Московский государственный технический университет
имени Н.Э. Баумана

Кафедра «Системы обработки информации и управления»

**Ю.Е. Гапанюк**

# КОНСПЕКТ ЛЕКЦИЙ

## по дисциплине «Архитектура автоматизированных систем обработки информации и управления» Профиль: «Архитектура программных систем»

## ЧАСТЬ 1
## РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ЯЗЫКЕ C#

Москва – 2026

# Оглавление

# 1 Краткая характеристика языка программирования C#

Современный объектно-ориентированный язык программирования общего назначения C# разработан компанией Microsoft для платформы .NET.

С его помощью можно разрабатывать консольные приложения, оконные приложения, веб-приложения, серверные API. Для обработки данных предназначены технологии LINQ и Entity Framework.

Первая версия C# появилась в начале 2000 годов и стех пор активно развивается. Создателем языка C# является Андерс Хейлсберг, который до работы над языком C# был разработчиком компилятора с языка Паскаль и среды разработки Delphi. Это безусловно сказалось на C#, который, не смотря на синтаксис, унаследованный от C++, впитал в себя лучшие структурные черты Паскаля.

# 2 Создание консольного приложения на языке C#

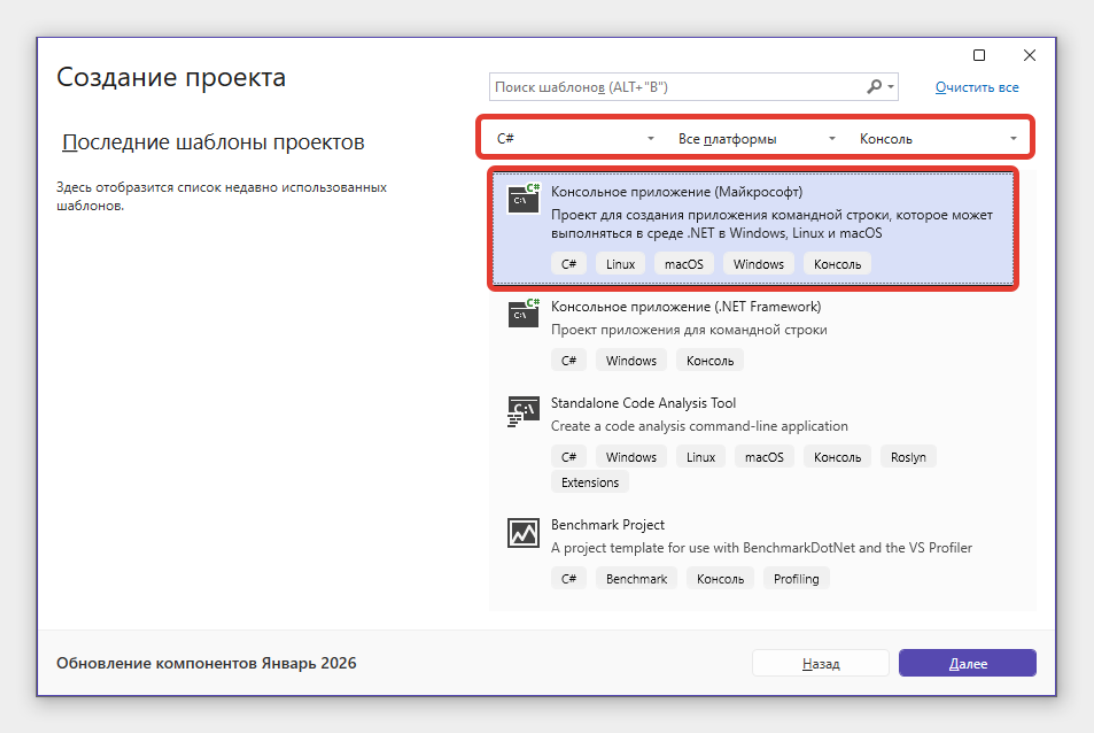


Рис. 1. Окно создания проекта.

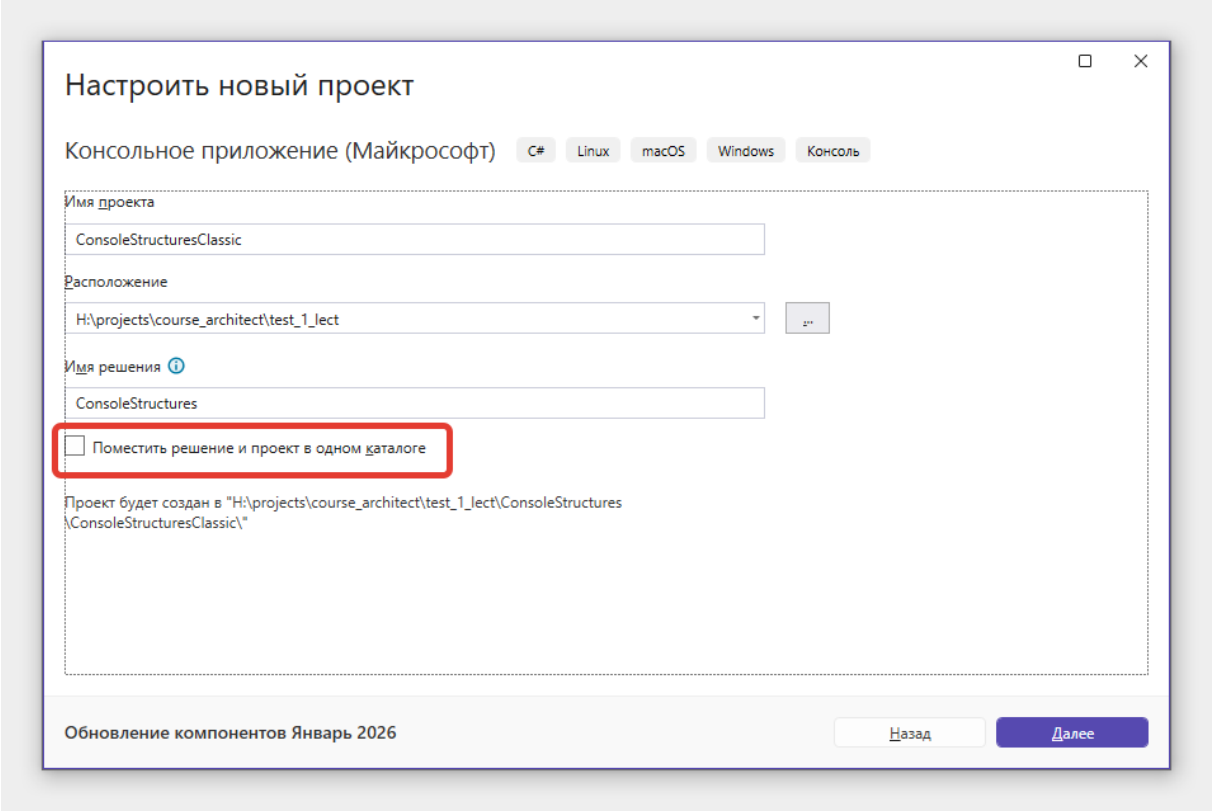


Рис. 2. Окно настройки проекта.

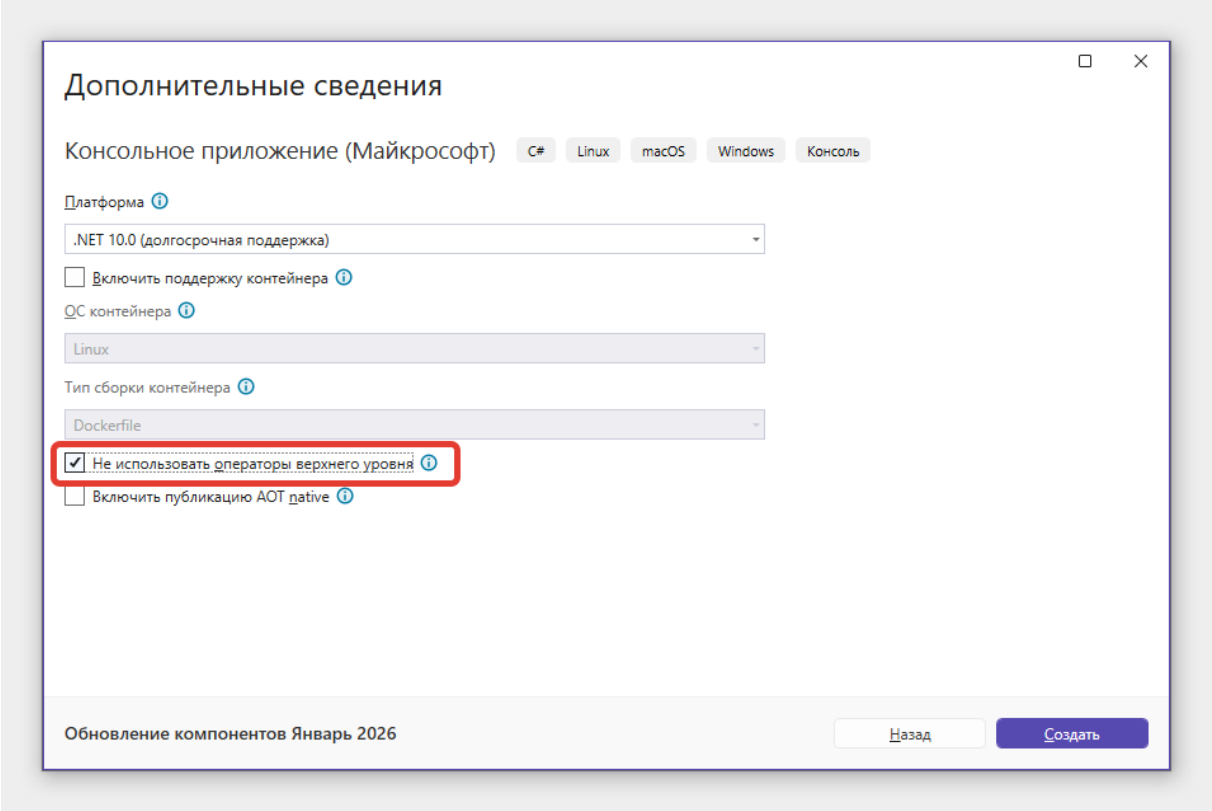


Рис. 3. Окно настройки проекта – дополнительные сведения.

