Информационные технологии и программирование

Лекция 4. Основы языка C#

Содержание лекции:

- Методы
- Область видимости (Scope)
- Main
- Стек вызовов (Call Stack)
- Копирование значений при передаче в методы
- Модификаторы параметров — ref и out
- Кортежи
- Перегрузка методов
- Необязательные параметры
- Генерация псевдослучайных чисел

Преподаватель курса: Клюкин Даниил Анатольевич, ст. преподаватель каф. ПМиИТ

Дублирование кода – ПЛОХО.

Если в дублированном коде будут ошибки, то искать и исправлять их придется везде, куда скопирован код.

```
var average1 = 0.0;
for (int i = 0; i <= numbers1.Length; i++)
    average += numbers1[i];

average1 /= numbers1.Length;

// Спустя 500 строк скопировали этот же код
var average1 = 0.0;
for (int i = 0; i <= numbers2.Length; i++)
    average += numbers[i];

average2 /= numbers2.Length;
```

Методы

Метод – блок кода, содержащий ряд инструкций.

```
[тип возвращаемого значения] [Имя]([аргументы])
{
   float a = 10.2f;
   float b = 3.4f;

   return a+b;
}
```

```
float getSum(float a, float b)
{
   return a+b;
}
```

При передаче аргументов в метод они **копируются.** Если это **тип значения**, то копируются сами значения, если тип **ссылочный**, то копируется **ссылка**.

Методы также удобны тем, что скрывают сложную логику внутри себя.

Бывают:

"Метод-действие" (сохранить, напечатать)

и "Метод-функция" (посчитать, найти).

void – специальное слово, означающее, что метод **не возвращает значения**.

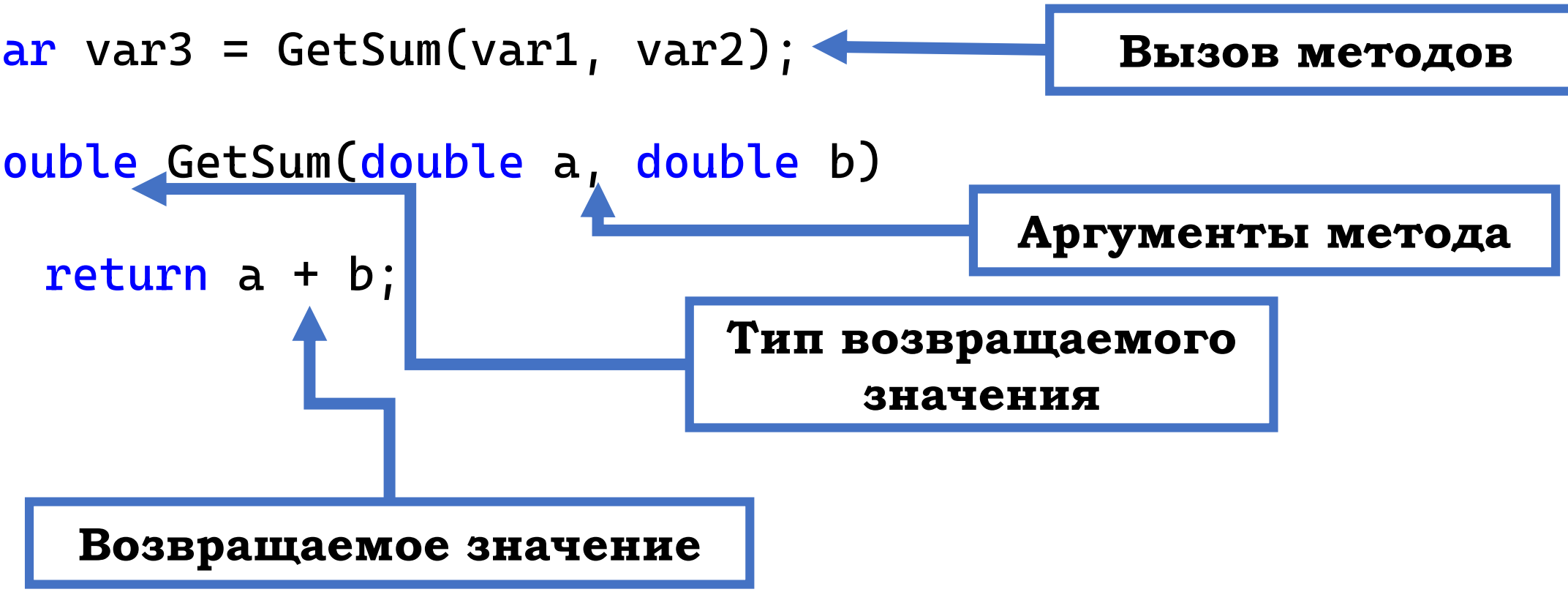
```
var var1 = Convert.ToDouble(Console.ReadLine());
var var2 = Convert.ToDouble(Console.ReadLine());

WriteSum(var1, var2);

void WriteSum(double a, double b)
{
   Console.WriteLine(a + b);
}
```

