В.В. Подбельский

Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

04 Часть 2

Методы в C#

Благодарность Д.Д. Сагалову за помощь в переработке презентаций

Методы. Заголовок Метода

Метод: именованный блок кода, состоящий из заголовка и тела.

Заголовок метода включает:

- *Модификаторы* (например, доступа или static);
- *Тип возвращаемого значения* – тип результата, получаемого в результате работы метода и возвращаемого в точку вызова;
- *Идентификатор* – имя метода;
- *Список параметров* (возможно пустой), заключённый в круглые скобки;

Тело метода – набор последовательно выполняемых операторов (блок кода).

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods

Методы. Сигнатура Метода

Для компилятора намного более важную роль играет **сигнатура метода**:

- Идентификатор (имя) метода;
- Количество параметров;
- Типы параметров и их порядок в списке;
- Модификаторы параметров.

В сигнатуру **не** входит:

- Тип возвращаемого значения метода;
- Модификаторы метода.

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

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods

Перегрузка Методов

Сигнатура позволяет определять **перегрузки** – «варианты» методов *с одинаковым именем*, но при этом различающиеся по сигнатуре (хотя бы по одной из составляющей сигнатуры).

Правило: В объявлении типа (class или struct) не может быть двух методов с одинаковой сигнатурой, иначе возникнет ошибка компиляции.

Обратите внимание: методы, отличающиеся именем уже не являются перегрузками.

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods

Лямбда-операция в методах

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

ЭКВИВАЛЕНТНО

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

Перегрузка Методов

НЕ является частью сигнатуры при перегрузке

↓ ↓ ↓

long **AddValues(int** a, **out int** b**)** { ... }

```
public class SumMethods
{
    static long AddValues(int a, int b) => a + b;
    static long AddValues(int c, int d, int e) => c + d + e;
    static long AddValues(float f, float g) => (long)(f + g);
    static long AddValues(long h, long m) => h + m;
}
```

Заголовки Методов: Примеры

Заголовки методов с разной сигнатурой:

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

int Add(int z, double g)
int Add(double z, int g)
```

Заголовки методов с одинаковой сигнатурой:

```
static void Swap(int x, double y)
// Одинаковые типы параметров – ошибка компиляции.
static void Swap(int y, double x)
// Отличия только в возвращаемом значении – ошибка компиляции.
static int Swap(int x, double y)
```

Управляющие Операторы

• **Операторы ветвления** (selection statements): осуществляют выбор, какой оператор или блок выполнять.

• **Операторы цикла** (iteration statements): позволяют выполнять или итерироваться по оператору/блоку.

• **Операторы перехода** (jump statements): позволяют осуществить переход из одной точки блока/метода в другую.

Операторы Ветвления

- **if**: условное выполнение блока/оператора;
- **if**...**else**: условное выполнение одного блока/оператора или другого;
- **switch**: условное выполнение одного блока из списка.

Операторы Циклов

- **for**: проверка условия перед выполнением;
- **while**: проверка условия перед выполнением;
- **do-while**: проверка условия *после* выполнения;
- **foreach**: выполнение для каждого элемента коллекции.

Операторы Перехода

- **break**: может быть использован только внутри циклов или switch, позволяет выйти из текущего блока;
- **continue**: осуществляет переход к следующей итерации цикла;
- **goto**: осуществляет безусловный переход к именованной метке;
- **return**: завершает выполнение текущего метода.

break

break завершает выполнение ближайшего оператора внешнего цикла или switch, в котором он находится. Управление передается оператору, который расположен после завершенного оператора.

```
for (int i = 1; i <= 100; i++)
{
    if (i == 5)
    {
        break;
    }
    Console.Write(i + " ");
}
```

Результат выполнения:
1 2 3 4

continue

Передает управление следующей итерации вложенного оператора **while**, **do**, **for** или **foreach** в котором она встречается.

```
for (int i = 1; i <= 10; i++)
{
    if (i < 9)
    {
        continue;
    }
    Console.Write(i + " ");
}
```

Результат выполнения:
9 10

Возврат Значений из Методов

```
using System;
class Program
{
    // При использовании сокращённого синтаксиса с => return опускается.
    static int HourNow() => DateTime.Now.Hour;

    static Random rnd;
    static void Main()
    {
        int currentHour = HourNow();
        rnd = new Random(currentHour);
        Console.WriteLine($"time = {currentHout}");
        if (currentHour < 12)
            return;                 // Досрочное завершение Main.
        int res = rnd.Next();
        Console.WriteLine(res);
    }
}
```

Параметры и Аргументы

```
using System;

double w = 39.5;
int res = HeavySide(w);
// Неявное приведение int → double.
double s = HeavySide(44);
Console.WriteLine(s);
Console.WriteLine(Average(res, 5, 7 / 3));

static int HeavySide(double x) => x >= 0 ? 1 : 0;

static double Average(double x, double y, double z)
{
    double sum = x + y + z;
    return sum / 3;
}
```

Аргументы – это значения параметров, используемые при вызове метода.

Целочисленное деление! Результат имеет тип int, приведение результата к double - неявное.

Параметры метода

Результат выполнения:

1

2,6666666666666665

Передача Аргументов по Значению

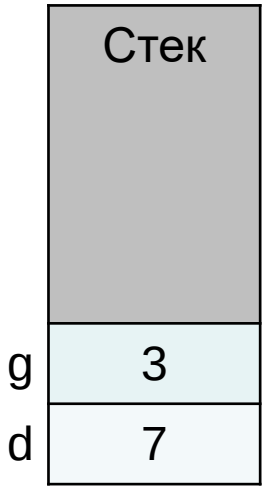
```
using System;

