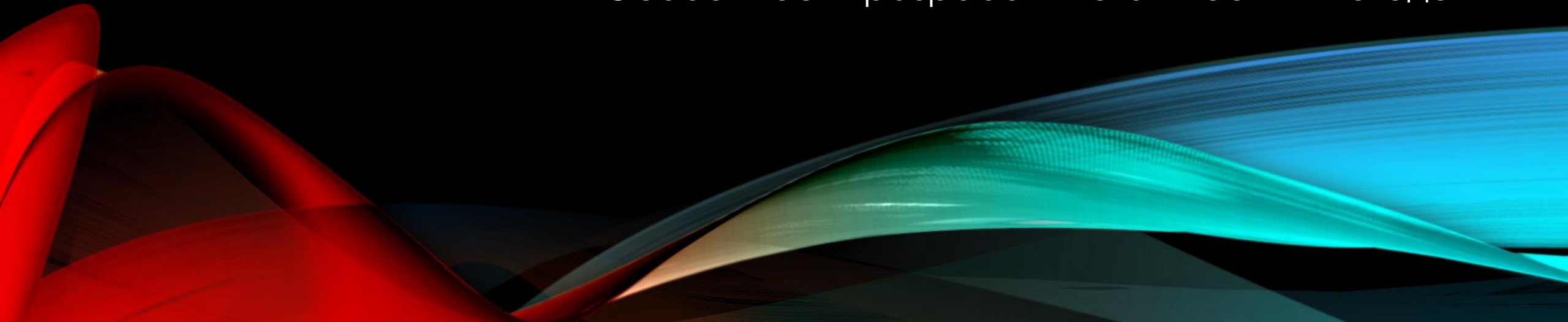
# ПРИВЕТ ОТ ГРИШИ



# СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

Термины

Особенности разработки статических методов





# СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

**Структурное программирование** [structured programming] – методология и технология разработки программных средств, основанная на использовании трёх базовых конструкций (следование, ветвление и цикл) и принципах:

- абстракции и инкапсуляции процессов в виде подпрограмм
- программирования «сверху-вниз» (нисходящего проектирования)
- модульного программирования с иерархическим упорядочением связей между модулями

# АБСТРАКЦИЯ

**Абстракция** [abstraction] –

- разграничение внешних (существенных с точки зрения надсистем) свойств системы и внутренних деталей её строения и функционирования
- принцип моделирования, заключающийся в игнорировании аспектов проблемы, не оказывающих существенного влияния на её решение. Хорошей абстракцией считается та, которая выделяет свойства системы, не зависящие от реализации подсистем и действительно существенные для рассмотрения и использования в данном контексте, опуская все остальные

При разработке информационных систем используется как минимум два вида абстракции:

- абстракция процессов
- абстракция данных



# АБСТРАКЦИЯ ПРОЦЕССА

- **Абстракция процесса** позволяет не углубляться в детали реализации, а рассматривать достаточно сложное действие как атомарное на данном уровне абстракции
  - Подпрограммы - пример абстракции процесса

# ИНКАПСУЛЯЦИЯ

- **Инкапсуляция** [encapsulation] - механизм процесса абстрагирования, обеспечивающий доступность интерфейса системы на заданном уровне абстракции, с одновременным сокрытием всего второстепенного (мешающего, затрудняющего описание) в некий чёрный ящик

Процедурная инкапсуляция используется в процедурной парадигме в виде подпрограмм, которые скрывают детали реализации части алгоритма





# ДЕКОМПОЗИЦИЯ

- При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо
- **Алгоритмическая декомпозиция** [algorithmic decomposition] - процесс разделения системы на части, каждая из которых отражает этап общего процесса



# СТРУКТУРНОЕ ПРОЕКТИРОВАНИЕ

- **Структурное проектирование по методу сверху вниз**
  - основной базовой единицей является подпрограмма, и программа в целом принимает форму дерева, в котором одни подпрограммы в процессе работы вызывают другие подпрограммы
- **SADT (Structured Analysis and Design Technique)**
  - методология структурного анализа и проектирования, интегрирующая процесс моделирования, управление конфигурацией проекта, использование дополнительных языковых средств и руководство проектом со своим графическим языком

# ЕДИНИЦА ДЕКОМПОЗИЦИИ

- Единицей декомпозиции исходного кода программ является **программный модуль** [program module, unit] – специально оформленная часть исходного кода, являющаяся единицей хранения, трансляции, объединения с другими модулями и загрузки в оперативную память

Строго говоря, для С# напрямую не применимо, т.к. мы всегда имеем дело с объектами и не можем избежать их использования

