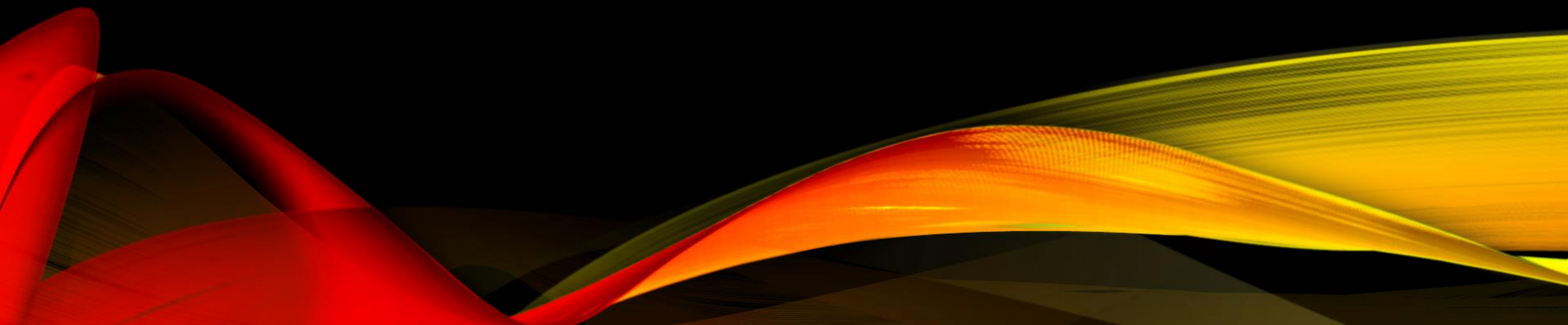
ЛЕКЦИЯ 3

- Модуль 3
- 17.01.2023
- Анонимные методы и Лямбда-функции

ЦЕЛИ ЛЕКЦИИ

- Изучить способы реализации анонимных методов и лямбда-выражений в C#

АНОНИМНЫЕ МЕТОДЫ



АНОНИМНЫЕ ФУНКЦИИ

Варианты синтаксиса анонимных функций:

- Анонимные методы
- Лямбда-выражения
  - альтернатива анонимным методам

ОБЪЯВЛЕНИЕ АНОНИМНЫХ МЕТОДОВ

Определяем анонимный метод с использованием ключевого слова **delegate**

Параметры анонимного метода (могут отсутствовать)

```
System.Func<int, int, int> sumFunc = delegate (int x, int y)
{
    return x + y;
};
```

Тело метода

У этого метода нет имени, только ссылка с типом делегата `System.Func<int,int,int>`

АНОНИМНЫЕ МЕТОДЫ VS. МЕТОДЫ КЛАССА

Именованный метод

```
class Program
{
    public static int Add20(int x)
    {
        return x + 20;
    }
    delegate int IntOperationDelegate(int x);
    static void Main(string[] args)
    {
        IntOperationDelegate del = Add20;
        Console.WriteLine($"{del(5)} {del(6)}");
    }
}
```

Анонимный метод

```
class Program
{
    delegate int IntOperationDelegate(int x);
    static void Main(string[] args)
    {
        IntOperationDelegate del = delegate (int x)
        {
            return x + 20;
        };

        Console.WriteLine($"{del(5)} {del(6)}");
    }
}
```

ПРОПУСК СПИСКА ПАРАМЕТРОВ

```
Action noParams = delegate {
            Console.WriteLine("111");
          };
noParams();

Action<int, double> twoParams = delegate {
                    Console.WriteLine("222");
                  };
twoParams(42, 3.14);
```

Нет параметров

Нет аргументов при вызове

Нет параметров

Лямбда-выражения так не могут!!!