int d = 7, g = 3;
string line = $"{Proc(d, g)}, d = {d}, g = {g}";
Console.WriteLine(line);

static int Proc(int a, int b)
{
    // Аргументы, подставленные в качестве a и b, не изменятся.
    if (a > b)
        return a += b;
    else
        return b -= a;
}
```

Результат выполнения:
10, d = 7, g = 3

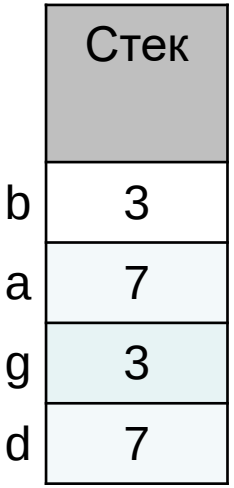
Стек при Вызове Методов

Посмотрим, как выглядит стек вызовов для примера с прошлого слайда:

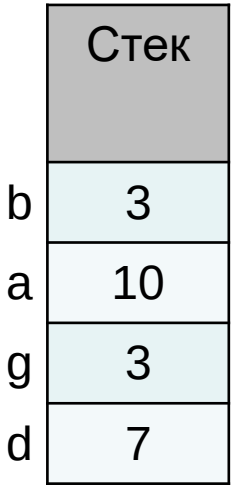
Перед вызовом



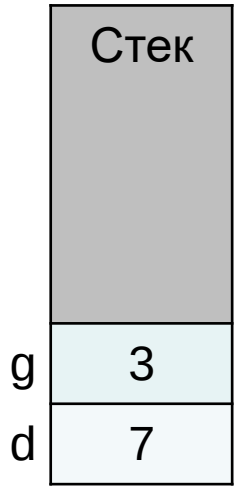
Начало метода



Конец метода



После метода



Запомните: по умолчанию параметры в C# передаются по значению. Для типов значений копируется само значение, для ссылочных типов – ссылка на объект в куче.

Передача Параметров по Ссылке

Для того, чтобы избежать копирования при передаче в метод, в C# используются специальные ключевые слова – **ref**, **in** и **out**.

При передаче значений по ссылке ключевое слово ref/out необходимо указывать каждый раз при передаче аргументов.

Рассмотрим реализацию методов **SwapInt** и **SwapString**:

```
ссылка: 1
static void SwapString(ref string a, ref string b) {
    string tmp = a;
    a = b;
    b = tmp;
}
ссылка: 1
static void SwapInt(ref int a, ref int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
Ссылок: 0
static void Main() {
    string s1 = "left", s2 = "right";
    int i1 = 1, i2 = 2;
    // Не забываем указать ref при вызове методов.
    SwapString(ref s1, ref s2);
    SwapInt(ref i1, ref i2);
    // Выведется: s1: right, s2: left
    Console.WriteLine("s1: " + s1 + ", s2: " + s2);
    // Выведется: i1: 2, i2: 1
    Console.WriteLine("i1: " + i1 + ", i2: " + i2);
}
```

Передача Параметров по Ссылке – ref

При передаче значения по ссылке с помощью **ref**:

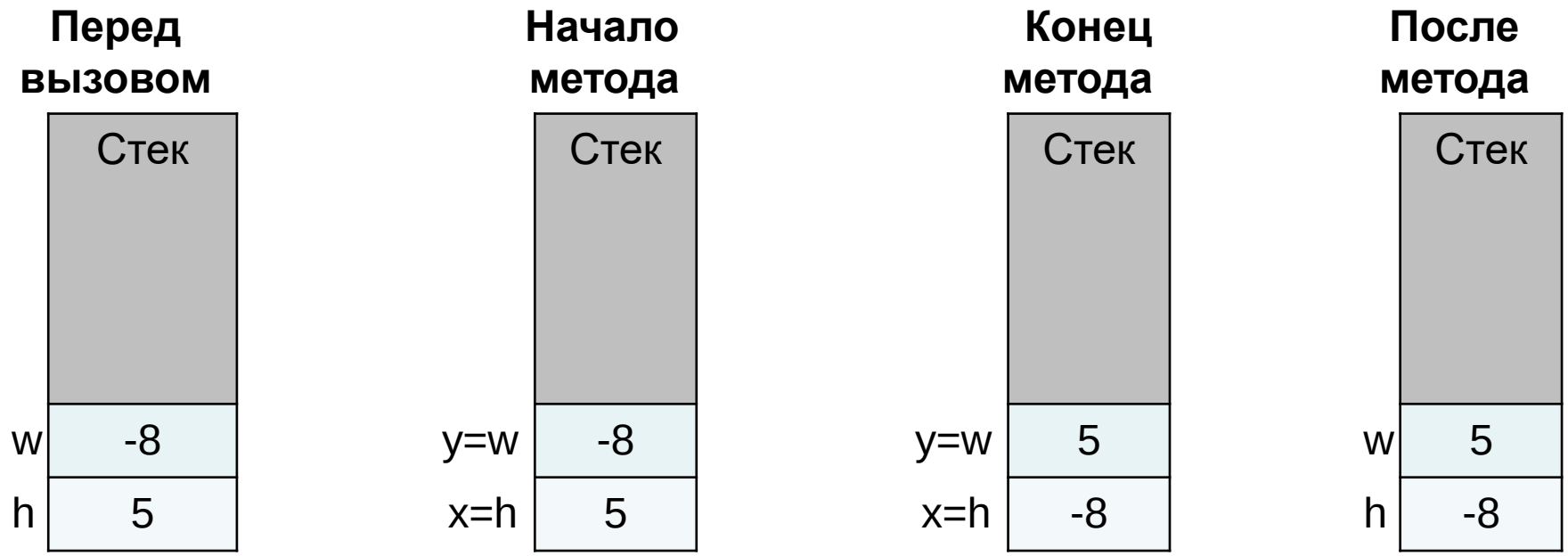
- Метод, в который передаётся значение, может менять его (но не обязан это делать);
- При вызове нужно указать ключевое слово ref явно перед каждым аргументом, помеченным модификатором ref в заголовке;
- Аргументами могут быть **только поля** и **инициализированные локальные переменные**. При использовании неинициализированных локальных переменных возникнет ошибка компиляции.

Передача по Ссылке – ref: Пример