# СТАТИЧЕСКИЕ МЕТОДЫ

Структура метода  
Описание и вызовы



# ПОДПРОГРАММА

- **Подпрограмма** [subroutine] – именованная часть программы, которая разрабатывается относительно независимо от других частей и может быть многократно вызвана (выполнена) по своему имени.
- Исходный код, включённый в подпрограмму, называется её **телом** [body].
- **Метод** – инкапсулированный набор команд, выполняющих конкретное задание [task]
  - Описать метод в C# можно в классе, структуре или интерфейсе



# ОПИСАНИЕ МЕТОДА

## Заголовок метода

модификатор  
доступа

static

тип

идентификатор

( список формальных  
параметров )

## Тело метода

{

операторы

}

# ПРИМЕР

```
static void MethId1()  
{  
  
}
```

```
static int MethId3(int x)  
{  
    return x++;  
}
```

```
static void MethId4(int x)  
{  
  
}
```

```
static int MethId2()  
{  
    return 1;  
}
```

# ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
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);
}
```

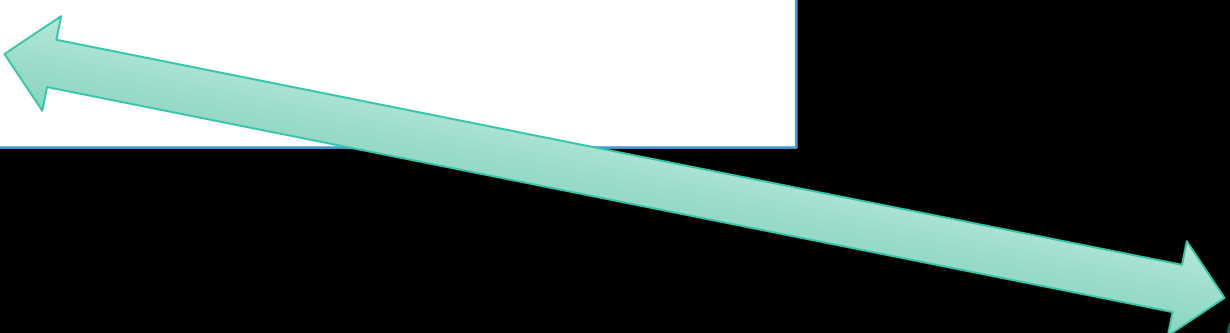
Назначение локальной  
переменной метода **Main()**.  
Делать так не надо

Локальная переменная  
метода **StrValConcat()**

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

# ЛЯМБДА-ОПЕРАЦИЯ В МЕТОДАХ

```
static long AddValues(int a, int b)
{
    return a + b;
}
```



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

Попробуйте переписать с лямбда-операцией методы, возвращающие значение с предыдущего слайда

# ПАРАМЕТРЫ И АРГУМЕНТЫ

```
static double SumSquare(double l, double r)
{
    return (l + r) * (l + r);
}
```

Вызов метода

```
double x = 1, y = 5;
double z = SumSquare(x, y);
Console.WriteLine(z);
Console.WriteLine(SumSquare(4, 2));
Console.WriteLine(SumSquare(Math.PI, 1));
```

- **Формальный параметр** [formal parameter] – параметр, понимаемый как его описание в заголовке подпрограммы при её объявлении
- **Фактический параметр** [actual parameter] или **аргумент** [argument] – выражение, подставленное вместо формального параметра при вызове подпрограммы



# ИМЕНОВАННЫЕ АРГУМЕНТЫ

При вызове метода допустимы **именованные аргументы** [named arguments], их использование освобождает программиста от соблюдения требования о порядке следования аргументов

```
string str = "123";  
int val;  
int.TryParse(result: out val, s: str);  
Console.Write(val);
```

```
public static bool TryParse([NotNullWhen(true)] string? s, out Int32 result);
```

# ВАРИАНТЫ ПЕРЕДАЧИ ИМЕНОВАННЫХ АРГУМЕНТОВ

явно поименовать все аргументы в произвольном порядке

1

2

поименовать только несколько аргументов, однако порядок их передачи должен совпадать с порядком объявления параметров в заголовке метода

Аргументы, находящиеся на своих местах, без явного указания имени в таком случае называют **ПОЗИЦИОННЫМИ**

# ПРИМЕР. НАРУШЕНИЕ ПОРЯДКА

```
static int Sum(int a, int b, int c) => a + b + c;
```

```
Sum(b: 4, c: 1, a: 3);    // Все аргументы именованные.
```

```
Sum(a: 4, b: 2, 0);    // Аргументы именуется в корректном порядке.
```

```
Sum(a: 6, 4, c: 2);    // Именованные аргументы идут не подряд, порядок корректный.
```

```
Sum(b: 2, a: 1, 5);    // Ошибка компиляции, b не на своём месте.
```

# ПРИМЕР