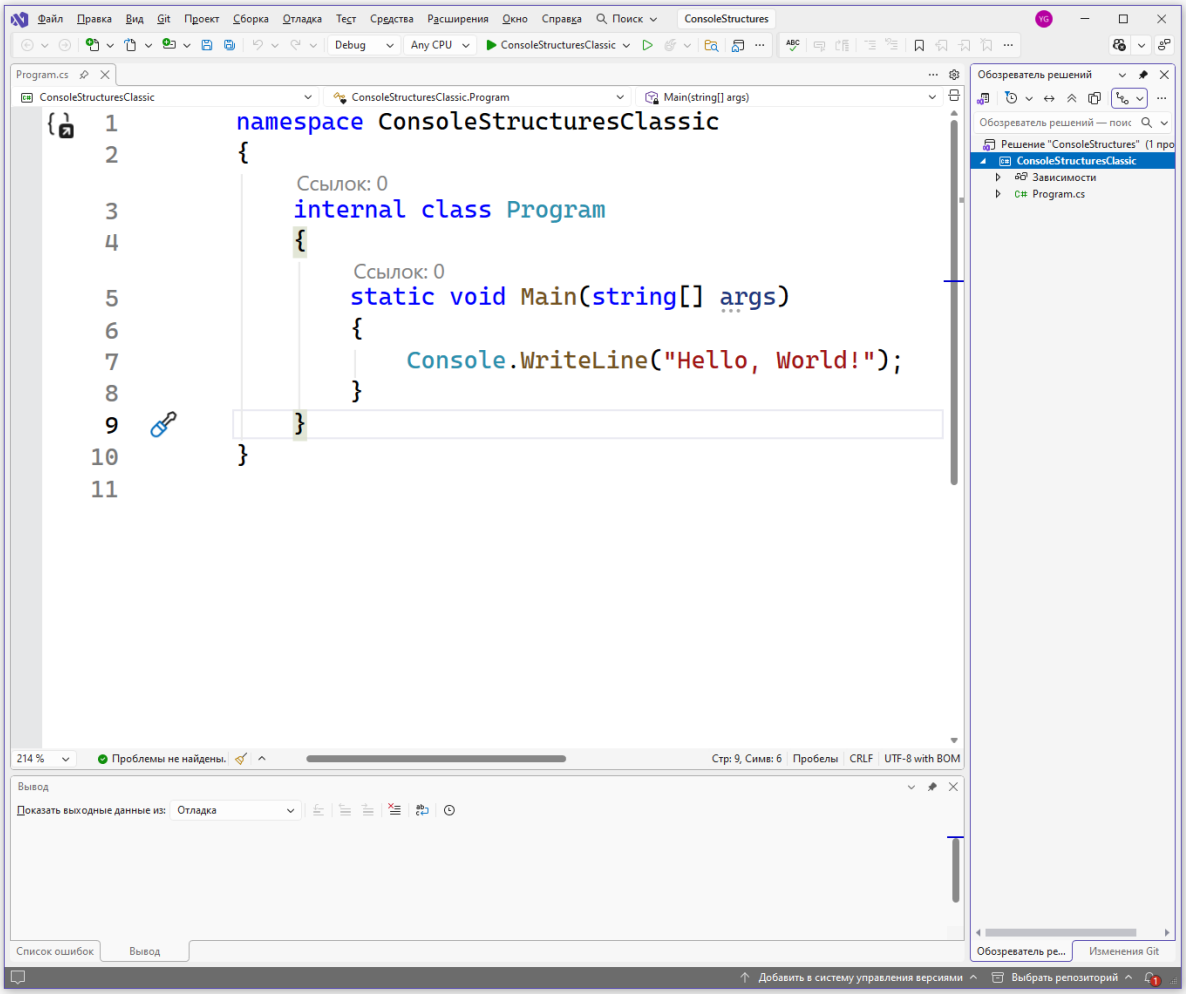


Рис. 4. Внешний вид с кодом «классического» проекта.

Запустим проект горячей клавишей F5.

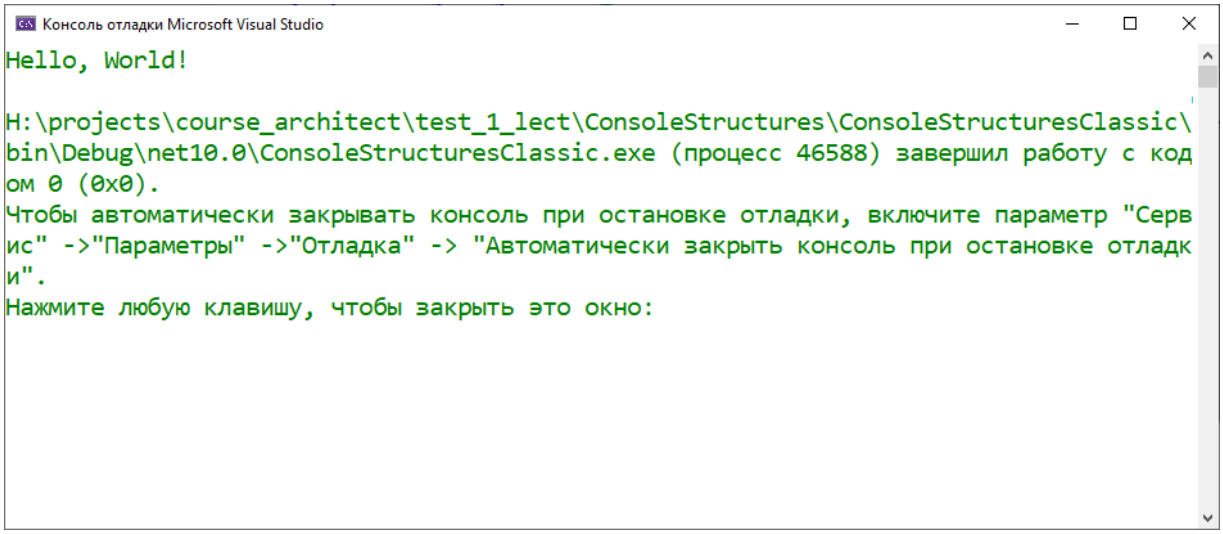


Рис. 5. Консоль с результатом запуска проекта.

Нажмем правую кнопку на решении, в контекстном меню выберем пункты «Добавить», «Создать в проект».

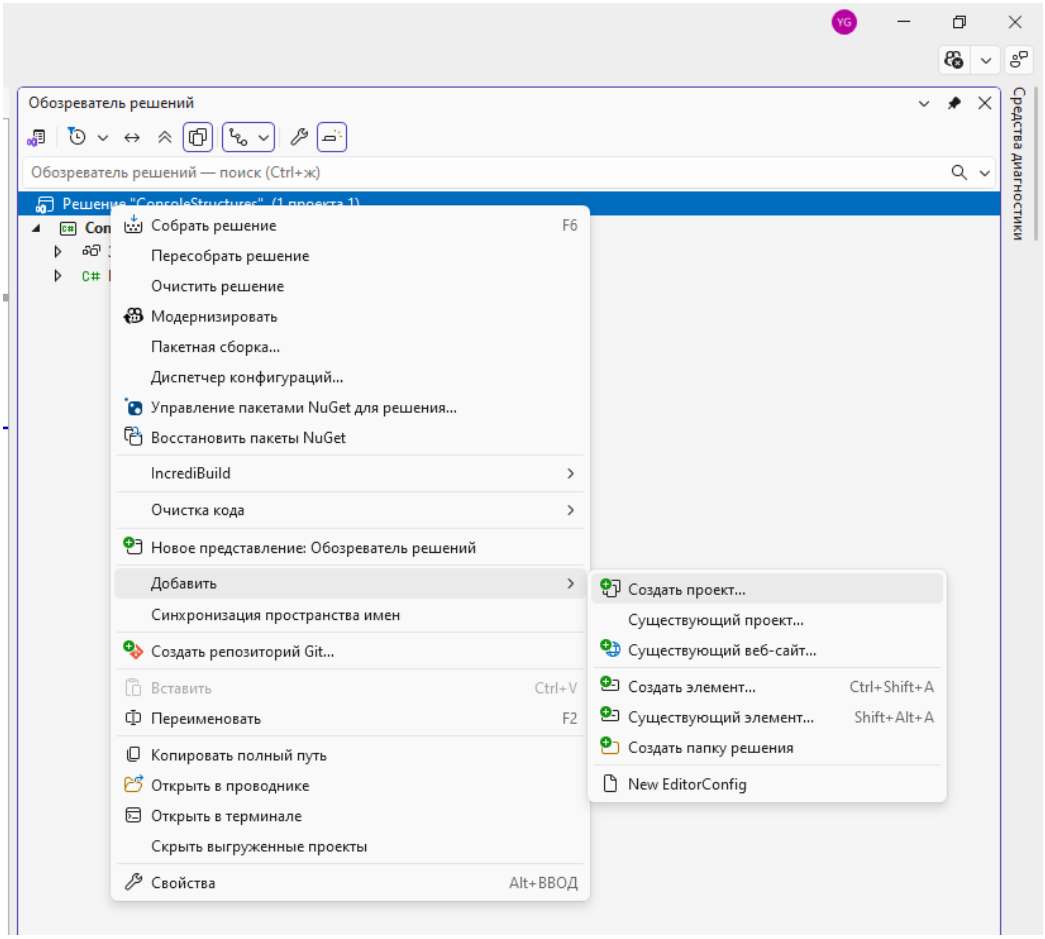


Рис. 6. Добавление проекта в решение.

Открывается окно выполнения проекта аналогично рис. 1, снова будем создавать консольный проект.

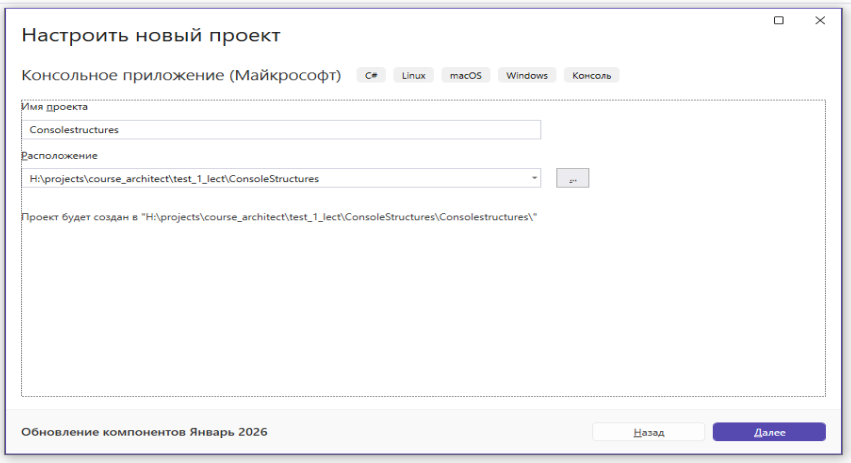


Рис. 7. Добавление проекта в решение – настройка проекта.

Обратите внимание, что в появившемся окне нет пункта про решение, так как оно уже создано.

В создаваемом проекте будем использовать «операторы верхнего уровня», флаг «Не использовать операторы верхнего уровня» должен быть выключен.

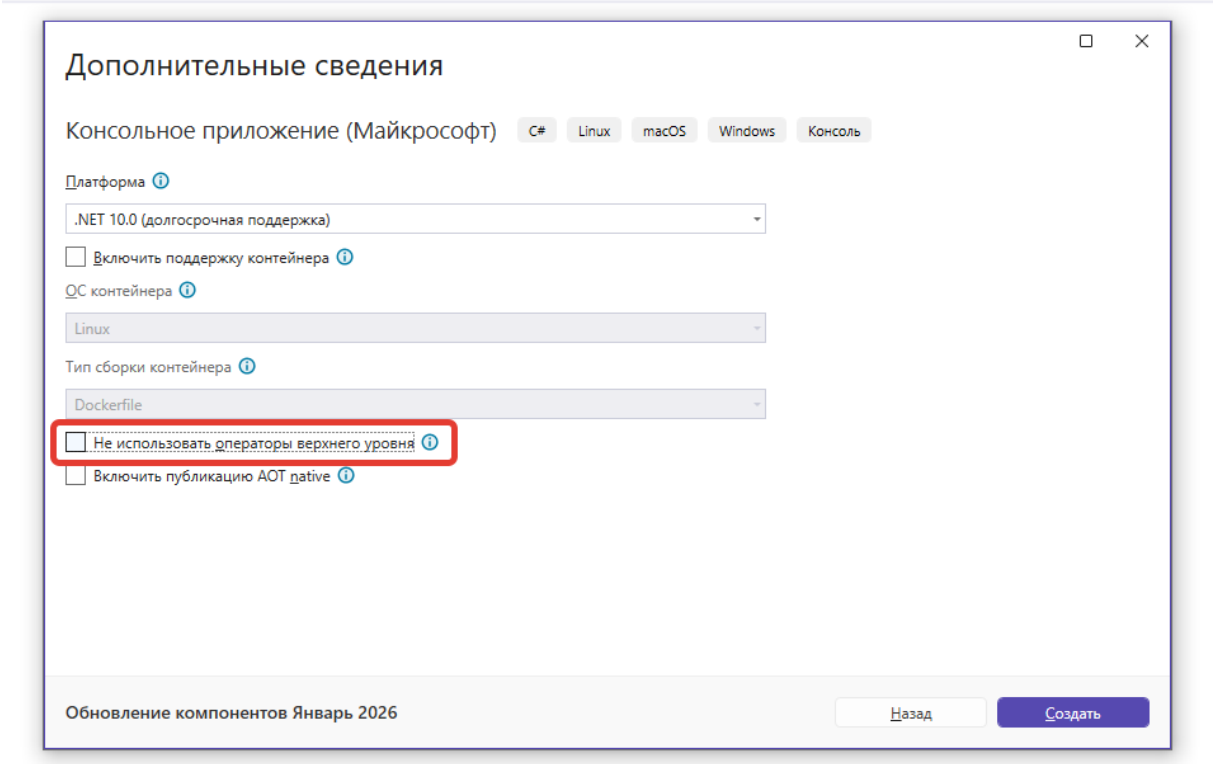


Рис. 8. Добавление проекта в решение – настройка проекта, доп. сведения.