```
using System;

class Program {
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int h = 5, w = -8;
        Swap(ref h, ref w);
        Console.WriteLine($"h = {h}; w = {w}");
    }
}
```

Результат выполнения:
h = -8; w = 5

Стек при Передаче Параметров по Ссылке




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

```
static void Main() {
        int h = 5, w = -8;
        Swap(ref h, ref w);
        Console.WriteLine($"h = {h}; w = {w}");
    }
```

Передача Параметров по Ссылке – out

При передаче значения по ссылке с помощью **out**:

- Метод, в который передаётся аргумент, **ОБЯЗАН инициализировать** его внутри тела перед передачей управления, иначе возникнет ошибка компиляции;

- При вызове метода нужно указать ключевое слово **out** явно.

- Аргументами могут быть **любые поля** и **локальные переменные**, включая неинициализированные.

Начиная с версии C# 7.0, Вы можете объявлять локальную переменную прямо при вызове метода с модификатором out, например: **int.TryParse(Console.ReadLine( ), out int output)**.

Передача по Ссылке – out: Пример

```
using System;

Separate(-5.4, out int temp, out double frac);
Console.WriteLine($"integer = {temp};  fraction = {frac}");

static void Separate(double real, out int whole, out double frac)
{
    whole = (int)real;
    frac = real - integer;
}
```

Результат выполнения:
integer = -5; fraction = -0,4

Передача по Ссылке – out: Ошибка

```
static void Pow(int pow, out double real)
{
    // Ошибка компиляции – не гарантируется, что
    // real – инициализированная переменная.
    real = Math.Pow(real, pow);
}
```

Запомните: подразумевается, что в метод, содержащий параметры с модификатором out могут передаваться неинициализированные значения.

Передача Параметров по Ссылке – in

При передаче значения по ссылке с помощью **in**:

- Метод, в который передаётся аргумент, **не имеет права** менять его;
- При вызове ключевое слово in необязательно и может быть применено только к полям и локальным переменным;
- Аргументами может быть что угодно, кроме неинициализированных локальных переменных.

Компилятор создаёт локальную переменную вместо передачи по ссылке, если в качестве аргументов используются *константы* или *результаты вызовов методов*. В месте вызова писать in в этих случаях нельзя – возникнет ошибка компиляции.

https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/in-parameter-modifier

Модификатор params

params – модификатор параметров метода, позволяющий передать переменное число аргументов через запятую или в виде одномерного массива.

params обладает рядом особенностей:

- Только один параметр может быть помечен модификатором params, причём он должен быть последним в списке параметров;

- При вызове метода в качестве аргумента для параметра, помеченного params, можно ничего не передавать, тогда компилятор подставит массив нулевой длины;

- Нельзя создать метод, отличающийся только модификатором params у массива – из-за неоднозначности вызова возникнет ошибка компиляции.

Параметр с Модификатором params: Пример 1

```
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

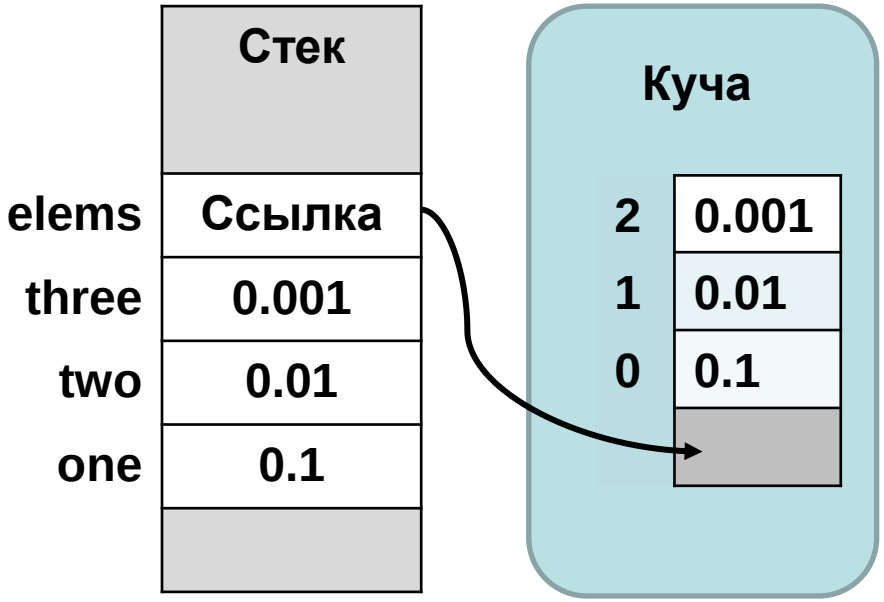
Параметр с Модификатором params: Пример 2.1

```
using System;

double one = 0.1, two = 0.01, three = 0.001;
Console.WriteLine($"\nInverse sum = {Inverse(one, two, three)}");
Console.WriteLine($"one = {one}; two = {two}; three = {three}");

static double Inverse(params double[] elems) {
    double res = 0;
    for (int k = 0; k < elems.Length; k++) {
        res += (elems[k] = 1 / elems[k]);
        Console.Write($"elems[{k}] = {elems[k]} ");
    }
    return res;
}
```