Два аргумента при вызове

ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ И ПАРАМЕТРОВ

- Локальные переменные и параметры анонимных методов существуют только внутри охватывающих фигурных скобок

```
delegate void IntOperationDelegate(int x);
    static void Main(string[] args)
    {
        IntOperationDelegate del = delegate (int x)
        {
            int z = 10;
            Console.WriteLine($"{x}, {z}");
        };

        Console.WriteLine($"{x}, {z}");
    }
```

Область видимости x, z

Ошибка компиляции, x, z не существуют в данном контексте

ВНЕШНИЕ ПЕРЕМЕННЫЕ И АНОНИМНЫЕ МЕТОДЫ

- Для нестатических анонимных методов внешние переменные захватываются автоматически и доступны без дополнительных ограничений

```
delegate void IntOperationDelegate(int x);
static void Main(string[] args)
{
    int y = 15;
    IntOperationDelegate del = delegate (int x)
    {
        int z = y + 10;
        Console.WriteLine($"{x}, {z}");
    };
}
```

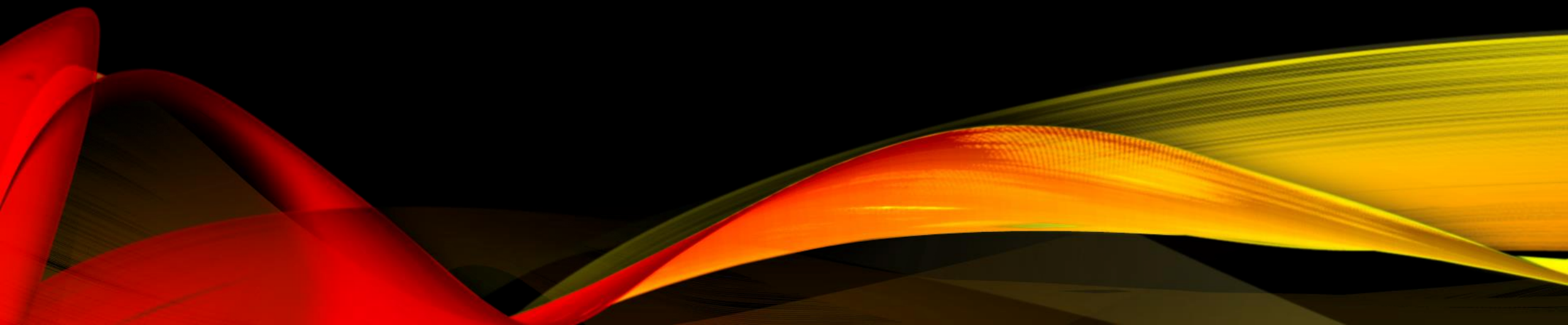
Переменная y определена во внешнем по отношению к анонимному методу контексте

Обращение к y внутри анонимного метода допустимо

ПРОДЛЕНИЕ ВРЕМЕНИ ЖИЗНИ ЛОКАЛЬНЫХ ПЕРЕМЕННЫХ

```
delegate void IntOperationDelegate();
    static void Main(string[] args)
    {
        IntOperationDelegate del;
        { // начало области видимости переменной y
            int y = 15;
            del = delegate
            {
                // происходит "захват" переменной y анонимным методом
                Console.WriteLine($"Value of y: {y}");
            };
        } // конец области видимости переменной y
        // Console.WriteLine($"Value of y: {y}"); // Error!!!
                                                  // переменная не существует в данном контексте
        del?.Invoke(); // это работает, т.к. значение переменной "захвачено"
    }
```

ЛЯМБДА-ВЫРАЖЕНИЯ



ЛЯМБДА-ВЫРАЖЕНИЯ

Лямбда-выражение является альтернативой анонимному методу и используется для создания анонимной функции

Различия анонимных методов и лямбда-выражений:

- Анонимные методы позволяют полностью опустить список параметров
- Лямбда-выражения позволяют опускать и выводить типы параметров
- Тело лямбда-выражения может быть выражением или блоком, тело анонимного метода – только блоком
- Только лямбда-выражения имеют преобразования к совместимым типам дерева выражений

СИНТАКСИС ЛЯМБДА-ВЫРАЖЕНИЙ

- С телом, представленным единственным выражением:
  (входные_параметры) => выражение;

```
Func<int, int> lambda4 = x => x + 1;
Action lambda5 = () => Console.WriteLine("lambda"); // Без параметров.
// С неявным указанием параметров:
Action<int, int> lambda6 = (x, y) => Console.WriteLine(x * y);
```