```
using System;
```

```
Console.WriteLine($"Div(y: 10, x: 3 + 8) = {Div(y: 10, x: 3 + 8)}");
```

```
static int Div(int x, int y) => x / y;
```

**Результат выполнения:**  
Div(y: 10, x: 3 + 8) = 1

Ещё варианты

```
Div(10, y: 3);
```

```
Div(10, x: 3+8)
```

# ПОВЫШЕНИЕ ЧИТАЕМОСТИ КОДА

```
class DemoProgramNamedArguments
{
    static double Volume(double radius, double height) => Math.PI * radius * radius * height;

    static void Main()
    {
        // При чтении непонятны аргументы.
        Console.WriteLine($"Volume 1: {Volume(3.0, 4.0):F3}");

        // Из имён проще догадаться, что это какая-то фигура вращения.
        Console.WriteLine($"Volume 2 (named): " +
            $"{Volume(radius: 4.0, height: 5.0):F3}");
    }
}
```

Volume 1: 113,097  
Volume 2 (named): 251,327



# УМАЛЧИВАЕМЫЕ ЗНАЧЕНИЯ ПАРАМЕТРОВ

- **Параметр по умолчанию** [default parameter] – параметр подпрограммы, для которого можно не указывать аргумент при вызове подпрограммы. В этом случае он принимает значение по умолчанию, которое указывается при объявлении подпрограммы
- В C# используются **опциональные (необязательные) аргументы** [optional arguments], для них значения в параметрах метода заданы по умолчанию

```
static string StrForLabel(string output = "no values...")  
{  
    return "Here:: " + output;  
}
```

```
Console.WriteLine(StrForLabel());  
Console.WriteLine(StrForLabel("Something"));
```

# НЕОБЯЗАТЕЛЬНЫЕ АРГУМЕНТЫ

- Константное выражение
- Выражение `new ТипЗначение()`
- Выражение `default(ТипЗначение)`

Если есть необязательные, то должны быть и обязательные аргументы

```
static string StrForLabel(int t, string str = "no values...")  
{  
    string output = "";  
    for (int i = 0; i < t; i++)  
    {  
        output += str;  
    }  
    return "Here:: " + output;  
}
```

```
Console.WriteLine(StrForLabel(3));  
Console.WriteLine(StrForLabel(2, "Something"));
```

# DEFAULT

## оператор `default`

### `default(аргумент)`

Возвращает значение по умолчанию,  
полученное по аргументу

Аргумент:

- Имя типа
- Параметр типа

## литерал `default`

### `default`

Используется в выражениях для  
формирования значения по умолчанию:

- При назначении переменной, в т.ч. Инициализации
- Как умалчиваемое значение для необязательного параметра метода
- Как аргумент при вызове метода
- При возврате с использованием `return` или при использовании синтаксиса выражений

# ПРИМЕР

```
static string StrForLabel(int t, string? str = default)
{
    string output = "";
    for (int i = 0; i < t; i++)
    {
        output += str;
    }
    return "Here:: " + output;
}
```

Здесь значение по умолчанию - **null**

```
Console.WriteLine(StrForLabel(3));
Console.WriteLine(StrForLabel(2, "Something"));
```

```
Here::
Here:: SomethingSomething
```

# ПРИМЕР

```
class OptionalParamsDemo
{
    static int Calc(int a = 2, int b = 3, int c = 4) => (a + b) * c;
    static void Main()
    {
        int r0 = Calc(5, 6, 7); // Все аргументы указаны явно.
        int r1 = Calc(5, 6); // Аргумент по умолчанию для c.
        int r2 = Calc(5); // Аргументы по умолчанию для c и b.
        int r3 = Calc(); // Аргументы по умолчанию для всех параметров.
        Console.WriteLine($"{r0}, {r1}, {r2}, {r3}");
    }
}
```

**Результат выполнения:**  
77, 44, 32, 20



# ПРИМЕР

```
long res1 = ArithmeticProgressionSum(2, 10);           // step по умолчанию == 1.  
long res2 = ArithmeticProgressionSum(5, 1000, 5);      // step задан явно.  
Console.WriteLine($"Sum 1: {res1}; Sum2: {res2}");
```