Результат выполнения:

elems[0] = 10 elems[1] = 100 elems[2] = 1000
Inversed sum = 1110
one = 0,1; two = 0,01; three = 0,001

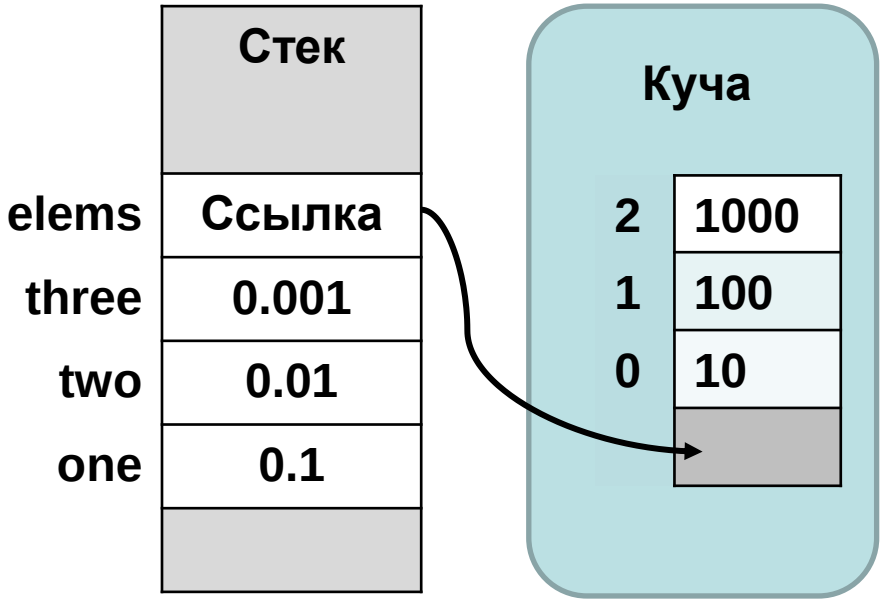
Параметр с Модификатором params: Состояние Памяти (Пример 2.1)



Начало тела метода

В данном примере в момент вызова метода с params средой выполнения создаётся массив elems, заполняемый указанными элементами.

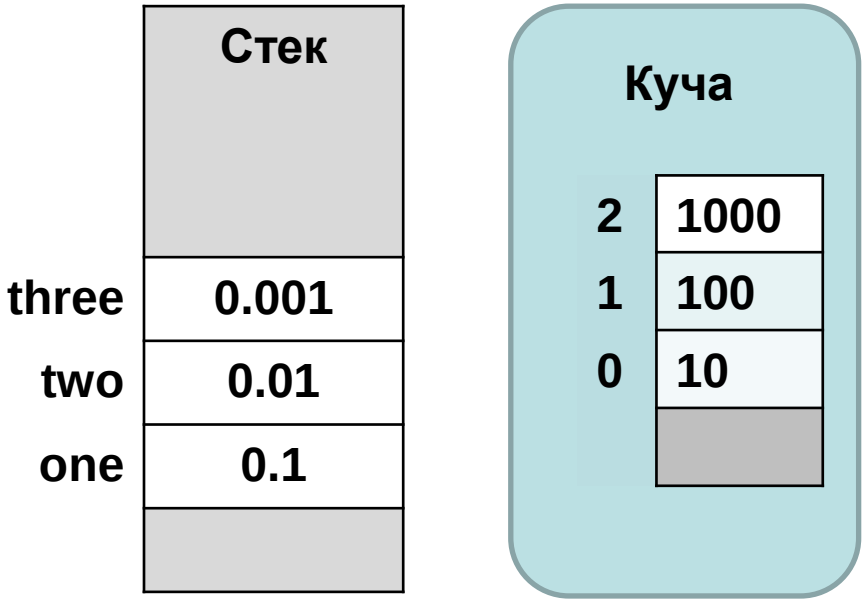
Параметр с модификатором params: Состояние Памяти (Пример 2.1)



Конец тела метода

Метод обрабатывает копии переменных one, two, three, оригинальные значения остаются нетронутыми.

Параметр с модификатором params: Состояние Памяти (Пример 2.1)



После выхода из метода

По окончанию выполнения метода ссылка на массив на стеке удаляется, а сам массив остаётся в куче для сборщика мусора.

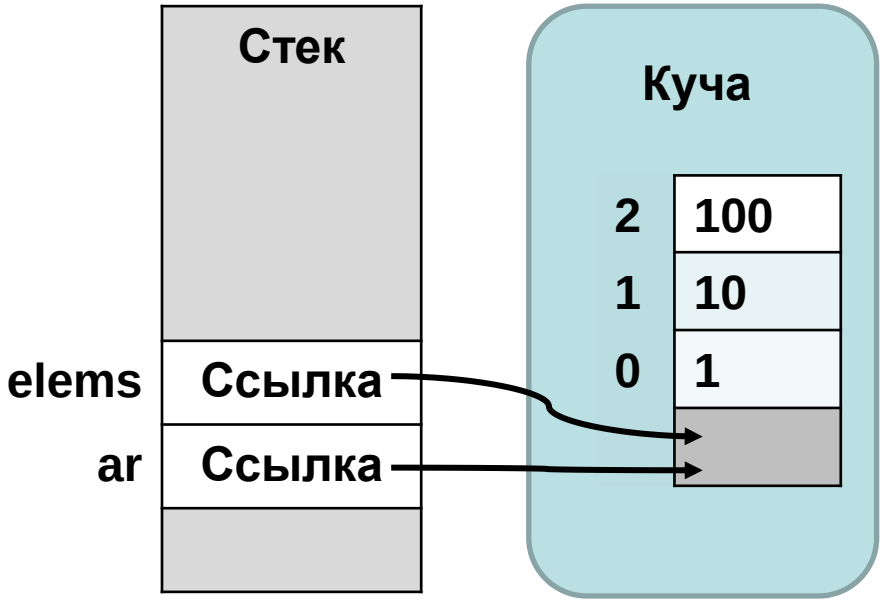
Параметр с Модификатором params: Пример 2.2

```
using System;

double[] array = { 1, 10, 100 };
Console.WriteLine(Inverse(array));
foreach (double value in array)
{
    Console.WriteLine(value);
}
```