СИНТАКСИС ЛЯМБДА-ВЫРАЖЕНИЙ

- С телом, представленным блоком операторов:
  `(входные_параметры) => { <операторы> };`

```
Func<int, int> lambda1 = (int x) => { return x + 1; };

Func<int, int> lambda2 = (x) => { return x + 1; };  // Без явного типа параметра.

Func<int, int> lambda3 = x => { return x + 1; };    // Без скобок при параметре.
```

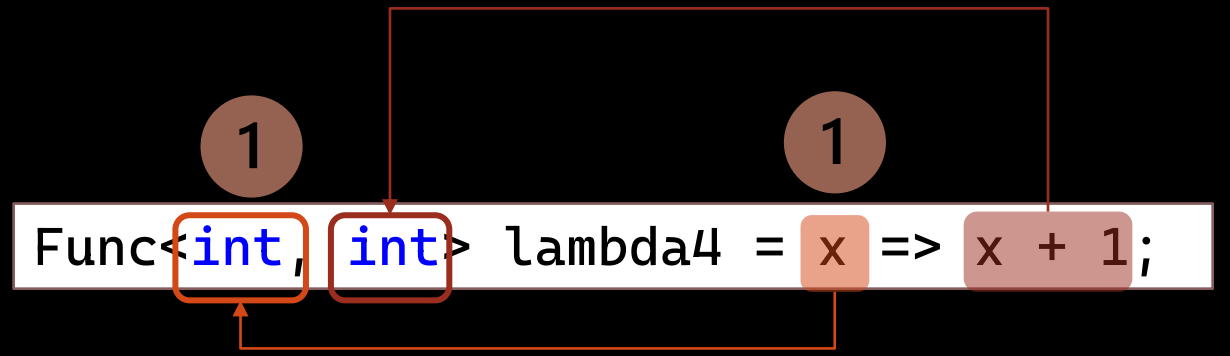
ПРЕОБРАЗОВАНИЕ ЛЯМБДЫ К ДЕЛЕГАТУ-ТИПУ

```
Action<int, int> lambda6 = (x, y) => Console.WriteLine(x * y);
```

Преобразование происходит при присваивании к переменной делегата

ТИПЫ ПАРАМЕТРОВ В ЛЯМБДА-ВЫРАЖЕНИЯХ

Возвращаемое значение (при наличии) должно неявно преобразовываться к типу возвращаемого значения, определённого делегатом-типом



Тип входного параметра должен быть неявно преобразуем к типу, определённому для этого параметра в делегатом-типом

ПРАВИЛА ОБЪЯВЛЕНИЯ ЛЯМБДА-ВЫРАЖЕНИЙ (1)

Между списком параметров и телом лямбда-оператор => указывается всегда

Типы всех параметров указываются либо полностью явно, либо полностью неявно:

```
Func<string, int, string> lambda = (string li, int num) => li + num;
Func<string, int, string> lambda = (line, num) => line + num;
```

Если у лямбда-выражения только один входной параметр, круглые скобки можно опустить:

```
Action<int> lambda = num => Console.WriteLine(x + 1);
```

ПРАВИЛА ОБЪЯВЛЕНИЯ ЛЯМБДА-ВЫРАЖЕНИЙ (2)

Для лямбда-выражений без параметров круглые скобки обязательны:

```
Action lambda = () => Console.WriteLine("lambda");
```

Символ _ (подчёркивание) можно использовать для пропуска двух и более параметров (для одиночного _ считается именем локальной переменной в целях обратной совместимости):

```
Func<int, int, int> lambda = (_, _) => 42;
```

Начиная с C# 10.0, для лямбда-выражений допускается явное указание типа возвращаемого значения перед списком параметров

```
ByRefDel lambda7 = ref int (ref int val) => ref val;

delegate ref int ByRefDel(ref int x);
```

ПРИМЕР: ИСПОЛЬЗОВАНИЕ ЛЯМБДА-ВЫРАЖЕНИЙ

```
string inputStr = Console.ReadLine();

Func<string, bool> condition = line => line.Length <= 10;
// Если строка не длиннее 10 символов, вывести ее
CondWrite(inputStr, condition);


condition = line => line.Length % 2 == 0;
// Если в строке чётное число символов, вывести ее
CondWrite(inputStr, condition);

void CondWrite(string output, Func<string, bool> condition)
{
    if (condition(output))
    {
        Console.WriteLine(output);
    }
}
```