Вызов метода

Аргументы метода

return – специальное слово, **завершающее** выполнение метода и **возвращающее** результат (при его наличии).

```
var var1 = Convert.ToDouble(Console.ReadLine());
var var2 = Convert.ToDouble(Console.ReadLine());

var var3 = GetSum(var1, var2);

double GetSum(double a, double b)
{
   return a + b;
}
```



Сокращенный способ записи методов с помощью оператора => Данный способ возможен только для однострочных методов.

```
var var1 = Convert.ToDouble(Console.ReadLine());
var var2 = Convert.ToDouble(Console.ReadLine());

WriteSum(var1, var2);

var var3 = GetSum(var1, var2);

void WriteSum(double a, double b) =>
    Console.WriteLine(a + b);


double GetSum(double a, double b) => a + b;
```

Пример. Предположим, что нам необходимо найти среднее значение по массиву данных:

```
double GetAverage(double[] numbers)
{
    var average = 0.0;

    for (int i = 0; i < numbers.Length; i++)
        average += numbers[i];

    average /= numbers.Length;
    return average;
}

var numbers = new double[] { 1, 2, 3, 4, 5, 6 };

Console.WriteLine(GetAverage(numbers));
```

Область видимости (Scope)

"Видим" ли мы переменную i или average вне метода GetAverage?

```
double GetAverage(double[] numbers)
{
    var sum = 0.0;
    for (int i = 0; i < numbers.Length; i++)
        sum += numbers[i];
    return sum / numbers.Length;
}
var numbers = new double[] { 1, 2, 3, 4, 5, 6 };
GetAverage(numbers);
```

Переменные, созданные внутри метода (локальные переменные), живут только пока выполняется этот метод.

Как только мы выходим из фигурной скобки }, эти переменные уничтожаются.

Именно поэтому мы используем return, чтобы "выбросить" готовый результат наружу, прежде чем метод закроется.

Main

На самом деле, когда мы создаем консольное приложение, в его основе всегда есть **метод Main** — это **точка входа** в программу. Когда вы запускаете программу, операционная система ищет именно этот метод и начинает выполнение кода с него.

Типичная структура выглядит так:

```
namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
        }
    }
}
```

```
namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
        }
    }
}
```

ТОЖЕ САМОЕ ЧТО И

```
Console.WriteLine("Hello world!");
```

Стек вызовов (Call Stack)

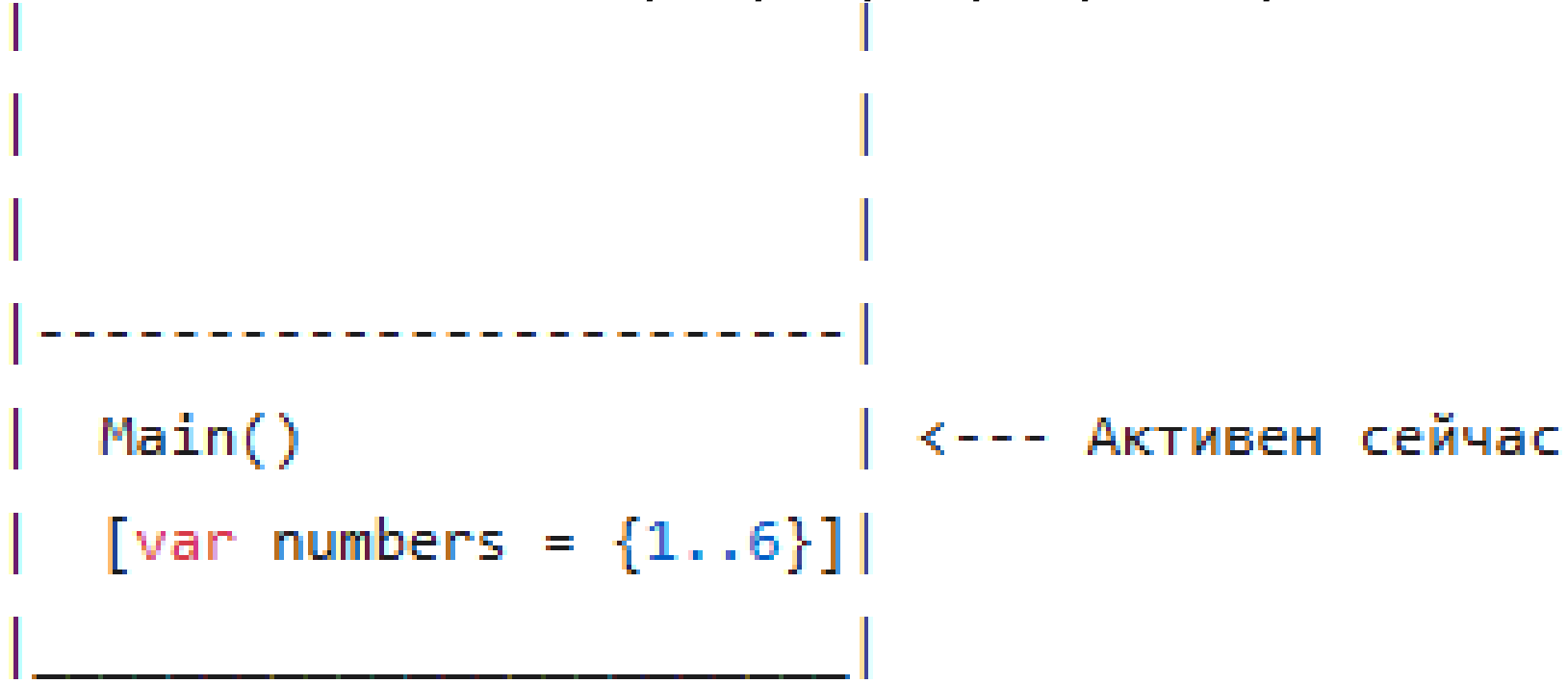
Компьютер управляет памятью очень организованно. Для выполнения методов используется структура данных, называемая **Стек (Stack)**. Представьте его как стопку тарелок или книг.