Начиная с .NET 6 в консольном проекте по умолчанию используются операторы верхнего уровня (инструкции верхнего уровня) которые позволяют обходиться без описания класса и функции Main.

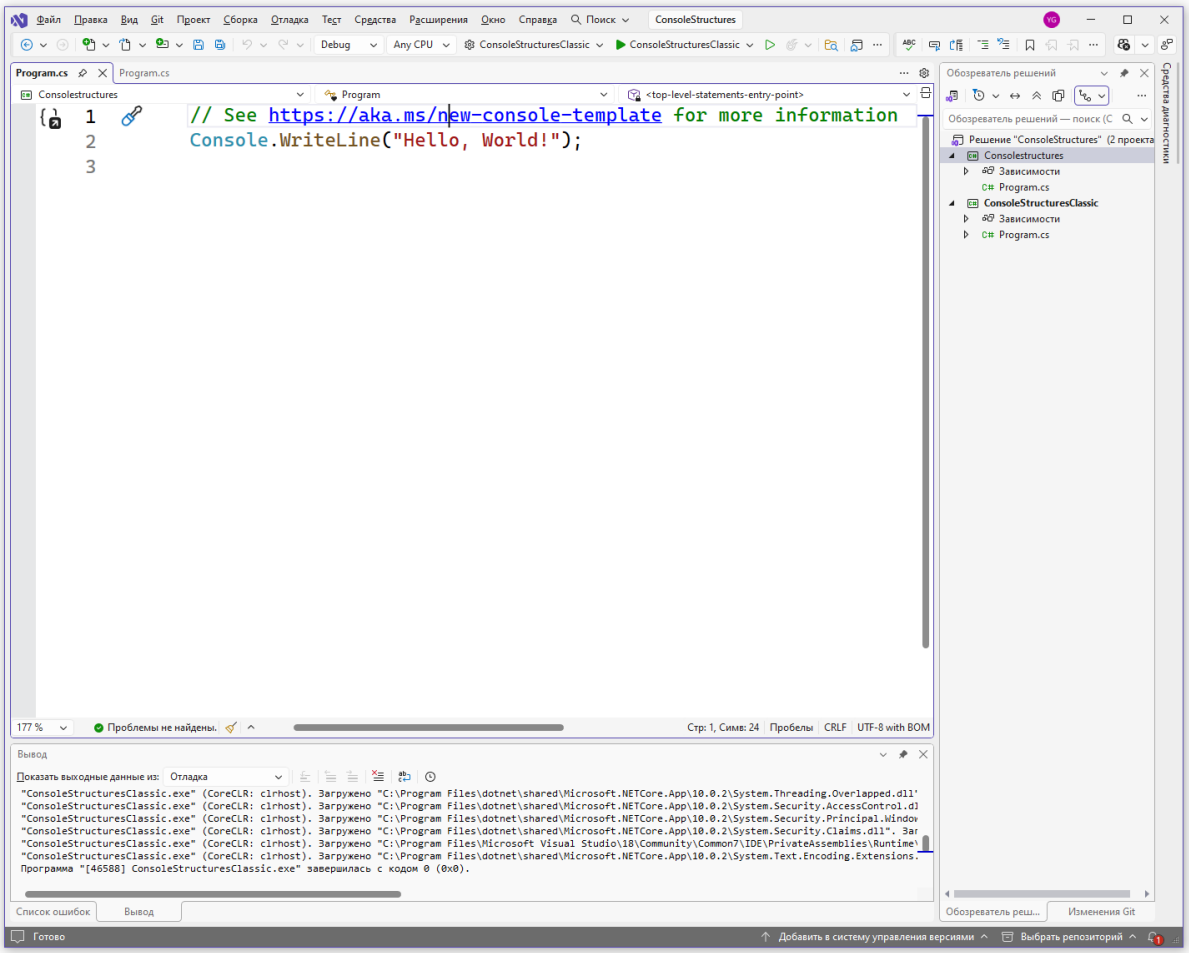


Рис. 9. Внешний вид проекта с инструкциями верхнего уровня.

Результат запуска проекта совпадает с рисунком 5.

Сделаем текущий проект активным запускаемым проектом. Для этого необходимо нажать правую кнопку на новом проекте, в контекстном меню выбрать пункт «Назначить в качестве запускаемого проекта». После этого новый проект будет отображаться жирным шрифтом в списке проектов.

В решении Visual Studio только один проект может быть назначен как запускаемый проект. Это текущий активный проект, который запускается при нажатии горячей клавиши F5.

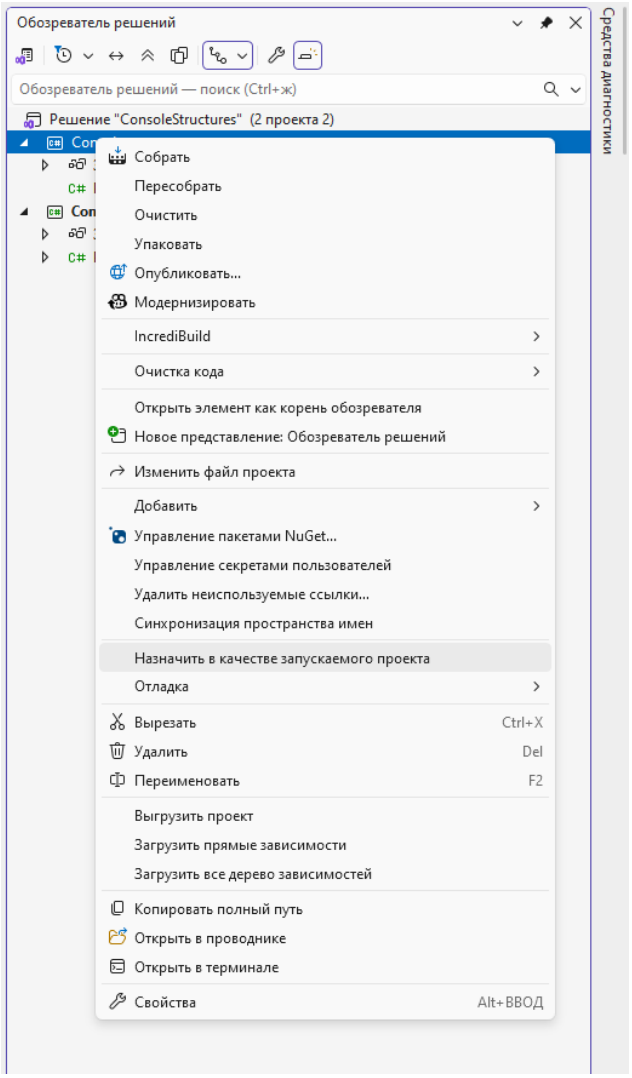


Рис. 10. Назначение запускаемого проекта (текущего активного проекта).

# 3 Базовые возможности языка C#, необходимые для создания консольных приложений

## *3.1 Консольный ввод-вывод*

Средства консольного ввода-вывода в C# напоминают аналогичные средства в Java и очень уступают по возможностям консольному вводу-выводу в C++.

Ввод-вывод в консоль осуществляется посредством класса Console, у которого предусмотрены статические методы ввода-вывода данных:

- Console.WriteLine – вывод данных с переводом строки;
- Console.Write – вывод данных без перевода строки;
- Console.Read – чтение текущего символа;
- Console.ReadKey – чтение текущего символа или функциональной клавиши;
- Console.ReadLine – чтение строки до нажатия ввода.

У класса Console также есть методы для чтения и изменения цвета текста, размеров окна и другие.

Для ввода данных в консоли существует только метод Console.ReadLine, позволяющий ввести только строковый тип данных, но не позволяющий вводить числовые типы данных. Здесь язык C# уступает по возможностям консольной библиотеке C++. Поэтому в языке C# при разработке консольных приложений актуальным является преобразование строкового типа данных в другие типы.

Методы Console.WriteLine и Console.Write имеют большое количество перегрузок для вывода данных различных типов.

Также существует перегрузка для форматированного вывода, когда первым параметром указывается строка формата, а далее передается произвольное количество параметров, которые подставляются в строку формата.

*Пример использования метода WriteLine со строкой формата:*

```
int p1 = 2;
int p2 = 4;
Console.WriteLine("{0} умножить на {1} = {2}", p1, p2, p1*p2);
```

В консоль будет выведено:

```
2 умножить на 4 = 8
```

Фигурные скобки в строке формата означают ссылку на соответствующий параметр, причем параметры нумеруются с нуля. Таким образом, вместо {0} будет подставлена переменная p1, вместо {1} – переменная p2, вместо {2} – выражение p1*p2.

*Пример, содержащий форматирование параметра:*

```
double d3 = 1.12345678;
Console.WriteLine("{0} округленное до 3 разрядов = {0:F3}", d3);
```

В консоль будет выведено:

```
1,12345678 округленное до 3 разрядов = 1,123
```

Выражение {0:F3} означает, что нулевой параметр нужно вывести в виде числа с плавающей точкой, округлив до 3 знаков после разделителя разрядов.

**ЗАДАНИЕ**

**Введите название страны и столицы страны и выведите в консоль, например «Москва – столица России!».**

### *3.2 Строковая интерполяция*

Строковая интерполяция (string interpolation) – это возможность упрощения синтаксиса, которая появилась в версии C# 6.

Строковая интерполяция позволяет указывать в строке шаблон для ее форматирования на основе переменных. Необходимо отметить, что похожие возможности есть в других языках программирования, в частности в Python и PHP.

Пример, использующий строковую интерполяцию:

```
string City = "Moscow";
string Country = "Russia";

//Старый способ вывода с помощью конкатенации строк
Console.WriteLine(City + ", " + Country);
//Старый способ вывода с помощью string.Format
Console.WriteLine(string.Format("{0}, {1}", City, Country));
//Новый способ вывода с помощью строковой интерполяции
Console.WriteLine($"{City}, {Country}");
```

Признаком использования строковой интерполяции является то, что при объявлении строки перед кавычками ставится символ доллара. Если в

такой строке указано выражение в фигурных скобках, то оно автоматически вычисляется.

Необходимо отметить, что строковая интерполяция в настоящее время является основным способом соединения строк при консольном выводе.

Результаты вывода в консоль:

```
Moscow, Russia
Moscow, Russia
Moscow, Russia
```

**ЗАДАНИЕ**

**Введите название страны и столицы страны и выведите в консоль с использованием строковой интерполяции, например «Москва – столица России!».**

### *3.3 Преобразования типов*

Преобразование типов выполняется аналогично тому, как это происходит в C++ и Java с использованием оператора приведения типов – круглые скобки.