```
static long ArithmeticProgressionSum(int first, int last, uint step = 1)  
{  
    long sum = 0;  
    for (long i = first; i < last; i += step)  
    {  
        sum += i;  
    }  
    return sum;  
}
```

**Результат выполнения:**  
Sum 1: 44; Sum2: 99500

# ПЕРЕДАЧА ПАРАМЕТРОВ

По ссылке  
По значению



# ПЕРЕДАЧА ПАРАМЕТРОВ

- **Передача параметров** [parameters passing] – интерпретация того, что происходит при подстановке фактических параметров на место формальных в момент вызова подпрограммы

## По значению [by value]

Тип-значение  
[value type]

По умолчанию, назначение переменной нового значения не видно вызывающему коду

Тип-ссылки  
[reference type]

По умолчанию, ссылка копируется, т.е. назначение новой ссылки в методе не видно вызывающему коду

## По ссылке [by reference]

Назначение значения переменной в методе видно вызывающему коду

требуется  
модификатор **ref**

Назначение новой ссылки в методе видно вызывающему коду

# ПРИМЕР. ПЕРЕДАЕМ ТИП ЗНАЧЕНИЕ

## По значению [by value]

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

```
int a = 6, b = 7;
Console.WriteLine($"a={a} b={b}");
Swap(a, b);
Console.WriteLine($"a={a} b={b}");
```

a=6 b=7  
a=6 b=7

## По ссылке [by reference]

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

```
int a = 6, b = 7;
Console.WriteLine($"a={a} b={b}");
Swap(ref a, ref b);
Console.WriteLine($"a={a} b={b}");
```

a=6 b=7  
a=7 b=6

# ПРИМЕР. ПЕРЕДАЕМ ТИП ССЫЛКИ ПО ССЫЛКЕ

```
static void Swap (ref int[] x, ref int[] y)
{
    int[] tmp = x; // Временная ссылка.
    x = y;
    y = tmp;
}
```

```
int[] arr = { 1, 2, 3 };
int[] arr2 = { 7, 8, 9 };
```

```
Array.ForEach(arr, Console.Write);
Array.ForEach(arr2, Console.Write);
Swap(ref arr, ref arr2);
Console.WriteLine($"{Environment.NewLine}After Swap() calling...");
Array.ForEach(arr, Console.Write);
Array.ForEach(arr2, Console.Write);
```

123789  
After Swap() calling...  
123789

# ВОЗВРАТ ЗНАЧЕНИЯ ИЗ МЕТОДА

return

Возврат значения по ссылке



# ВОЗВРАТ ЗНАЧЕНИЯ

- Для возврата из методов используется оператор безусловного перехода **return**
- **return** прерывает выполнение метода/функции, в котором появился) и передаёт управление и результат работы (при наличии) в вызывающий КОД

```
return;
```

```
return;
```

```
return выражение;
```

```
return sum;
```

```
return 1 + 2;
```



Куда разместить значение,  
возвращённое из метода?

Назначить переменной, использовать в выражении, передать в качестве  
параметра в метод...

# ВОЗВРАТ ЗНАЧЕНИЯ ПО ССЫЛКЕ

- По умолчанию **return** возвращает значение выражения
- Допустимо вернуть на значение переменной с использованием ключевого слова **ref**:
  - **return ref переменная;**

В методе явно указано, что значение вернётся по ссылке

Переменная передана **a** по ссылке

Возврат переменной **a** по ссылке

```
static ref int DbInc(ref int a, int b)
{
    a = ++a + ++b;
    return ref a;
}
```

## Куда разместить значение, по ссылке возвращённое из метода?

Если планируется изменение значения, переданного по ссылке из метода, то обязательно потребуется объявить локальную ссылочную переменную в вызывающем коде

# ССЫЛОЧНЫЕ ПЕРЕМЕННЫЕ

- **Ссылочная переменная** (переменная ссылки или `ref local`) [*reference variable*] – это переменная, которая ссылается на другую переменную – **референта** (referent)

```
int x = 0;  
ref int b = ref x;  
Console.WriteLine(x);  
b++;  
Console.WriteLine(x);
```

Здесь переменная **b** связывается со значением переменной **x** (без копирования, по **b** и **x** просматривается один и тот же участок памяти)

Назначение значения ссылочной переменной изменяет значение переменной-референта

0  
1

# ПРИМЕР

```
static ref int DbInc(ref int a, int b)
{
    a = ++a + ++b;
    return ref a;
}
```

Ссылочная локальная  
переменная – **r-value**

```
static void Main(string[] args)
{
    int a = 5, b = 6;
    ref int c = ref DbInc(ref a, b);
}
```

# ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Introduction to classes [<http://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/classes>]
- Hunt K.P. An introduction to structured programming [[http://www.researchgate.net/publication/225724365\\_An\\_introduction\\_to\\_structured\\_programming](http://www.researchgate.net/publication/225724365_An_introduction_to_structured_programming)]
- Floyd R.W. The paradigms of programming [<http://www.0861.ru/paradigm/pr-1.pdf>]
- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods>
- <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/method-parameters#pass-a-reference-type-by-value>
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments>
- <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/declarations#reference-variables>