Вы можете положить новую тарелку только наверх (**Push**).

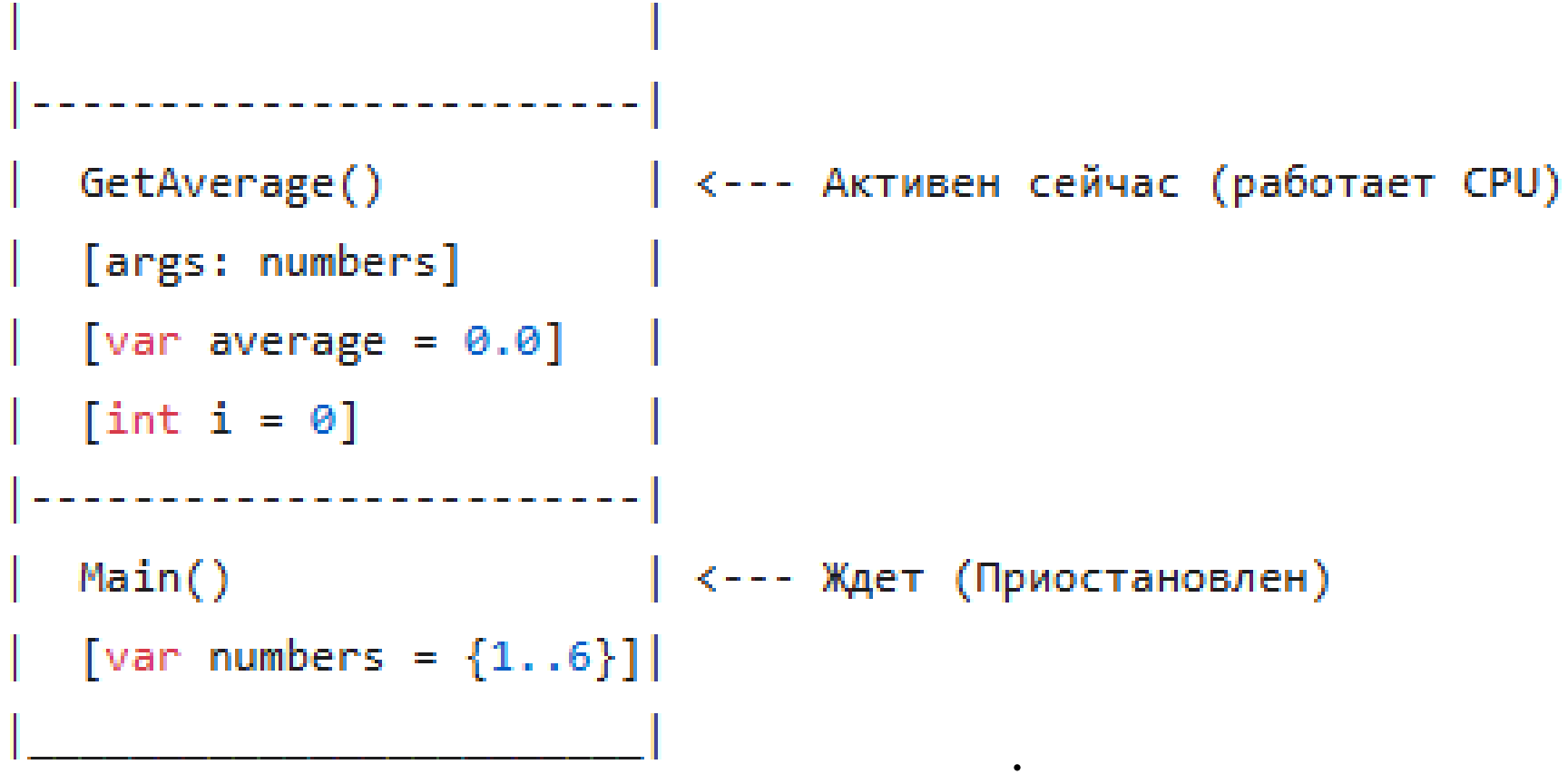
Вы можете взять тарелку только сверху (**Pop**).