*Пример преобразования переменной типа double в тип int:*

```
double d1 = 123.45;
int i1 = (int)d1;
```

Очень часто, особенно при консольном вводе, необходимо преобразовать строковое представление числа (введенное с клавиатуры) в числовой формат. Это можно сделать тремя способами.

Первый состоит в том, что у каждого класса-типа существует статический метод Parse, преобразующий строку в числовой тип данных.

*Пример преобразования типов с использованием метода Parse:*

```
int i2 = int.Parse("123");
double d2 = double.Parse("123,45");
```

Метод Parse использует настройки локализации операционной системы, в частности для переменной типа double в качестве разделителя целой и дробной части используется запятая. Если строка не содержит корректного представления числа, метод Parse генерирует исключение FormatException (исключения подробнее рассмотрены ниже).

При втором способе вместо метода Parse используется TryParse. В случае неудачного преобразования данный метод не генерирует исключение, но возвращает логическое значение false. Возвращаемое число представляется выходным параметром (out-параметр).

*Пример использования метода TryParse:*

```
int i3;
bool result = int.TryParse("123", out i3);
```

*В современных версиях C# можно объявить переменную при вызове функции:*

```
bool result = int.TryParse("123", out int i3);
```

Третий способ заключается в применении класса Convert, который содержит большое количество статических методов и может преобразовать значения большинства базовых между собой.

*Пример использования класса Convert:*

```
int i4 = Convert.ToInt32("123");
```

Если же строка не содержит корректное представление числа, класс Convert, как и метод Parse, генерирует исключение FormatException.

## ЗАДАНИЕ

**Введите с клавиатуры целое число (оно будет введено как строка). Преобразуйте введенную строку к целому числу. С использованием строковых интерполяций выведите, например «Число 333 введено успешно» или «Введенная строка qwerty не может быть преобразована в число».**

**Если конвертация в целое число произведена успешно (используйте условный оператор из примера в следующем разделе), то с помощью метода Parse и класса Convert преобразуйте целое число в действительное число. Выведите полученные действительные числа с помощью строковой интерполяции.**

### *3.4 Простые массивы, условные операторы и циклы*

Для объявления массивов в C# используется синтаксис:

```
int[] array;
```

В языке C++ квадратные скобки должны ставить не после имени типа, а после имени переменной: int array[]. В Java допустимы оба варианта объявления.

Если необходимо присвоить начальные значения элементам массива, то форма объявления следующая:

```
int[] array = new int[3] {1, 2, 3};
```

Аналогично для строкового типа:

```
string[] strs =
         new string[3] { "строка1", "строка2", "строка3" };
```

*Пример условного оператора if:*

```
if (str == "строка1")
{
    Console.WriteLine("if: str == \"строка1\"");
}
else
{
    Console.WriteLine("if: str != \"строка1\"");
}
```

В качестве условия проверяется значение логического типа, который отсутствует в стандартном C++, но есть в Паскале, Java, C#.

Счетный цикл for такой же как в C++ и Java. В круглых скобках после for указываются три оператора – начальное присвоение значения переменной цикла; условие выхода из цикла; оператор, который выполняется при переходе к следующему шагу цикла. Операторы разделяются точкой с запятой.

*Пример цикла for:*

```
for (int i = 0; i < 3; i++)
    Console.Write(i);
```

В данном примере оператор вывода переменной i не вложен в фигурные скобки, использование скобок не является обязательным, так как это единственный оператор цикла. Если бы в цикле выполнялось несколько действий, то их необходимо было бы вложить в фигурные скобки.

## ЗАДАНИЕ

**Объявите в программе массив со значениями: 1,2,3,4,5,6,7,8 (ввод с клавиатуры не используется. Выведите на экран только четные элементы массива.**

### *3.5 Задание для закрепления материала*

Необходимо написать на C# программу в виде консольного приложения, которая решает биквадратное уравнение и сортирует корни. Программа выполняет следующие действия:

1. Позволяет ввести с клавиатуры три коэффициента – a, b, c.
2. Не нужно делать проверок на нулевые значения, некорректный ввод, предполагается, что пользователь вводит корректные значения для случая имеющего точно четыре корня.
3. Вычисляет корни по формуле:

$$x_{1,2,3,4} = \pm\sqrt{\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}}$$

4. Не нужно делать дополнительных проверок, предполагается, что дискриминант всегда больше нуля, и корней точно четыре.
5. Полученные корни необходимо поместить в массив или изначально сохранять корни как элементы массива.
6. Полученный массив корней необходимо отсортировать по возрастанию любым алгоритмом. Пожалуйста укажите в комментариях какой алгоритм сортировки Вы используете.
7. Отсортированный массив необходимо вывести на экран.

Решение:

```
using System;

// ================= ОСНОВНОЙ КОД ПРОГРАММЫ =================

double a, b, c;
double[] roots = new double[4];
double[] roots_bubble = new double[4];
double[] roots_selection = new double[4];
int n;
string[] input;

Console.WriteLine("Решение биквадратного уравнения: ax^4 + bx^2 + c = 0");
Console.WriteLine();

// Ввод коэффициентов
Console.Write("Введите коэффициенты a, b, c через пробел: ");
input = Console.ReadLine().Split(' ');
a = double.Parse(input[0]);
b = double.Parse(input[1]);
c = double.Parse(input[2]);

Console.WriteLine();
Console.WriteLine("-----------------------------------------------");
Console.WriteLine("Уравнение: {0:F2}x^4 + {1:F2}x^2 + {2:F2} = 0", a, b, c);
Console.WriteLine("-----------------------------------------------");

// Решение уравнения
n = SolveBiquadratic(a, b, c, roots);

Console.WriteLine();
Console.WriteLine("КОРНИ ДО СОРТИРОВКИ:");
PrintArray(roots, n);

// Копируем массив для двух разных сортировок
CopyArray(roots, roots_bubble, n);
CopyArray(roots, roots_selection, n);

// Применяем обе сортировки
BubbleSort(roots_bubble, n);
SelectionSort(roots_selection, n);

// Выводим результаты
Console.WriteLine();
Console.WriteLine("===============================================");
Console.WriteLine("РЕЗУЛЬТАТЫ СОРТИРОВКИ:");
Console.WriteLine("===============================================");

Console.WriteLine();
Console.WriteLine("1. ПУЗЫРЬКОВАЯ СОРТИРОВКА (Bubble Sort):");
PrintArray(roots_bubble, n);

Console.WriteLine();
Console.WriteLine("2. СОРТИРОВКА ВЫБОРОМ (Selection Sort):");
PrintArray(roots_selection, n);

// Проверка идентичности результатов
Console.WriteLine();
Console.WriteLine("-----------------------------------------------");
if (ArraysEqual(roots_bubble, roots_selection, n))
{
    Console.WriteLine("Оба метода дают ОДИНАКОВЫЙ результат");
}
```

```
else
{
    Console.WriteLine("ВНИМАНИЕ: Результаты РАЗЛИЧАЮТСЯ!");
}

Console.WriteLine();
Console.WriteLine("============================================");
Console.WriteLine("ТЕСТОВЫЕ ПРИМЕРЫ:");
Console.WriteLine("============================================");
Console.WriteLine("1) a=1, b=-5,  c=4  → корни: -2, -1, 1, 2");
Console.WriteLine("2) a=1, b=-10, c=9  → корни: -3, -1, 1, 3");
Console.WriteLine("3) a=1, b=-13, c=36 → корни: -3, -2, 2, 3");
Console.WriteLine("4) a=1, b=-17, c=16 → корни: -4, -1, 1, 4");
Console.WriteLine("============================================");

Console.WriteLine();
Console.Write("Нажмите любую клавишу для выхода...");
Console.ReadKey();


// ================= ФУНКЦИИ =================

/* Решение биквадратного уравнения ax^4 + bx^2 + c = 0
   Возвращает количество корней */
static int SolveBiquadratic(double a, double b, double c, double[] roots)
{
    // Замена y = x^2, получаем ay^2 + by + c = 0
    double D = b * b - 4 * a * c;
    double y1 = (-b + Math.Sqrt(D)) / (2 * a);
    double y2 = (-b - Math.Sqrt(D)) / (2 * a);

    // Находим x = ±√y для каждого y
    roots[0] = Math.Sqrt(y1);
    roots[1] = -Math.Sqrt(y1);
    roots[2] = Math.Sqrt(y2);
    roots[3] = -Math.Sqrt(y2);

    return 4;
}

/* Пузырьковая сортировка массива по возрастанию */
static void BubbleSort(double[] arr, int n)
{
    int i, j;
    double temp;

    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

/* Сортировка выбором - находим минимум и ставим на своё место */
static void SelectionSort(double[] arr, int n)
{
    int i, j, min_idx;
```

```
    double temp;

    // Для каждой позиции находим минимум в оставшейся части
    for (i = 0; i < n - 1; i++)
    {
        min_idx = i;  // предполагаем, что минимум на позиции i

        // Ищем минимум среди оставшихся элементов
        for (j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[min_idx])
            {
                min_idx = j;
            }
        }

        // Меняем местами найденный минимум с элементом на позиции i
        temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

/* Вывод массива */
static void PrintArray(double[] arr, int n)
{
    for (int i = 0; i < n; i++)
    {
        Console.WriteLine("  x{0} = {1,7:F3}", i + 1, arr[i]);
    }
}

/* Копирование массива */
static void CopyArray(double[] src, double[] dst, int n)
{
    for (int i = 0; i < n; i++)
    {
        dst[i] = src[i];
    }
}

/* Проверка идентичности массивов */
static bool ArraysEqual(double[] arr1, double[] arr2, int n)
{
    for (int i = 0; i < n; i++)
    {
        if (Math.Abs(arr1[i] - arr2[i]) > 0.0001)
        {
            return false;
        }
    }
    return true;
}
```

# 4 Основные типы данных языка C#

## *4.1 Организация типов данных в языке C#*

Особенность типов данных языка C# заключается в том, что для всей среды исполнения .NET используется единая общая система типов – CTS (Common Type System), причем как в C#, так и в Visual Basic .NET и других языках .NET-платформы.

Все типы данных разделяются на типы-значения (value type) и ссылочные типы (reference type).

Типы-значения хранятся в области памяти, доступ к которой организован в виде стека. К ним относятся:

- базовые типы данных;
- перечисления (enum);
- структуры (struct).

Если переменная типа-значения передается как параметр в метод, то в стеке делается копия значения и передается в метод, а исходное значение не изменяется. В частности, такой порядок копирования значений удобен для примитивных типов.

Ссылочные типы хранятся в динамически распределяемой области памяти, которую принято называть кучей (heap). Для работы с ней используются ссылки, фактически являющиеся указателями. Ссылочными типами являются:

- классы;
- интерфейсы;
- массивы;
- делегаты.

Если переменная ссылочного типа передается в метод как параметр, то фактически передается указатель на эту переменную, при этом не делается копия значения (как в случае типа-значения). Поэтому если

значение переменной, переданное по ссылке, изменяется в методе, то исходное значение вследствие этого также меняется. Такой порядок копирования значений удобен, в частности, для передачи объектов классов.

В отличие от C++, в языке C# использование типов-значений и ссылочных типов практически ничем не различается, для обозначения ссылок не применяют специальные символы «&». Использовать указатели (которые как и в C++ обозначаются символом «*») можно в так называемом небезопасном (unsafe) режиме. Этот режим подходит для взаимодействия с аппаратными средствами, оптимизации производительности, в обычных приложениях применять unsafe-режим категорически не рекомендуется. В данном издании unsafe-режим не рассматривается.