Результат выполнения:

elems[0] = 1 elems[1] = 0,1 elems[2] = 0,01
1,11
1
0,1
0,01

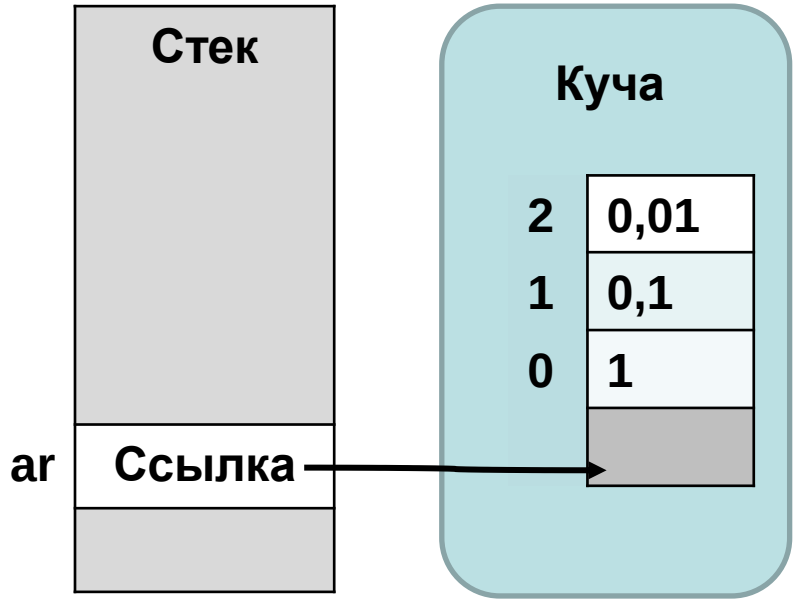
Параметр с Модификатором params: Состояние Памяти (Пример 2.2)



Начало тела метода

При явной передаче массива происходит абсолютно то же самое, что и в случае с передачей массива без модификатора params – скопируется ссылка.

Параметр с Модификатором params: Состояние Памяти (Пример 2.2)



После выхода из метода

При выходе из метода удаляется только копия ссылки, сам массив остаётся нетронутым.

Сравнение Модификаторов Параметров

Модификатор	Нужен при вызове?	Что может быть аргументом	Особенности
<пусто>	–	Всё кроме неинициализированных переменных.	Передача по значению.
ref	да	Поля и инициализированные локальные переменные.	Передача по ссылке, допустимы изменения.
out	да	Поля и любые локальные переменные.	Передача по ссылке, требуется инициализация.
in	не всегда, иногда недопустим	Всё кроме неинициализированных переменных.	Передача по ссылке (для инициализированных переменных и полей) и по значению для всего остального, изменения запрещены.
params	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.

Именованные Аргументы (Named Arguments)

При передаче аргументов в методы с явным указанием имени необходимо:

- Либо явно проименовать все аргументы в произвольном порядке;

- Либо только несколько, однако порядок передачи неименованных аргументов должен совпадать с порядком объявления параметров в заголовке метода. Аргументы, находящиеся на своих местах, без явного указания имени в таком случае называют **позиционными**.

https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments

Именованные Аргументы: Пример 1

```
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 именуются в корректном порядке.
Sum(a: 6, 4, c: 2);    // Хотя именованные параметры стоят не
                       // подряд, порядок сохраняется.
```

Важно: Код **Sum(b: 2, a: 1, 5);** не скомпилируется, т. к. не все аргументы именуются, а имена b и a перепутаны местами (error CS8323: Named argument 'b' is used out-of-position but is followed by an unnamed argument).

Именованные Аргументы: Пример 2

```
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

Пример позиционного и именованного аргумента в одном списке:

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

Пример ошибки – *именованный аргумент “x” задаёт параметр, для которого был уже установлен позиционный аргумент*:
Div(10, x: 3+8)

Предназначение Именованных Аргументов

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

    static void Main()
    {
        // Без именования сходу непонятно, что такое 3.0 и 4.0.
        Console.WriteLine($"Volume 1: {Volume(3.0, 4.0):F3}");

        // Для каждого аргумента явно прописывается имя: ясно, для чего он нужен.
        Console.WriteLine("Volume 2 (named): " +
                          $"{Volume(radius: 4.0, height: 5.0):F3}");
    }
}
```

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

Параметры со Значениями по Умолчанию (опциональные аргументы)

При объявлении метода можно указать параметрам **значения по умолчанию** через = в заголовке. При вызове метода соответствующие аргументы можно не передавать – вместо них будут подставлены значения по умолчанию.

Параметры со значениями по умолчанию *должны быть в конце списка параметров*, в противном случае код не скомпилируется.