LIFO (Last In, First Out) — Последним пришел, первым ушел.

```
var numbers = new double[] { 1, 2, 3, 4, 5, 6 };
```



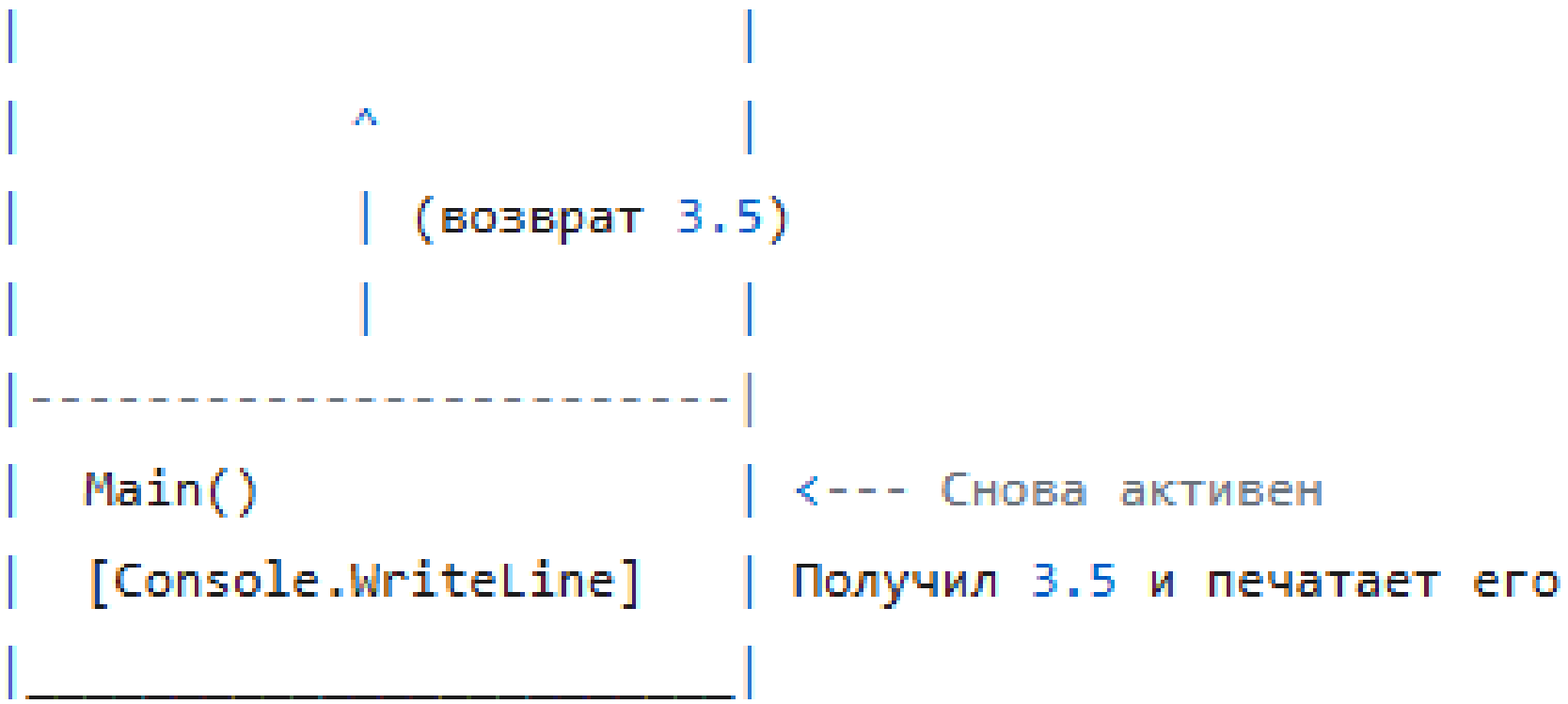
Метод Main доходит до строчки GetAverage(numbers). Main **приостанавливается**. Он не может продолжить работу, пока не получит результат. Поверх него в стек кладется новый кадр — GetAverage.