Структура – это механизм языка, который представляет собой аналог класса, но реализован на основе стека. Синтаксически структуры и классы в программе на C# мало различаются. Однако в некоторых случаях использование структур вместо классов позволяет повысить производительность приложения, иногда очень значительно. Например, массив структур хранится в стеке последовательно, обработка такого фрагмента данных не требует дополнительных переходов. Массив объектов классов хранится в виде массива указателей, при их обработке необходимо выполнить дополнительные переходы по указателям, что может существенно увеличить время обработки данных. В современном варианте языка C# структуры в основном являются механизмом оптимизации производительности приложений.

### *4.2 Базовые типы данных*

Базовые типы данных являются типами-значениями и хранятся в стеке.

У каждого типа есть полное наименование, принятое в CTS, а также краткое, используемое в C#. Допустимы оба варианта наименований, но для удобства разработчики обычно предпочитают краткие наименования.

Целочисленные типы данных представлены в таблице 1.

*Таблица 1*

**Целочисленные типы данных**

| **Наименование в C#** | **Наименование в CTS** | **Описание** |
|---|---|---|
| sbyte | System.SByte | Целое 8-битное число со знаком |
| short | System.Intl6 | Целое 16-битное число со знаком |
| int | System.Int32 | Целое 32-битное число со знаком |
| long | System.Int64 | Целое 64-битное число со знаком |
| byte | System.Byte | Целое 8-битное число без знака |
| ushort | System.UIntl6 | Целое 16-битное число без знака |
| uint | System.UInt32 | Целое 32-битное число без знака |
| ulong | System.UInt64 | Целое 64-битное число без знака |

Типы данных с плавающей точкой представлены в таблице 2.

*Таблица 2*

**Типы данных с плавающей точкой**

| **Наименование в C#** | **Наименование в CTS** | **Описание** |
|---|---|---|
| float | System.Single | 32-битное число с плавающей точкой одинарной точности, около 7 значащих цифр после запятой |
| double | System.Double | 64-битное число с плавающей точкой двойной точности, около 15 значащих цифр после запятой |
| decimal | System.Decimal | 128-битное число с плавающей |

| | | |
|---|---|---|
| | | точкой повышенной точности, около 28 значащих цифр после запятой |

Внутреннее представление типа decimal отличается от внутреннего представления типов float и double.

Описание логического типа представлено в таблице 3.

*Таблица 3*

**Логический тип данных**

| **Наименование в C#** | **Наименование в CTS** | **Описание** |
|---|---|---|
| bool | System.Boolean | Может принимать значения true или false. |

В условных операторах C# и Java, как и в Паскале используется логический тип. В классических C и C++ он отсутствует, его роль выполняет целочисленный тип, но в некоторых современных диалектах C++ логический тип применяется.

Описание символьного и строкового типов данных представлено в таблице 4.

*Таблица 4*

**Символьный и строковый типы данных**

| **Наименование в C#** | **Наименование в CTS** | **Описание** |
|---|---|---|
| char | System.Char | Символ Unicode |
| string | System.String | Строка символов Unicode |

Тип string является не типом-значением, а ссылочным типом.

В языке C# ограничителем для символа служит одинарный апостроф, а для строки двойная кавычка. Пример объявления символа и строки:

```
char c = '1';
```

```
string str = "строка";
```

Корневой тип в CTS – тип System.Object (в C# – object). От него наследуются все ссылочные типы и типы-значения. Сам тип object является ссылочным.

Большинство базовых типов являются типами-значениями, но типы string и object стали исключением из этого правила.

# 5 Основные конструкции программирования языка C#

## *5.1 Пространства имен и сборки*

Программа на языке C# начинается с операторов using, каждый из которых указывает пространство имен для библиотечных классов. Любой класс в языке C# должен быть объявлен в каком-либо пространстве имен с использование оператора namespace.

Пространства имен представляют собой древовидную структуру. В каждой ее ветви содержатся вложенные классы и вложенные пространства имен. Если с помощью оператора using подключаются классы какого-либо пространства имен, например

```
using System;
```

то классы вложенных пространств имен при этом автоматически не подключаются. Поэтому их необходимо подключать отдельными директивами:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

Пространства имен представляют собой логическую структуру для систематизации классов. Выясним, как эта структура соотносится с откомпилированными классами.

Откомпилированный бинарный код для платформы .NET хранится в файлах сборок (assembly). Файл может иметь расширение .dll или .exe по аналогии с библиотеками и исполняемыми файлами ОС Windows. Но данные файлы содержат бинарный код, выполняющийся только на платформе .NET. Если же она не установлена, то ОС Windows не запустит такой исполняемый файл.

Файлы сборок следует подключать в разделе References проекта (рис. 11). В русской версии Visual Studio раздел References называется Ссылки. При нажатии правой кнопкой мыши на пункт References открывается диалог по добавлению новых сборок.

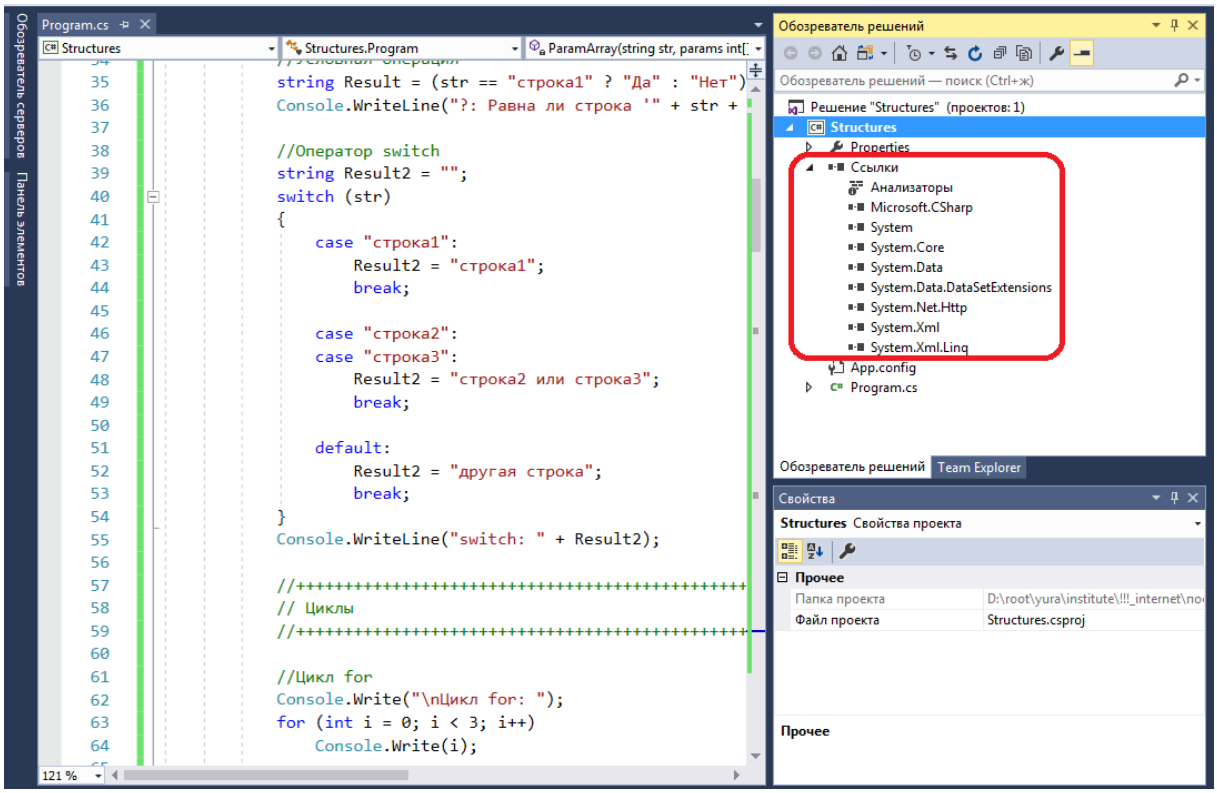


Рис. 11. Файлы сборок.

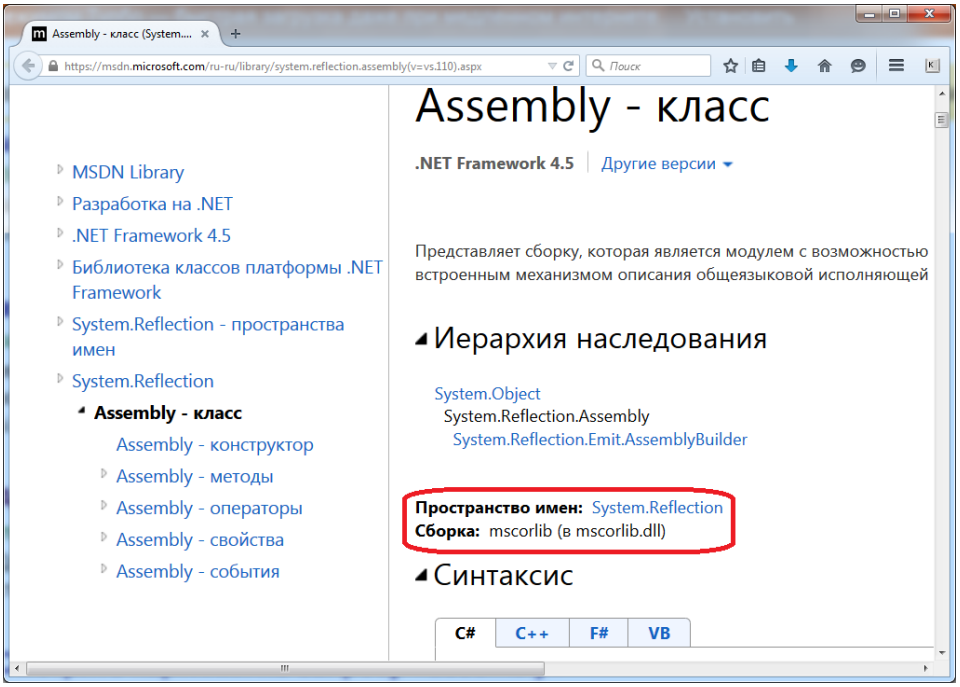


Рис. 12. Указание пространства имен и сборки в справочной системе.

Из рисунка 11 видно, что раздел References обычно содержит те же значения что и операторы using. Однако это принципиально разная информация. В разделе References находятся имена физических файлов сборок – файлов, которые содержат бинарный код, присоединяемый к проекту. Секция using ссылается на логическое название в дереве пространства имен.

При этом возможна ситуация, когда классы из одного и того же пространства имен находятся в разных сборках, и наоборот, одна сборка содержит классы из разных пространств имен. Поэтому в справочной системе Microsoft для каждого класса указано как имя сборки, так и пространство имен (рис 12).

Стоит отметить, что концепция пространства имен – специфика Microsoft.

В классическом языке C++ пространства имен отсутствуют, вместо них применяется подключение заголовочных файлов. Однако в версии языка C++, предлагаемой Microsoft, используется такой же механизм пространств имен, как и в C#.

В Java существует концепция похожая на пространства имен – пакеты (package). По аналогии с пространством имен каждый класс может быть включен в пакет. Но в данной концепции существуют ограничения – пакет является каталогом файловой системы, в который вложены файлы классов. Если имя пакета составное (содержит точку), то это предполагает вложенность соответствующих каталогов. Один файл в Java не может содержать более одного класса, следовательно, файл-класс однозначно соответствует пакету-каталогу. Пространства имен в языке C# – более гибкий механизм, однако сторонники Java полагают, что подобная жесткость позволяет избежать ошибок и несоответствий, встречающихся в пространствах имен.