Ввод:
Hello

Вывод:
Hello

Ввод:
ab

Вывод:
ab
ab

ПРИМЕР РАБОТЫ С КОРТЕЖАМИ В ЛЯМБДАХ

```
Func<(int, int), (int, int)> mod10 = ns => (ns.Item1 % 10, ns.Item2 % 10);

var numbers = (15,16);
var doubledNumbers = mod10(numbers);

Console.WriteLine($"{numbers} mod 10: {doubledNumbers}");
```

РЕАЛИЗАЦИЯ ЧЛЕНОВ КЛАССОВ ЧЕРЕЗ ЛЯМБДА-ВЫРАЖЕНИЯ

Синтаксис выражений допустим для:

- методов
- свойства только для чтения
- свойств
- конструктором
- деструкторов
- индексаторов

Реализация с использованием выражения хороша при однострочной реализации соответствующего члена класса

ПРИМЕР ЛЯМБДА СИНТАКСИСА КОНСТРУКТОРА И МЕТОДА

```
public class Person
{
    public Person(string firstName) => fname = firstName;
    public Person(string firstName, string lastName)
    {
        fname = firstName;
        lname = lastName;
    }

    private string fname;
    private string lname;

    public override string ToString() => $"{fname} {lname}".Trim();
    public void DisplayName() => Console.WriteLine(ToString());
}
```

Однострочная запись конструктора

Методы, допускающие однострочную запись

ПРИМЕР ЛЯМБДА СИНТАКСИС СВОЙСТВА

```
public class Location
{
    private string locationName;

    public Location(string name)
    {
        locationName = name;
    }
    public string Name => locationName;
}
```

Свойство, дающее только возможность чтения

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Геттер и сеттер, записанные в лямбда-синтаксисе

ПРИМЕР ЛЯМБДА СИНТАКСИСА ИНДЕКСАТОРА

```
using System;
using System.Collections.Generic;

public class Sports
{
    private string[] types = { "Baseball", "Basketball", "Football",
                               "Hockey", "Soccer", "Tennis",
                               "Volleyball" };

    public string this[int i]
    {
        get => types[i];
        set => types[i] = value;
    }
}
```

Геттер и сеттер индексатора, записанные в лямбда-синтаксисе

Пример взят из официального руководства Microsoft [http://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members]

СТАТИЧЕСКИЕ АНОНИМНЫЕ ФУНКЦИИ

- Нестатические и не **const** переменные и поля не захватываются
  - Статические члены и константы по прежнему доступны для захвата
- Нет доступа к ссылкам this/base, даже если такие имелись в охватывающей области видимости

В объявлении используется модификатор **static**

```
int value = 42;

Func<int> lambda = static () =>
{
    // Ошибка компиляции при попытке раскомментировать следующую строку.
    // return value + 10;
    return 42;
};
```

АНОНИМНЫЕ МЕТОДЫ КАК ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Анонимные методы могут выступать в качестве возвращаемых значений в функциях, возвращающих экземпляры делегат-типов или `Expression`:

```
static Func<double, double, double, double> GetFunction(int a, int b, int c)
{
    return (a, b, c) => a * a * b + c;
}


static Expression<Func<string, int>> GetExpression(string line)
{
    return (line) => int.Parse(line);
}
```

Уже упоминавшийся тип для дерева выражений

АНОНИМНЫЕ ФУНКЦИ И МОДИФИКАТОР PARAMS

При использовании делегат-типов с модификатором **params** при объявлении анонимных методов ключевое слово **params** опускается

```
int prod = 1;
int sum = 0;
ParamsDel mydel = (params int[] numbers) => { // Compile error, needs to hide
    foreach (int value in numbers)
    {
        sum += value;
        prod *= value;
    }
    Console.WriteLine("Sum = " + sum);
    Console.WriteLine("Composition = " + prod);
};
mydel(5, 1, 2, 3);
```

ЗАХВАТ ИТЕРАЦИОННОЙ ПЕРЕМЕННОЙ В ЦИКЛЕ FOR

Анонимные функции могут захватывать итерационные переменные цикла for (сохраняется последнее состояние переменной, т.е. копии значений не создаются)

```
Action[] actions = new Action[3];
for (int i = 0; i < actions.Length; ++i)
{
    actions[i] = () => Console.Write(i + " "); // Здесь нет вызова.
}
foreach (var lambda in actions)
{
    lambda(); // Вызов произойдёт здесь.
}
```