Значениями по умолчания могут быть только следующие типы выражений:

- Константы;
- new <Type>( ) (где, Type – struct или enum) или new() *(C# 9.0)*;
- default(<Type>) или default.

Опциональные Аргументы: Пример 1

```
using System;

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

Опциональные Аргументы: Пример 2

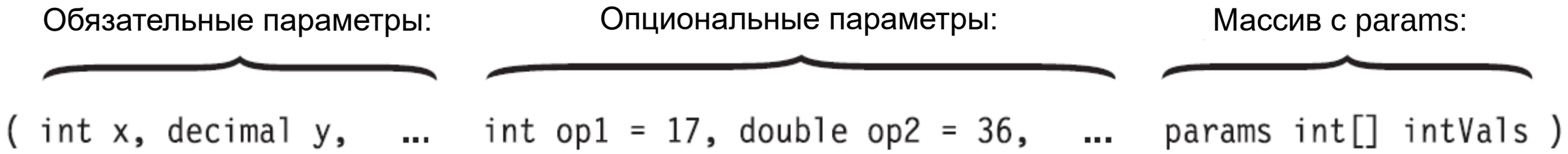
```
using System;
class OptionalParamsDemo
{
    static int Calc(int a = 2, int b = 3, int c = 4) => (a + b) * c;
    static void Main()
    {
        int r0 = Calc(5, 6, 7); // Все аргументы указаны явно.
        int r1 = Calc(5, 6); // Аргумент по умолчанию для c.
        int r2 = Calc(5); // Аргументы 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

Порядок Указания Параметров в Заголовке Метода

В случае комбинирования различных видов параметров необходимо использовать определённый порядок:

1. Обязательные параметры;
2. Опциональные параметры;
3. Параметр-массив с модификатором params.



Примеры: Позиционные и Именованные Аргументы, Опциональные Параметры

```
using System;

static double Volume(double radius = 3.0, double height = 4.0)
        => Math.PI * radius * radius * height;

double volume = Volume(3.0, 4.0);
volume = Volume(radius: 2.0);          // height по умолчанию.
volume = Volume(1.0, height: 2.0);     // Позиционные аргументы.
volume = Volume(height: 2.0);          // radius по умолчанию.
volume = Volume();                     // Оба аргумента - по умолчанию.
```

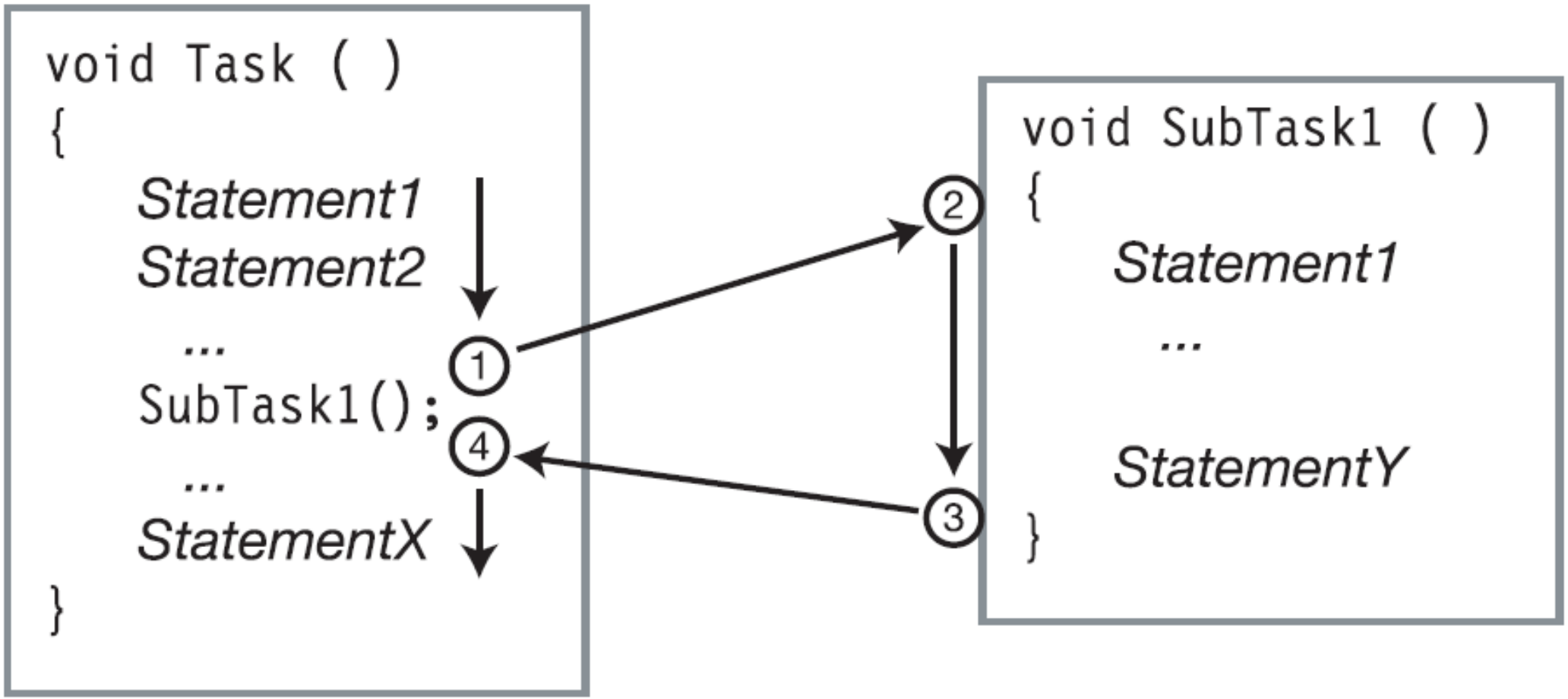
Заголовки с Опциональными Параметрами

```
// 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) { }
```