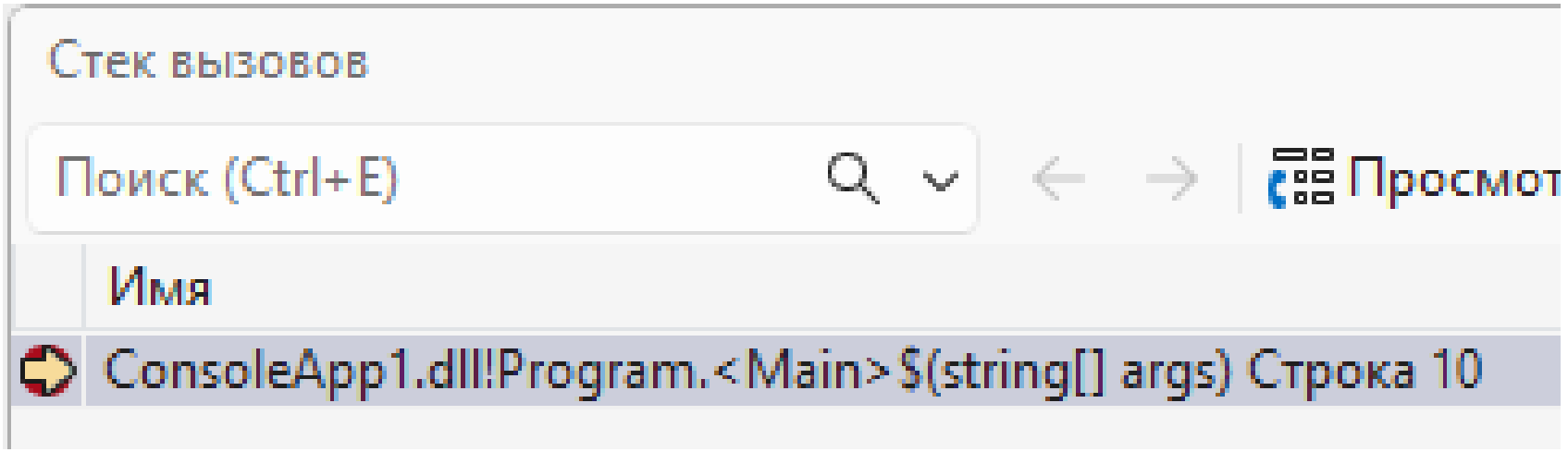
Обратите внимание: переменные average и i существуют только в верхнем блоке.

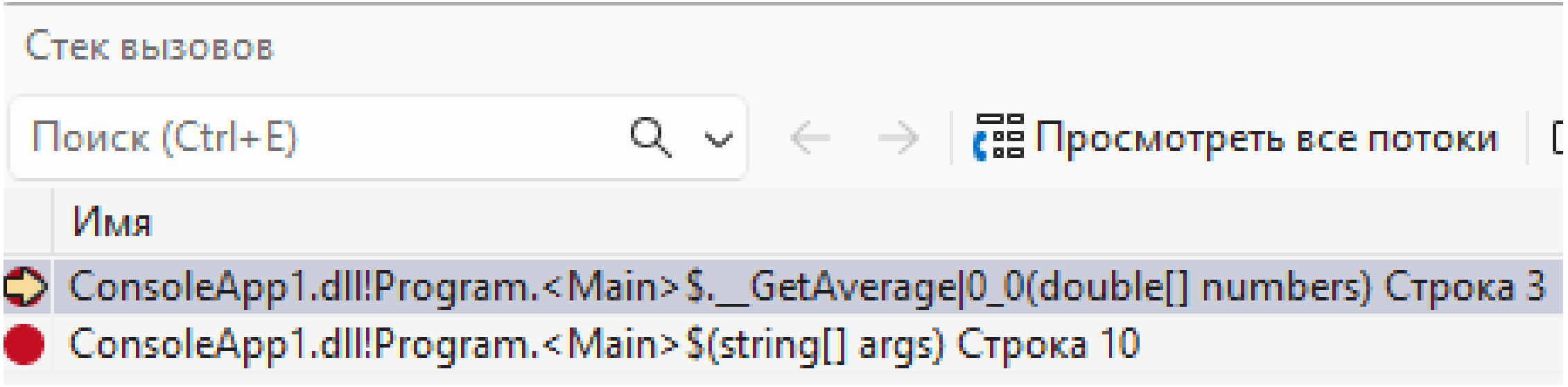
Метод GetAverage выполнил вычисления, дошел до return average (вернул 3.5).

Его кадр **уничтожается** (выбрасывается из стека). Память под i и локальный average очищается. Управление возвращается в Main.



Стек вызовов отображается на одноименной вкладке в VS.