При захвате переменная **i** не копируется

Вывод:

3 3 3

РЕШЕНИЕ ПРОБЛЕМЫ ЗАХВАТА ПЕРЕМЕННОЙ В ЦИКЛЕ FOR

- Для решения проблемы при захвате достаточно добавить локальную переменную, которая каждый раз будет создаваться заново и захватываться

```
Action[] actions = new Action[3];

for (int i = 0; i < actions.Length; ++i)
{
    int tmp = i;
    actions[i] = () => Console.Write(tmp + " "); // Здесь нет вызова.
}
foreach (var lambda in actions)
{
    lambda(); // Вызов происходит здесь.
}
```

Каждый раз захватывается новое значение переменной

Вывод:
0 1 2

ЗАХВАТ ИТЕРАЦИОННОЙ ПЕРЕМЕННОЙ В ЦИКЛЕ FOREACH

- Захват итерационной переменной анонимными методами работает предсказуемо: т.к. захватывается конкретный элемент последовательности, пересечений в принципе не возникает

```
  Action[] actions = new Action[3];

  foreach(int val in new[] { 0, 1, 2})
  {
      actions[val] = () => Console.Write(val + " "); // Здесь нет вызова.
  }
  foreach (var lambda in actions)
  {
      lambda(); // Вызов происходит здесь.
}
```

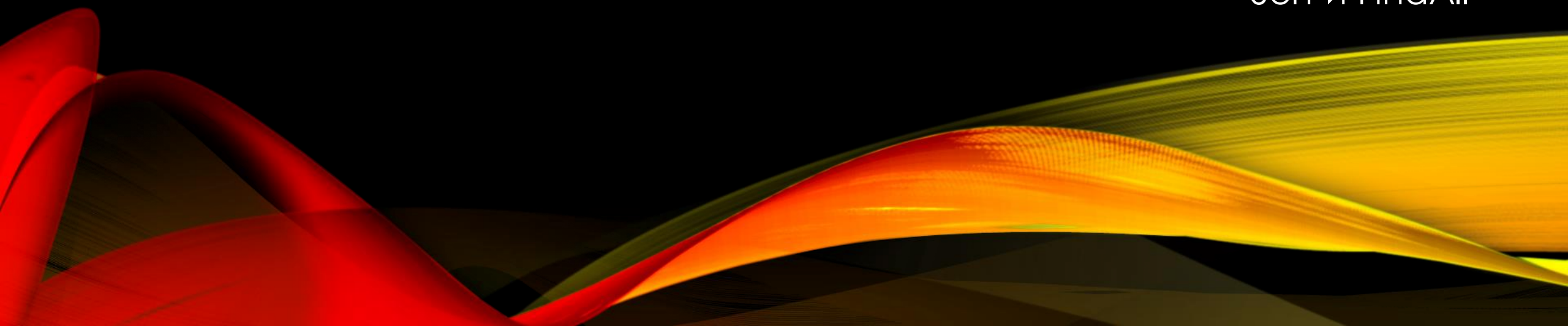
Захватываются никак не связанные друг с другом значения

Вывод:
0 1 2

МЕТОДЫ КЛАССА ARRAY С ПАРАМЕТРОМ-ДЕЛЕГАТОМ

ConvertAll и ForEach

Sort и FindAll



ПРИМЕР ИСПОЛЬЗОВАНИЯ CONVERTALL И FOREACH

```
using System;

int[] ints = { 2, 6, 4, 5, 17, 8, 10, 3, 1, 14 };
// Конвертация всех чисел в double:
double[] doubles = Array.ConvertAll(ints,
    value => value >= 10 ? value + 0.1 : value * 0.1);

// Вывод всех чисел в полученном массиве.
Array.ForEach(doubles, value => Console.Write($"{value:F1} "));
```

https://copyprogramming.com/howto/how-to-implement-array-convertall-method-in-c

Вывод:

```
0.2 0.6 0.4 0.5 17.1 0.8 10.1 0.3
0.1 14.1
```

ПОДРОБНО О CONVERTALL

Исходный тип элементов массива

Целевой тип элементов массива

Исходный массив, элементы которого будут преобразованы

```
public static TOutput[] ConvertAll<TInput,TOutput> (TInput[] array,
                                                    Converter<TInput,TOutput> converter);
```