## *5.2 Основная программа и параметры командной строки*

В тексте программы (если мы не используем инструкции верхнего уровня) приведены конструкции:

```
namespace Structures
{
    internal class Program
    {
    …
    }
}
```

Команда namespace объявляет пространство имен, а class – класс. Модификатор internal указывает что класс виден только в рамках текущей сборки. Команда namespace может содержать несколько классов. Чтобы сослаться на класс Program следует применить директиву:

```
using Structures;
```

Основной исполняемый метод консольного приложения – метод Main:

```
static void Main(string[] args)
  {
    …
  }
```

Строковый массив параметров метода args содержит параметры (аргументы) командной строки, которые могут быть заданы при вызове консольного приложения. В Visual Studio в целях отладки данные параметры можно указать в меню «Отладка», «Свойства отладки для проекта …».

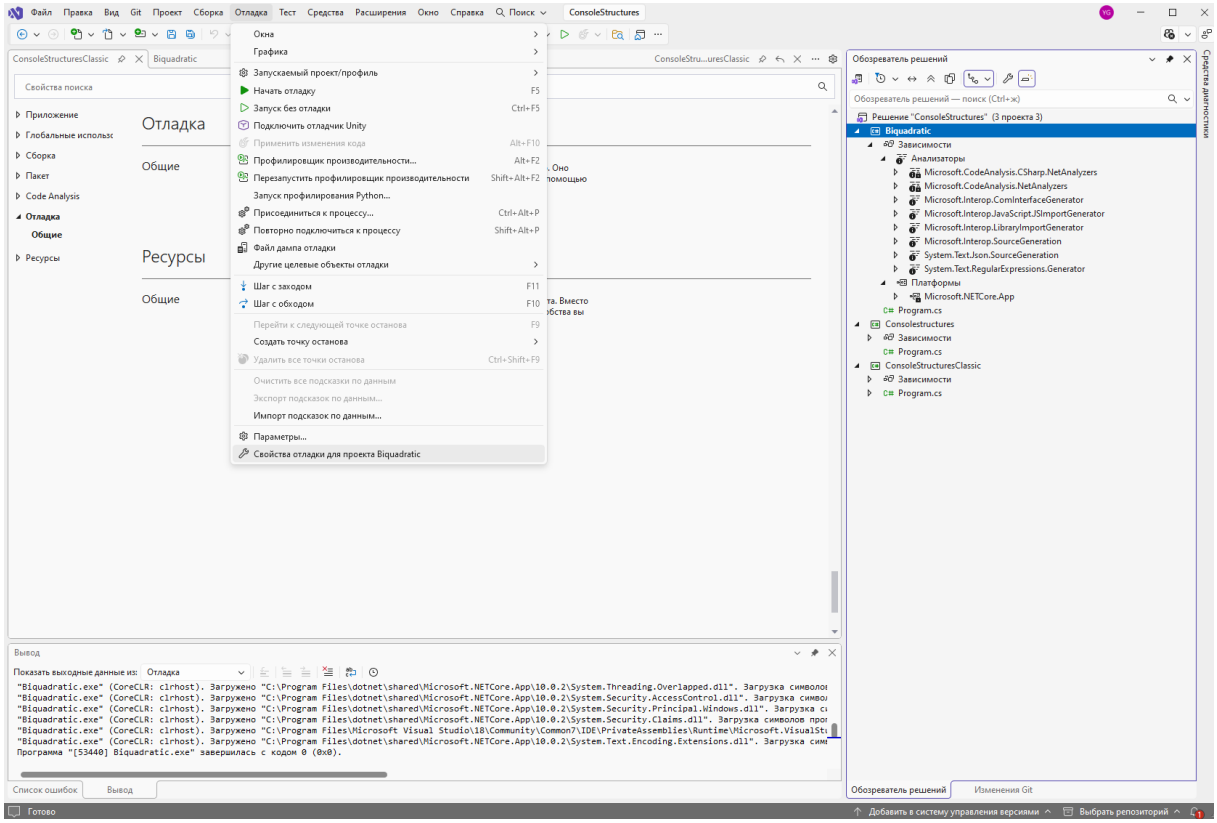


Рис. 13. Меню свойств отладки.

При выборе пункта меню открывается окно «Профили запуска», в котором есть поле для аргументов командной строки, их можно указать через пробел.

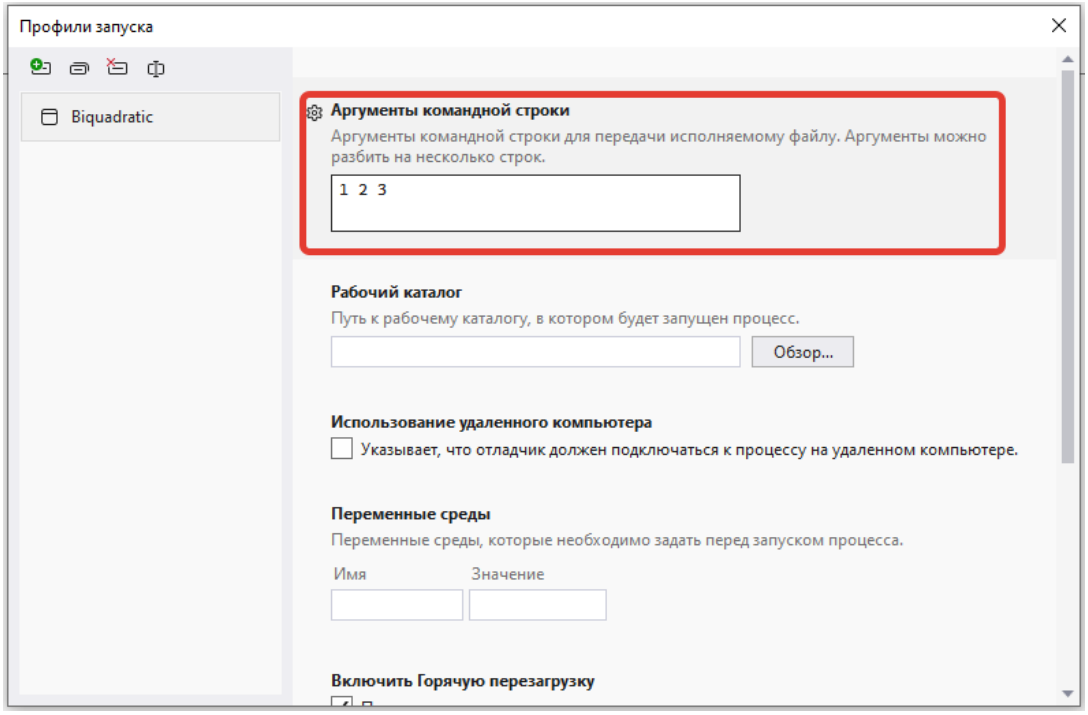


Рис. 14. Окно «Профили запуска».

Следующий фрагмент консольного приложения выводит параметры командной строки:

```
namespace ConsoleStructuresClassic
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Command line args:");
            foreach(string arg in args)
            {
                Console.WriteLine(arg);
            }
        }
    }
}
```

Результаты вывода в консоль:

```
Command line args:
1
2
3
```

Развернутый шаблон консольного приложения в настоящее время используется редко. В современных проектах C# используется более компактный код, можно выделить два варианта:

1. Начиная с C# 10 (.NET 6) используется подход file-scoped namespaces (пространства имен с областью действия на файл).

**Пример до C# 10:**

```
using System;
namespace Structures
{
    internal class FileName
    {
        // код
    }
}
```

**Пример начиная с C# 10:**

```
using System;

namespace Structures;

internal class FileName
{
    // код
}
```

2. Начиная с C# 9.0 (.NET 5) можно писать код основной консольной программы без объявления класса и пространства имен. Такой шаблон используется по умолчанию в современных версиях Visual Studio.

**Пример до C# 9:**

```
using System;

namespace MyApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
```

```
    }
}
```

**Пример начиная с C# 9:**

```
using System;

Console.WriteLine("Hello, World!");
```

### *5.3 XML-комментарии*

При объявлении классов, методов и других структур возможно задать комментарии к ним в виде XML-тэгов.

Такие комментарии сохраняются в откомпилированной сборке и используются Visual Studio для работы технологии дополнения кода IntelliSense.

*Пример метода с XML-комментариями:*

```
/// <summary>
/// Метод сложения двух целых чисел
/// </summary>
/// <param name="p1">Первое число</param>
/// <param name="p2">Второе число</param>
/// <returns>Результат сложения</returns>
static int Sum(int p1, int p2)
{
    return p1 + p2;
}
```

Перед XML-комментарием ставится три прямых слеша, далее указывается соответствующий тэг XML. Существует довольно большое количество XML-тэгов комментариев, но наиболее часто используются три из них:

- summary – краткое описание;
- param – описание входного параметра;
- returns – описание возвращаемого значения.

При работе в Visual Studio не нужно вводить полную структуру XML-комментариев. Чтобы добавить блок XML-комментариев, необходимо установить курсор на строку кода перед объявлением функции (класса и т.д.) и три раза набрать прямой слеш «/». После этого автоматически

добавляется заголовок XML-комментария. Visual Studio автоматически определяет, для какой структуры добавляется XML-комментарий, и генерирует набор тэгов, подходящих для данного случая.

*Пример автоматически сгенерированных XML-комментариев для метода:*

```
/// <summary>
///
/// </summary>
/// <param name="p1"></param>
/// <param name="p2"></param>
/// <returns></returns>
static int Sum(int p1, int p2)
{
    return p1 + p2;
}
```

Информация о наборе параметров при генерации комментария формируется автоматически, однако в случае изменения параметров после добавления XML-комментария невозможна повторная генерация комментария, информация о новых параметрах должна быть добавлена в XML-комментарий вручную.

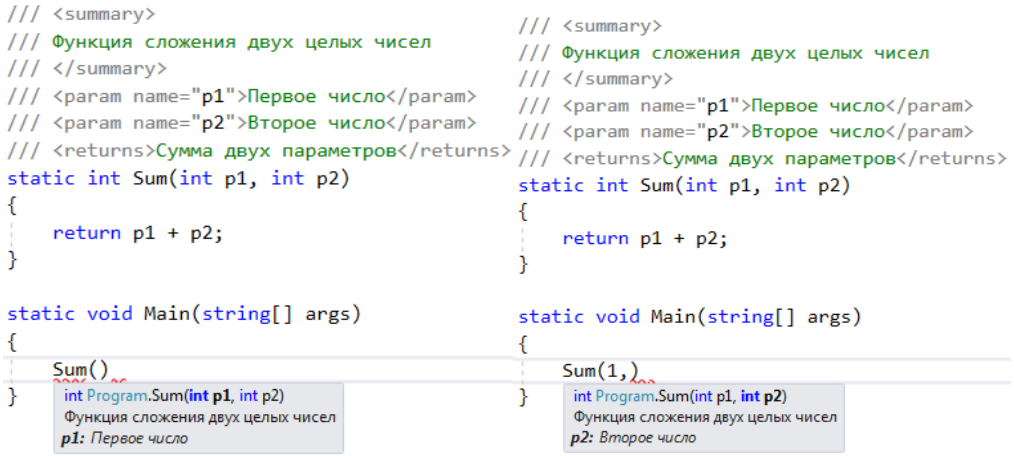


Рис. 15. Использование XML-комментариев при вызове функции Sum.

При вызове функции Sum на основе XML-комментариев будет автоматически сгенерирована подсказка, которая выводится с помощью IntelliSense (рис. 4).

Использование механизма XML-комментариев позволяет разработчику создавать хорошо документируемый код, при этом документация автоматически используется механизмом IntelliSense при

вызове кода. Поэтому использование XML-комментариев чрезвычайно желательно при разработке проектов.