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)) { }
```

Поток Управления при Вызове Метода



О Стековых Фреймах (Часть 1 Примера)

```
using System;
public class StackFramesDemo {
    public static void MethodA(int par1, int par2) {
        Console.WriteLine($"Entered MethodA: {par1}, {par2}.");
        MethodB(11, 18);                    // Вызов MethodB.
        Console.WriteLine("Left MethodA.");
    }
    static void MethodB(int par1, int par2) {
        Console.WriteLine($"Entered MethodB: {par1}, {par2}.");
        Console.WriteLine("Left MethodB.");
    }
}
```

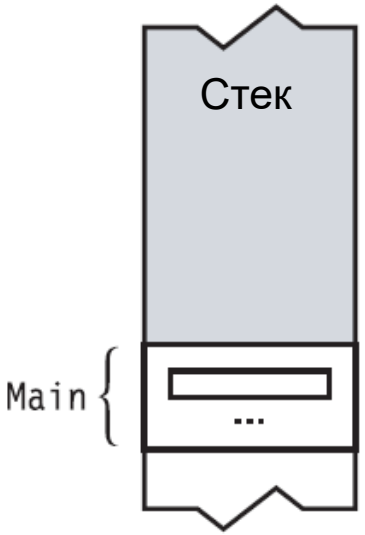
О Стековых Фреймах (Часть 2 Примера)

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Entered Main.");
        StackFramesDemo.MethodA(15, 30);    // Вызов MethodA.
        Console.WriteLine("Left Main.");
    }
}
```

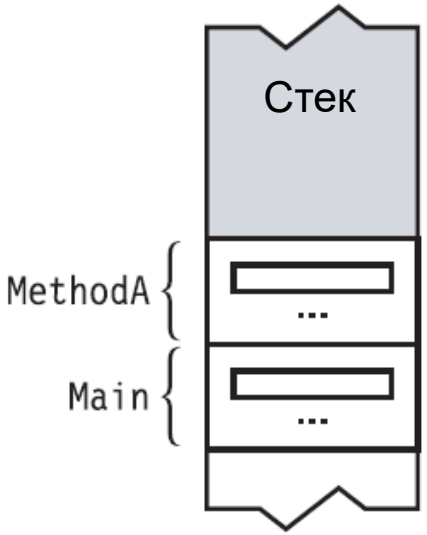
Результат выполнения:

```
Entered Main.
Entered MethodA: 15, 30.
Entered MethodB: 11, 18.
Left MethodB.
Left MethodA.
Left Main.
```

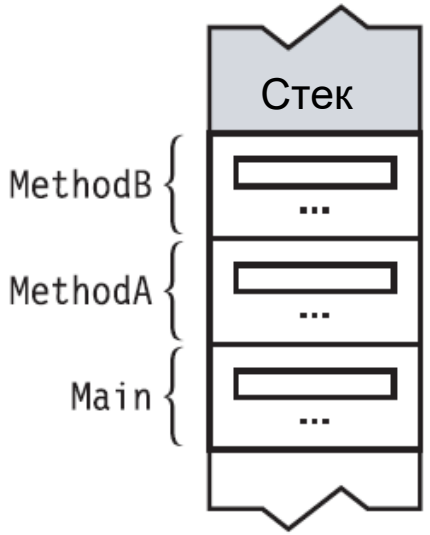
Визуализация Стековых Фреймов Примера



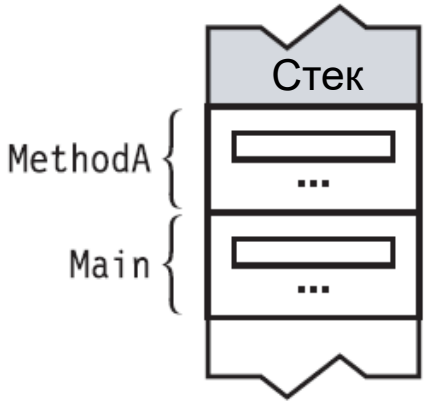
Выполнение Main начинается.



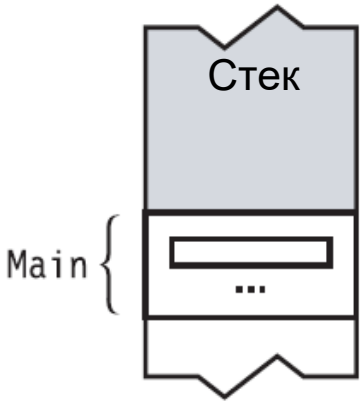
Вызов MethodA из Main.



Вызов MethodB из MethodA.



Удаление MethodB со стека.



Удаление MethodA со стека.