Возвращаемый массив, с преобразованными элементами

Преобразование, задано делегатом

```
public delegate TOutput Converter<in TInput,out TOutput>(TInput input);
```

ПРИМЕР ПРИМЕР МЕТОДЫ SORT И FINDALL

```
using System;

double[] doubles = { 0.2, 0.6, 0.4, 0.5, 17.1, 0.8, 10.1, 0.3, 0.1, 14.1 };

// Сортировка всех чисел в порядке убывания и вывод:
Array.Sort(doubles, (value1, value2) => -value1.CompareTo(value2));
Array.ForEach(doubles, value => Console.Write($"{value:F1} "));
Console.WriteLine("\n");

// Собрать и вывести все числа, заканчивающиеся на .1:
double[] endWithPointOne = Array.FindAll(doubles, value => {
        double temp = Math.Round(value - Math.Truncate(value), 1);
        return temp - 0.1 <= 0.00001;
    });
Array.ForEach(endWithPointOne, value => Console.Write($"{value:F1} "));
```

Код работает некорректно, исправьте самостоятельно

Вывод:

```
17.1 14.1 10.1 0.8 0.6 0.5 0.4 0.3 0.2 0.1

17.1 14.1 10.1 0.1
```

ЕСТЕСТВЕННЫЕ ТИПЫ АНОНИМНЫХ МЕТОДОВ

Сами по себе анонимные методы не имеют типа вне контекста. Они преобразуются компилятором либо к делегат-типам, либо к Expression.

Начиная с C# 10.0 появляется такое понятие, как **естественные типы** анонимных методов. Компилятор может выводить тип по следующим правилам:

- Action/Func выводится при наличии подходящей сигнатуры;
- Генерируется автоматически (например, при наличии ref);
- Выведение типа для групп методов возможно только при отсутствии перегрузок/методов расширений;

- Вывод типа работает по ссылкам базового типа (Object/Delegate или Expression/LambdaExpression) по описанным выше правилам.

Вывод типа может быть невозможен, если компилятор не может однозначно определить типы аргументов.

ПРИМЕР: ЕСТЕСТВЕННЫЕ ТИПЫ АНОНИМНЫХ МЕТОДОВ

```
using System;
using System.Linq.Expressions;
```

```
// Тип выведется как Func<string, int>.
var parseToInt = (string line) => int.Parse(line);
// Тип выведется как Action.
var simplePrint = delegate () { Console.WriteLine("Natural type"); };
// Тип будет сгенерирован компилятором, хранение по ссылке Delegate:
Delegate refReturn = (ref string line) => ref line;
// Явное определение лямбда-выражения как Expression<Func<int, int, int>>:
Expression expression = (int first, int second) => first + second;
// Определение типа группы методов как Func<int> по ссылке Object.
var consoleRead = Console.Read;

// Ошибка: тип параметра не вывести однозначно (string или ReadOnlySpan<char>).
var parseToUnknown = line => int.Parse(line);
// Ошибка: группа методов представляет несколько перегрузок.
var methodGroupError = Console.Write;
```

ИСТОЧНИКИ

- Создание анонимных методов: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/delegate-operator
- О лямбда-выражениях: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambАнонимные методы: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/delegate-operator
- Лямбда-выражения: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions
- da-expression
- Общая спецификация анонимных функций: https://docs.microsoft.com/en-us/dotnet/csharp/delegate-class
- Статические анонимные функции в C# 9.0: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/static-anonymous-functions
- Отбрасывание параметров анонимных функций: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/lambda-discard-parameters
- Улучшение анонимных функций в C# 10.0: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-10.0/lambda-improvements
- Исходный код класса Array, содержащий методы с делегатами: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Array.cs
- Анонимные методы vs. Лямбда-выражения: https://stackoverflow.com/questions/299703/delegate-keyword-vs-lambda-notation
- Verhoeff, T. From Callbacks to Design Patterns (https://www.win.tue.nl/~wstomv/edu/2ip15/downloads/Series_04/callbacks.pdf)