### *5.4 Вызов методов, передача параметров и возврат значений*

Вызов методов происходит также как и в языках C++ и Java. Метод вызывается по имени, параметры передаются в круглых скобках после его имени. При объявлении метода указываются формальные параметры, при вызове в них выполняется подстановка фактических параметров, с которыми метод осуществляет работу при данном вызове.

В языке C# есть особенность, связанная с передачей параметров – ключевые слова ref и out.

Если перед параметром указывается ключевое слово ref, то значение передается по ссылке, то есть передается ссылка на параметр. Если параметр изменяется в методе, то эти изменения сохраняются в вызывающем методе.

При этом необходимо учитывать, что все значения ссылочных типов (объекты классов) автоматически передаются по ссылке даже без указания ключевого слова ref, их изменения в методе автоматически сохраняются.

Таким образом, ключевое слово ref актуально прежде всего для типов-значений. Оно является аналогом передачи параметра по указателю в языке C++, хотя в современных вариантах C++ также используется понятие ссылки.

Если перед параметром указывается ключевое слово out, то значение параметра обязательно должно быть инициализировано в вызываемом методе, что проверяется на этапе компиляции. До передачи в вызываемый метод значение переменной может быть не инициализировано. Инициализированное значение параметра становится доступно в вызывающем методе.

Ключевые слова ref и out должны быть указаны и при объявлении параметров в методе, и при вызове метода. Их указание при вызове метода, очевидно, излишне для компилятора, которому достаточно информации при объявлении метода. Однако это позволяет программисту избежать ошибок, связанных с незапланированным изменением значений параметров в методе.

*Пример вызова методов:*

```
string RefTest = «Значение до вызова функций»;

ParamByVal(RefTest);
Console.WriteLine(«\nВызов функции ParamByVal. Значение переменной:
« + RefTest);

ParamByRef(ref RefTest);
Console.WriteLine(«Вызов функции ParamByRef. Значение переменной: «
+ RefTest);

int x = 2, x2, x3;
ParamOut(x, out x2, out x3);
Console.WriteLine(«Вызов функции ParamOut. X={0}, x^2={1}, x^3={2}»,
x, x2, x3);
```

*Пример объявления методов:*

```
/// <summary>
/// Передача параметра по значению
/// </summary>
static void ParamByVal(string param)
{
    param = «Это значение НЕ будет передано в вызывающую функцию»;
}

/// <summary>
/// Передача параметра по ссылке
/// </summary>
static void ParamByRef(ref string param)
{
    param = «Это значение будет передано в вызывающую функцию»;
}

/// <summary>
/// Выходные параметры объявляются с помощью out
/// </summary>
static void ParamOut(int x, out int x2, out int x3)
{
    x2 = x * x;
```

```
    x3 = x * x * x;
}
```

В версии языка C# 7 реализовано синтаксическое усовершенствование, которое позволяет объявлять out-параметры непосредственно при вызове функции. Ранее для передачи out-параметров использовался следующий синтаксис:

```
int i;
OutFunction(out i);
```

Теперь можно использовать упрощенную конструкцию:

```
OutFunction(out int i);
```

Использование упрощенной конструкции позволяет существенно сократить код при большом количестве out-параметров.

Если какой-то out-параметр не используется, то его можно заменить на конструкцию discard (_):

```
OutFunction(out int i, out string j, out float j);
// результаты параметров j и k не нужны при вызове
OutFunction(out int i, out string _, out float _);
```

В метод может быть передано переменное количество параметров, для этого при объявлении параметра метода используется ключевое слово params.

*Пример вызова метода с переменным количеством параметров:*

```
ParamArray("Вывод параметров: ", 1, 2, 333);
```

*Пример объявления метода с переменным количеством параметров:*

```
/// <summary>
/// Переменное количество параметров задается с помощью params
/// </summary>
static void ParamArray(string str, params int[] ArrayParams)
{
    Console.Write(str);
    foreach (int i in ArrayParams)
        Console.Write(" {0} ", i);
}
```

Параметр с ключевым словом params должен быть объявлен последним в списке параметров.

Для возврата значений из методов, так же как и в языках C++ и Java, используется ключевое слово return.

## 5.5 Условные операторы и операторы сопоставления с образцом

*Пример условного оператора if:*

```
if (str == "строка1")
{
    Console.WriteLine("if: str == \"строка1\"");
}
else
{
    Console.WriteLine("if: str != \"строка1\"");
}
```

В качестве условия проверяется значение логического типа, который отсутствует в стандартном C++, но есть в Паскале, Java, C#.

Условный оператор вопрос-двоеточие выполняется аналогично тому, что в C++ и Java. До знака вопроса указывается логическое выражение. Если оно истинно, то выполняется код от вопроса до двоеточия, если ложно – код после двоеточия.

*Пример оператора вопрос-двоеточие:*

```
string Result = (str == "строка1" ? "Да" : "Нет");
```

Оператор switch выполняется аналогично тем, что в C++ и Java.

*Пример оператора switch:*

```
switch (str)
{
    case "строка1":
        Result2 = "строка1";
        break;

    case "строка2":
    case "строка3":
        Result2 = "строка2 или строка3";
        break;

    default:
        Result2 = "другая строка";
        break;
}
```

В круглых скобках после switch указывается проверяемое выражение, а после case – возможные варианты проверяемых значений.

Если значения после switch и case совпадают, то выполняются операторы, указанные в case до первого оператора break. Если оператор break не указан, то выполняются операторы следующей по порядку секции case, поэтому обычно каждую секцию case завершает оператор break. Если ни один оператор case не удовлетворяет условию, то выполняются операторы секции default.

Операторы сопоставления с образцом (pattern matching) – это классическая особенность языков программирования, использующих функциональный подход, таких как F#, Scala, Erlang, Haskell.

В языке C# данная возможность появилась, начиная с версии 7. Сопоставление с образцом похоже на условные операторы и поэтому построено на их основе.

Пусть дан массив объектов, который содержит строки и целые числа:

```
object[] array1 = { 1, "строка 1", 2, "строка 2", 3 };
```

Для того чтобы расширить условный оператор до оператора сопоставления с образцом в язык C# была добавлена конструкция is.

*Пример сопоставления с образцом на основе условного оператора if:*

```
foreach(object obj in array1)
{
    if(obj is int i1)
    {
        Console.WriteLine("Число -> " + i1.ToString());
    }
    else if (obj is string s1)
    {
        Console.WriteLine("Строка -> " + s1);
    }
}
```

В данном примере в цикле foreach по очереди перебираются элементы массива.

Если элемент массива соответствует целому числу «obj is int i1», то он автоматически приводится к целому типу и помещается в переменную i1. Далее с ним можно выполнять необходимые действия, в примере это вывод в консоль.

Если элемент массива соответствует строке «obj is string s1», то он автоматически приводится к строке и помещается в переменную s1.

Результаты вывода в консоль:

```
Число -> 1
Строка -> строка 1
Число -> 2
Строка -> строка 2
Число -> 3
```

Вторым вариантом сопоставления с образцом в C# является использование оператора switch.

*Пример сопоставления с образцом на основе оператора switch:*

```
foreach (object obj in array1)
{
    switch(obj)
    {
        case string s1:
            Console.WriteLine("Строка -> " + s1);
            break;
        case int i1 when i1 > 2:
            Console.WriteLine("Число большее 2 -> " +
i1.ToString());
            break;
        case int i1:
            Console.WriteLine("Число -> " + i1.ToString());
            break;
    }
}
```

В этом случае в каждом операторе case указывается проверяемый тип и переменная, в которую сохраняется результат приведения типа.

С использованием ключевого слова when можно задавать так называемые «охранные выражения» (guards), которые накладывают дополнительные ограничения на проверку. В данном примере возможность приведения к целому типу проверяется дважды, случай i1>2 рассматривается отдельно и возникновение такого случая проверяется с помощью охранного выражения.

Результаты вывода в консоль:

```
Число -> 1
```

```
Строка -> строка 1
Число -> 2
Строка -> строка 2
Число большее 2 -> 3
```

Современный стиль сопоставления с образцом это switch expression (конструкция появилась в C# 8). Данная конструкция очень напоминает классические конструкции pattern matching из функциональных языков программирования.

```
// Современный стиль (switch expression)
// появился в C# 8
string result21 = str switch
{
    "строка1" => "строка1",
    "строка2" or "строка3" => "строка2 или строка3",  // логический
or
    _ => "другая строка"  // _ = default
};

// Более сложный пример с типами
int? obj = 3;
string description = obj switch
{
    int i => $"Целое число: {i}",
    null => "Null",
};

// С условиями (guards)
var number = 5;
string category = number switch
{
    < 0 => "Отрицательное",
    0 => "Ноль",
    > 0 and <= 10 => "Малое положительное",
    > 10 => "Большое положительное"
};
Console.WriteLine($"{result21} {obj} {number}");
```

Результаты вывода в консоль:

```
строка1 3 5
```

## ЗАДАНИЕ

**Дан массив:**

```
object[] data = { 15, "Hello", -5, null,
                  "World", 42, 0, "C#", -10 };
```

**Напишите программу на C#, которая анализирует массив объектов различных типов. Требуется реализовать 3 метода:**

- **Метод AnalyzeWithIf - используя условный оператор if и конструкцию is:**
  1. **Для положительных чисел выводить: "Положительное число: {значение}"**
  2. **Для нуля выводить: "Ноль: {значение}"**
  3. **Для отрицательных чисел выводить: "Отрицательное число: {значение}"**
  4. **Для строк выводить: "Строка: {значение}"**
  5. **Для null выводить: "Пустое значение"**
- **Метод AnalyzeWithSwitch - используя традиционный оператор switch с pattern matching и guard условиями (when)**
- **Метод GetCategory - используя современный switch expression, который возвращает строку с категорией элемента**

**Ожидаемый вывод программы должен показывать результаты работы всех трёх методов.**

РЕШЕНИЕ:

```
static void example_conditions()
{
    object[] data = { 15, "Hello", -5, null, "World", 42, 0, "C#", -10 };

    Console.WriteLine("=== Анализ с помощью if ===");
    AnalyzeWithIf(data);

    Console.WriteLine("\n=== Анализ с помощью switch ===");
    AnalyzeWithSwitch(data);

    Console.WriteLine("\n=== Категории (switch expression) ===");
    foreach (var item in data)
    {
        string category = GetCategory(item);
        Console.WriteLine($"{item ?? "null"} -> {category}");
    }

    // В решении используются вложенные методы (функции)
    // Метод 1: Использование if с pattern matching
    static void AnalyzeWithIf(object[] array)
    {
        foreach (object obj in array)
        {
            if (obj is null)
            {
                Console.WriteLine("Пустое значение");
            }
            else if (obj is int number && number > 0)
            {
                Console.WriteLine($"Положительное число: {number}");
            }
            else if (obj is int num && num == 0)
            {
                Console.WriteLine($"Ноль: {num}");
            }
            else if (obj is int negative && negative < 0)
            {
                Console.WriteLine($"Отрицательное число: {negative}");
            }
            else if (obj is string str)
            {
                Console.WriteLine($"Строка: {str}");
            }
        }
    }

    // Метод 2: Использование switch с pattern matching и guards
    static void AnalyzeWithSwitch(object[] array)
    {
        foreach (object obj in array)
        {
            switch (obj)
            {
                case null:
                    Console.WriteLine("Пустое значение");
                    break;

                case int i when i > 0:
                    Console.WriteLine($"Положительное число: {i}");
                    break;

                case int i when i == 0:
```

```
                    Console.WriteLine($"Ноль: {i}");
                    break;

                case int i when i < 0:
                    Console.WriteLine($"Отрицательное число: {i}");
                    break;

                case string s:
                    Console.WriteLine($"Строка: {s}");
                    break;
            }
        }
    }

    // Метод 3: Использование switch expression
    static string GetCategory(object obj)
    {
        return obj switch
        {
            null => "NULL",
            int i when i > 0 => "ПОЛОЖИТЕЛЬНОЕ",
            int i when i == 0 => "НОЛЬ",
            int i when i < 0 => "ОТРИЦАТЕЛЬНОЕ",
            string s => $"СТРОКА_ДЛИНЫ_{s.Length}",
            _ => "НЕИЗВЕСТНЫЙ_ТИП"
        };
    }
}
```

### 5.6 Операторы цикла

Счетный цикл for такой же как в C++ и Java. В круглых скобках после for указываются три оператора – начальное присвоение значения переменной цикла; условие выхода из цикла; оператор, который выполняется при переходе к следующему шагу цикла. Операторы разделяются точкой с запятой.