Рекурсия

```
class MyMath
{
    static long Factorial(long value)
    {
        if (value <= 1)
            return 1;
        else
            return value * Factorial(value - 1);
    }
}
```

Пример Рекурсии

```
using System;
class Program
{
    public void Count(int inVal)
    {
        Console.WriteLine($"Вход: {inVal}");
        if (inVal == 0) return;
        Count(inVal - 1);   // Рекурсивный вызов
        Console.WriteLine($"Выход: {inVal}");
    }
    static void Main()
    {
        Program pr = new Program();
        pr.Count(3);
    }   // Main()
}   // Program
```

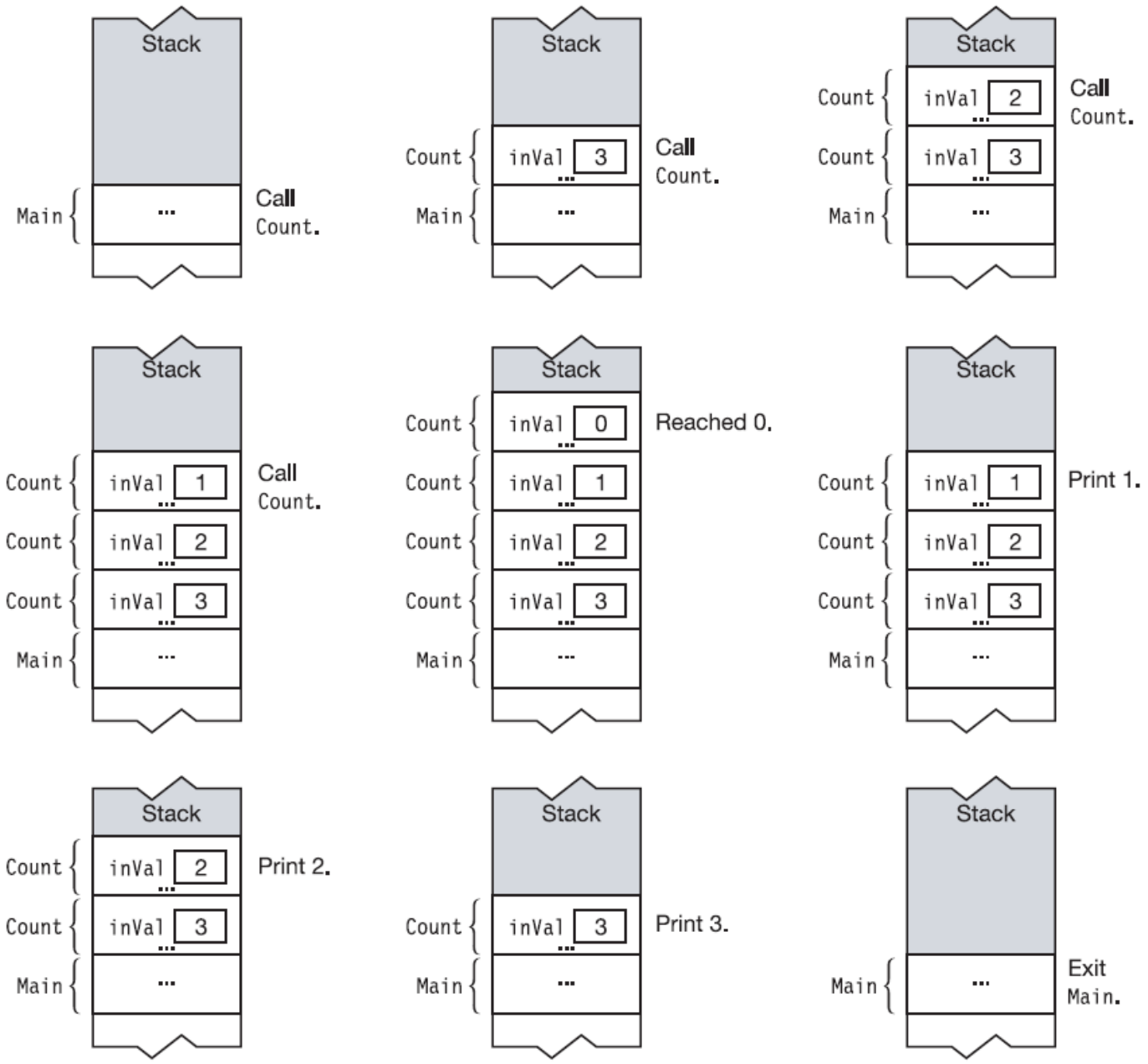
Результат выполнения:

Вход: 3
Вход: 2
Вход: 1
Вход: 0
Выход: 1
Выход: 2
Выход: 3

Рекурсия и Стек

При рекурсивных вызовах в C# стек заполняется аналогичным образом, как и при всех остальных.

По этой причине глубокая рекурсия может приводить к StackOverflowException.



Традиционная и Хвостовая Рекурсия

```
using System.Numerics;
static ulong Fact(uint k) { // Традиционная.
    if (k < 2) return 1;
    return k * Fact(k - 1);
}

static BigInteger FactTail(uint k, BigInteger product) { // Хвостовая.
    if (k < 2) return product;
    return FactTail(k - 1, k * product);
}
static BigInteger FactTailWrapper(uint k) {
    return FactTail(k, 1);
}
```

Традиционная и Хвостовая Рекурсия

```
static void Main()
{
    Console.WriteLine(Fact(10));
    Console.WriteLine(Fact(40));
    Console.WriteLine(FactTailWrapper(10));
    Console.WriteLine(FactTailWrapper(40));
}
```

Результат выполнения:

```
3628800
18376134811363311616
3628800
18376134811363311616
```

Обратите внимание: хвостовая рекурсия поддерживается для IL, но не компилятором C#.
Подробнее про хвостовую рекурсию в Intermediate language:
https://thomaslevesque.com/2011/09/02/tail-recursion-in-c/

Локальные методы (local methods)

```
static void DemoLocalMethods()
{
    int k = 5;
    Console.WriteLine(GetProductWithCoefficient(3));
    Console.WriteLine(GetExponent(3));

    int GetProductWithCoefficient(int value) {
        return k * value;
    }

    static int GetExponent(int value) {
        // Переменная k не видна.
        return value * value;
    }
}
```

Результат выполнения:

15
9