Рекурсия – вызов метода из самого метода.

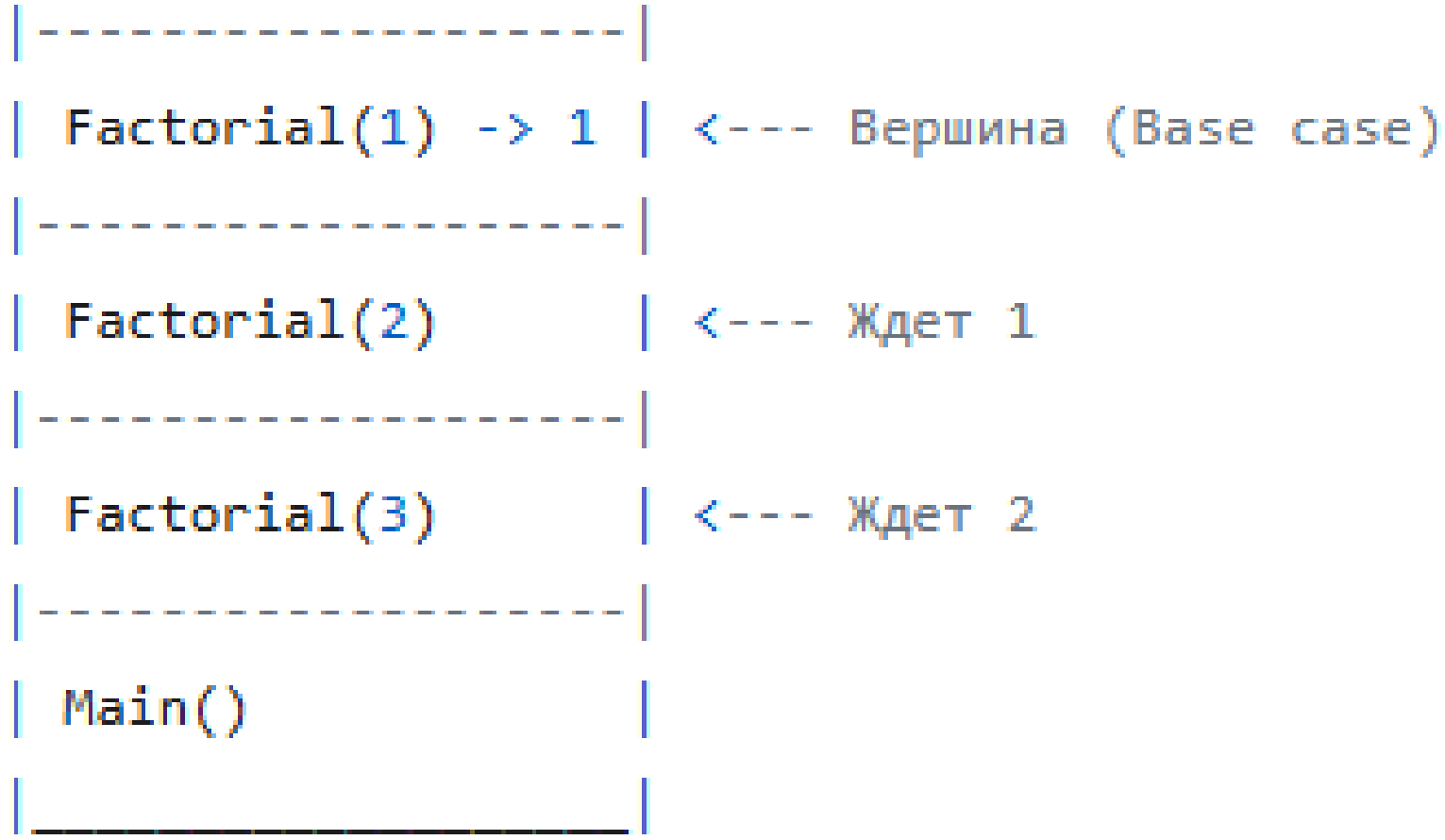
```
Console.WriteLine(Factorial(5));

double Factorial(double d)
{
   if (d <= 1)
         return 1;
   return d * Factorial(d - 1);
}
```

```
120
```

Факториал натурального числа n определяется как произведение всех натуральных чисел от 1 до n включительно.

Замечание: Любой рекурсивный алгоритм можно переделать в не рекурсивный, например, с помощью бесконечного цикла или специальных коллекций данных.

Представьте, что вы хотите узнать факториал 3.
Main: Вызывает F(3).
Стек: F(3) говорит: 'Я не знаю результат, мне нужно узнать 3 * F(2)'.
Стек: F(2) говорит: 'Я не знаю результат, мне нужно 2 * F(1)'.
Стек: F(1) говорит: 'О, я знаю! Это 1'. (Условие d <= 1 сработало).
Возврат: Стек начинает схлопываться обратно, перемножая результаты."



А теперь представьте, что мы убрали if (d <= 1).

Метод будет вызывать себя вечно: 1, 0, -1, -2...

Стек будет расти бесконечно вверх. Но память компьютера конечна. Как только место закончится, операционная система убьет вашу программу.

Эта ошибка называется **StackOverflowException**.

Кстати, именно в честь этой ошибки назван самый популярный сайт с ответами для программистов — Stack Overflow.

Копирование значений при передаче в методы

При передаче переменных в метод происходит копирование данных.

В случае со значимыми типами копируется значение.

При передаче ссылочных типов – копируется адрес в памяти.

```
var a1 = 5;
var arr1 = new int[5];

Console.Write(a1);
Console.Write(arr1[0]);

a1 = Method1(a1);
Method2(arr1);

Console.Write(a1);
Console.Write(arr1[0]);
```

```
50611
```

```
int Method1(int a1)
{
    a1 = 10;
    return 6;
}

void Method2(int[] arr)
{
    arr[0] = 11;
}
```

КОПИЯ!

Вопросы

Дан метод:

```
string WritePersonInfo(int age, string name)
{
   return $“{name} {age} лет”;
}
```

1. Как называется метод?
2. Какой тип данных он возвращает?
3. Сколько аргументов у метода?
4. Какой тип у аргументов?

Дан метод:

1. Как называется метод?
2. Какой тип данных он возвращает?
3. Сколько аргументов у метода?
4. Какой тип у аргументов?
5. Что делает метод?

```
bool IsPrime(int number)
{
        if (number == 2)
            return true;

        if (number % 2 == 0 || number < 2)
            return false;

        var sqrt = (int)Math.Sqrt(number);

        for (var i = 3; i <= sqrt; i += 2)
            if (number % i == 0)
                return false;

        return true;
}
```

Модификаторы параметров — ref и out

`ref` – используется для указания того, что переданный параметр **может быть изменен** методом.

`in` – используется для указания того, что переданный параметр **не может быть изменен** методом (только в версии >= C# 7.2).

`out` – используется для указания того, что переданный параметр **должен быть изменен** методом.

```
void Swap(ref int a, ref int b)
{
    var temp = a;
    a = b;
    b = temp;
}
int x = 1, y = 2;
Swap(ref x, ref y); // Передаем ссылки на x и y
// Теперь x = 2, y = 1.
```

Используем, когда метод должен вернуть больше одного результата. *Вспоминаем Лекцию 2 и int.TryParse.* Напишем свой метод деления, который возвращает результат и (отдельно) остаток.

```
// int result – это обычный return
// out int remainder – это "бонусное" возвращаемое значение
int Divide(int a, int b, out int remainder)
{
    remainder = a % b; // Обязаны записать значение в out!
    return a / b;
}

int rem; // Можно не инициализировать заранее
int res = Divide(10, 3, out rem);

Console.WriteLine($"Результат: {res}, Остаток: {rem}");
// Вывод: 3, 1
```

Кортежи

Проблема: Метод в C# может возвращать только **одно** значение (через return). Модификатор out позволяет обойти это, но он делает код громоздким.

Решение: Кортежи позволяют «склеить» несколько переменных в один объект и вернуть его целиком.

```
(string, int) tuple = ("Вася", 175);
```

Кортеж (tuple) в C# — это структура, позволяющая сгруппировать несколько разнородных значений в один объект.

```
// 1. Способ через out
int DivideWithOut(int a, int b, out int remainder)
{
    remainder = a % b;
    return a / b;
}

// 2. Способ через КОРТЕЖ (Tuple) — чисто и красиво
(int quotient, int remainder) Divide(int a, int b)
{
    return (a / b, a % b); // Возвращаем пару значений сразу
}

// Вызов метода:
var result = Divide(10, 3);
Console.WriteLine($"Частное: {result.quotient}, Остаток:
{result.remainder}");
```

Вам не обязательно создавать переменную result и обращаться к её полям через точку. Вы можете «распаковать» значения сразу в две независимые переменные.

```
// Без распаковки
var result = Divide(10, 3);
// Распаковываем результат метода прямо "на лету"
var (q, r) = Divide(10, 3);

Console.WriteLine($"Частное: {q}"); // 3
Console.WriteLine($"Остаток: {r}");  // 1
```

`params` – ключевое слово, означающее, что метод принимает переменное число аргументов (одного типа). Используется для удобства написания кода.

```
Method1(1, 2, 3, 4, 5);

void Method1(params int[] arguments)
{
    foreach (var argument in arguments)
    {
        Console.Write($"{argument} ");
    }
}
```

```
1 2 3 4 5
```

Пример на тему создания методов. Создание массива со случайными числами и вывод его на консоль.

```
// Метод для создания и заполнения двумерного массива случайными числами
int[,] CreateRandomArray(int rows, int columns)
{
    Random random = new Random();
    int[,] array = new int[rows, columns];
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            array[i, j] = random.Next(1, 101);
// Заполняем случайными числами от 1 до 100
        }
    }
    return array;
}
```

```
// Метод для красивого вывода двумерного массива на экран
void PrintArray(int[,] array)
{
    int rows = array.GetLength(0);
    int columns = array.GetLength(1);

    Console.WriteLine("Двумерный массив:");
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            Console.Write($"{array[i, j],4} ");
// Выводим элемент с выравниванием
        }
        Console.WriteLine();
// Переход на новую строку после каждой строки массива
    }
}
```

Использование созданных методов.

```
int rows = 5; // Количество строк
int columns = 5; // Количество столбцов

// Создаем и заполняем массив
int[,] randomArray = CreateRandomArray(rows, columns);

// Выводим массив на экран
PrintArray(randomArray);
```

```
Двумерный массив:
  95   92    1   37    4
  52   74   56   12   76
  52   70   74   13   93
  17    3   16   46   32
  83   15   19   35   78
```

Перегрузка методов

Мы можем создавать методы с одним и тем же именем, если у них разные наборы параметров.

Компилятор сам выбирает нужный метод, ориентируясь на типы передаваемых данных.

```
static void Main(string[] args)
{
        Print(42);                          // Вызовет версию для int
        Print("Привет, мир!");              // версия для string
        Print(new double[] { 1.5, 2.5 }); // версия для массива
}
// 1. Версия для целых чисел
static void Print(int number)
// 2. Версия для строк
static void Print(string text)
// 3. Версия для массивов
static void Print(double[] array)
```

Представьте, если бы не было перегрузки. Вам пришлось бы помнить названия функций для каждого типа данных:
PrintInt(10), PrintString("Text"), PrintArray(...).
Это неудобно. Перегрузка позволяет нам использовать **одно логическое действие** (напечатать, открыть, сохранить) для **разных типов данных**. Вы просто пишете Print, а C# сам разбирается, что делать.

Вы уже пользовались перегрузкой с первого занятия. Напишите в Visual Studio **Console.WriteLine**(и посмотрите подсказку. Там написано (+ 18 overloads).
Это значит, что разработчики Microsoft написали **19 версий этого метода**: для int, string, bool, char и так далее. Вы не задумываетесь об этом, просто пользуетесь.

Необязательные параметры

Позволяют вызывать метод, не передавая аргументы для некоторых параметров. Вместо отсутствующих аргументов компилятор подставит значение по умолчанию.

```
// 1. Вызываем только с обязательным параметром
// В переменную 'prefix' автоматически подставится "INFO"
PrintMessage("Запуск системы...");
// Вывод: [INFO] Запуск системы...
// 2. Явно передаем второй аргумент
// Значение по умолчанию игнорируется
PrintMessage("Критическая ошибка!", "ERROR");
// Вывод: [ERROR] Критическая ошибка!

void PrintMessage(string message, string prefix = "INFO")
{
    Console.WriteLine($"[{prefix}] {message}");
}
```

Компилятор читает аргументы слева направо.

Поэтому **обязательные** параметры (те, у которых нет = value) всегда должны идти первыми.

Нельзя написать void Method(int a = 1, int b), потому что при вызове Method(5) компьютер не поймет: 5 — это a или b?

Значение по умолчанию должно быть известно **во время компиляции** (константой).

Вы **не можете** написать:

void Log(string msg, DateTime date = DateTime.Now) ✕

Потому что DateTime.Now вычисляется во время выполнения программы.

Генерация псевдослучайных чисел

Когда требуется произвольный массив данных, довольно неудобно его вводить вручную, тем более если необходимо проверить программу на различных примерах. Здесь нам поможет генерация псевдослучайных чисел:

```
var rnd = new Random(); // Генератор псевдослучайных чисел

int min = 0, max = 100;
// Генерация числа
var number_1 = rnd.Next(); // от 0 до 2147483647
var number_2 = rnd.Next(max); // строго от 0 до max-1
var number_3 = rnd.Next(min, max); // строго от min строго до max-1
var number_4 = rnd.NextDouble(); // от 0 строго до 1 не включительно
```

Заполним массив:

```
var rnd = new Random(); // Генератор псевдослучайных чисел

double[] array = new double[10];
int min = 0, max = 100;

for (int i = 0; i < array.Length; i++)
{
    // Генерация целого числа от min до max (max не включается)
    array[i] = rnd.Next(min, max);
}
for (int i = 0; i < array.Length; i++)
{
    Console.Write($"{array[i]}\t");
}
```

```
57      59      83      24      30
```