*Пример цикла for:*

```
for (int i = 0; i < 3; i++)
    Console.Write(i);
```

В данном примере оператор вывода переменной i не вложен в фигурные скобки, использование скобок не является обязательным, так как это единственный оператор цикла. Если бы в цикле выполнялось несколько действий, то их необходимо было бы вложить в фигурные скобки.

В языке C# также существует форма счетного цикла, предназначенная для перебора коллекции – foreach. В классическом языке C++ такой цикл

отсутствует (однако он появился в новых версиях). В языке Java этот цикл имеет синтаксис for(переменная : коллекция).

*Пример цикла foreach:*

```
int[] array1 = { 1, 2, 3 };
foreach(int i2 in array1)
    Console.Write(i2);
```

В данном примере int i2 является объявлением переменной цикла, in отделяет переменную цикла от имени коллекции.

Следует отметить, что цикл foreach является одним из наиболее часто используемых видов цикла в C#. Это обусловлено тем, что в языке C# существует развитый набор коллекций, а реализация многих алгоритмов связана с перебором коллекций.

Циклы с предусловием и постусловием также очень похожи на те, что в C++ и Java.

*Пример цикла с предусловием:*

```
int i3 = 0;
while (i3 < 3)
{
    Console.Write(i3);
    i3++;
}
```

*Пример цикла с постусловием:*

```
int i4 = 0;
do
{
    Console.Write(i4);
    i4++;
} while (i4 < 3);
```

Операторы break и continue используются аналогично языкам C и C++.

## ЗАДАНИЕ

**Дан массив:**

```
int[] numbers = { 5, -3, 8, 12, -7,
                  0, 15, -2, 4, 20 };
```

**Напишите программу на C#, которая работает с массивом целых чисел и выполняет различные операции, используя разные типы циклов. Требуется реализовать 5 методов:**

1. **Метод PrintEvenIndices - используя цикл for, вывести элементы массива, находящиеся на четных индексах (0, 2, 4, ...)**
2. **Метод CalculateSum - используя цикл foreach, вычислить и вернуть сумму всех элементов массива**
3. **Метод FindFirstGreaterThan - используя цикл while и оператор break, найти и вернуть первое число больше заданного значения (параметр метода). Если такого числа нет, вернуть -1**
4. **Метод PrintUntilSumExceeds - используя цикл do-while, выводить элементы массива и их накопленную сумму, пока сумма не превысит заданное значение (параметр метода)**
5. **Метод PrintPositiveOnly - используя цикл foreach и оператор continue, вывести только положительные числа из массива (пропуская отрицательные и ноль)**

**Программа должна вызвать все методы и продемонстрировать их работу.**

РЕШЕНИЕ с использованием локальных функций (появились в C# 7):

```
static void example_loops()
{
    int[] numbers = { 5, -3, 8, 12, -7, 0, 15, -2, 4, 20 };

    Console.WriteLine("Исходный массив:");
    Console.WriteLine(string.Join(", ", numbers));
    Console.WriteLine();

    // 1. Вывод элементов на четных индексах (for)
    Console.WriteLine("=== 1. Элементы на четных индексах (цикл for)
===");
    PrintEvenIndices(numbers);
    Console.WriteLine();

    // 2. Сумма всех элементов (foreach)
    Console.WriteLine("=== 2. Сумма всех элементов (цикл foreach)
===");
    int sum = CalculateSum(numbers);
    Console.WriteLine($"Сумма: {sum}");
    Console.WriteLine();

    // 3. Первое число больше заданного (while с break)
    Console.WriteLine("=== 3. Первое число > 10 (цикл while с break)
===");
    int result = FindFirstGreaterThan(numbers, 10);
    if (result != -1)
        Console.WriteLine($"Найдено: {result}");
    else
        Console.WriteLine("Не найдено");
    Console.WriteLine();

    // 4. Вывод пока сумма не превысит значение (do-while)
    Console.WriteLine("=== 4. Вывод пока сумма <= 15 (цикл do-while)
===");
    PrintUntilSumExceeds(numbers, 15);
    Console.WriteLine();

    // 5. Только положительные числа (foreach с continue)
    Console.WriteLine("=== 5. Только положительные (foreach с
continue) ===");
    PrintPositiveOnly(numbers);

    // Метод 1: Цикл for - элементы на четных индексах
    static void PrintEvenIndices(int[] array)
    {
        Console.Write("Элементы на индексах 0, 2, 4, ...: ");
        for (int i = 0; i < array.Length; i += 2)
        {
```

```
            Console.Write(array[i] + " ");
        }
        Console.WriteLine();
    }

    // Метод 2: Цикл foreach - сумма элементов
    static int CalculateSum(int[] array)
    {
        int sum = 0;
        foreach (int num in array)
        {
            sum += num;
        }
        return sum;
    }

    // Метод 3: Цикл while с break - поиск первого элемента
    static int FindFirstGreaterThan(int[] array, int threshold)
    {
        int index = 0;
        while (index < array.Length)
        {
            if (array[index] > threshold)
            {
                return array[index];  // или можно использовать
break
            }
            index++;
        }
        return -1;  // не найдено
    }

    // Метод 4: Цикл do-while - вывод до превышения суммы
    static void PrintUntilSumExceeds(int[] array, int limit)
    {
        int index = 0;
        int currentSum = 0;

        do
        {
            currentSum += array[index];
            Console.WriteLine($"array[{index}] = {array[index]},
накопленная сумма = {currentSum}");
            index++;

            if (currentSum > limit)
            {
                Console.WriteLine($"Сумма превысила {limit}, выход
из цикла");
                break;
```

```
            }

        } while (index < array.Length);
    }

    // Метод 5: Цикл foreach с continue - только положительные
    static void PrintPositiveOnly(int[] array)
    {
        Console.Write("Положительные числа: ");
        foreach (int num in array)
        {
            if (num <= 0)
            {
                continue;  // пропускаем отрицательные и ноль
            }
            Console.Write(num + " ");
        }
        Console.WriteLine();
    }
}
```

Результаты вывода в консоль:

```
Исходный массив:
5, -3, 8, 12, -7, 0, 15, -2, 4, 20

=== 1. Элементы на четных индексах (цикл for) ===
Элементы на индексах 0, 2, 4, ...: 5 8 -7 15 4

=== 2. Сумма всех элементов (цикл foreach) ===
Сумма: 52

=== 3. Первое число > 10 (цикл while с break) ===
Найдено: 12

=== 4. Вывод пока сумма <= 15 (цикл do-while) ===
array[0] = 5, накопленная сумма = 5
array[1] = -3, накопленная сумма = 2
array[2] = 8, накопленная сумма = 10
array[3] = 12, накопленная сумма = 22
Сумма превысила 15, выход из цикла
```

```
=== 5. Только положительные (foreach с continue) ===
Положительные числа: 5 8 12 15 4 20
```

### 5.7 Использование массивов

Для объявления массивов в C# используется синтаксис:

```
int[] array;
```

В языке C++ квадратные скобки должны ставить не после имени типа, а после имени переменной: int array[]. В Java допустимы оба варианта объявления.

Если необходимо присвоить начальные значения элементам массива, то форма объявления следующая:

```
int[] array = new int[3] {1, 2, 3};
```

Аналогично для строкового типа:

```
string[] strs =
        new string[3] { "строка1", "строка2", "строка3" };
```

Многомерные массивы в языке C# бывают двух видов – прямоугольные и зубчатые (jagged).

Объявление прямоугольных массивов очень похоже на их объявление в Паскале.

*Пример объявления прямоугольного массива:*

```
int[,] matrix = new int[2, 2] { { 1, 2 }, { 3, 4 } };
```

*или:*

```
int[,] matrix = { {1,2}, {3,4} };
```

*или:*

```
int[,] matrix = new int[2, 2];
matrix[0, 0] = 1;
matrix[0, 1] = 2;
matrix[1, 0] = 3;
matrix[1, 1] = 4;
```

Объявление зубчатых массивов похоже на их объявление в языках C++ и Java.

*Пример объявления зубчатого массива:*

```
int[][] jagged = new int[3][];
jagged[0] = new int[3] { 1, 2, 3 };
jagged[1] = new int[2] { 4, 5 };
jagged[2] = new int[4] { 6, 7, 8, 9 };
```

В случае зубчатого массива каждый подмассив может иметь собственную размерность.

## ЗАДАНИЕ

**Для квадратной матрицы напишите функции проверки ее симметричности относительно главной диагонали.**

**Матрица называется симметричной относительно главной диагонали, если элемент на позиции [i, j] равен элементу на позиции [j, i] для всех i и j.**

**Например, матрица:**

**1 2 3**

**2 5 6**

**3 6 9**

**является симметричной, так как:**

- **matrix[0,1] = 2 равно matrix[1,0] = 2**
- **matrix[0,2] = 3 равно matrix[2,0] = 3**
- **matrix[1,2] = 6 равно matrix[2,1] = 6**

**Требуется реализовать:**

- **Метод IsSymmetric - принимает прямоугольный двумерный массив и возвращает true, если матрица симметрична, и false в противном**

случае. Если матрица не квадратная, метод должен вернуть false.

- Метод PrintMatrix - выводит матрицу в форматированном виде

В основном методе создать и проверить:

- Симметричную матрицу 3x3
- Несимметричную матрицу 3x3
- Неквадратную матрицу 2x3

Программа должна вывести каждую матрицу и результат проверки её симметричности.

РЕШЕНИЕ:

```
static void example_matrices()
{
    // 1. Симметричная матрица
    Console.WriteLine("=== Матрица 1: Симметричная ===");
    int[,] matrix1 = {
    { 1, 2, 3 },
    { 2, 5, 6 },
    { 3, 6, 9 }
    };
    PrintMatrix(matrix1);
    bool result1 = IsSymmetric(matrix1);
    Console.WriteLine($"Симметрична: {(result1 ? "Да" : "Нет")}");
    Console.WriteLine();

    // 2. Несимметричная матрица
    Console.WriteLine("=== Матрица 2: Несимметричная ===");
    int[,] matrix2 = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
    };
    PrintMatrix(matrix2);
    bool result2 = IsSymmetric(matrix2);
    Console.WriteLine($"Симметрична: {(result2 ? "Да" : "Нет")}");
    Console.WriteLine();

    // 3. Неквадратная матрица
    Console.WriteLine("=== Матрица 3: Неквадратная (2x3) ===");
    int[,] matrix3 = {
    { 1, 2, 3 },
    { 4, 5, 6 }
    };
    PrintMatrix(matrix3);
    bool result3 = IsSymmetric(matrix3);
    Console.WriteLine($"Симметрична: {(result3 ? "Да" : "Нет")}");
    Console.WriteLine();

    // 4. Единичная матрица
    Console.WriteLine("=== Матрица 4: Единичная матрица 4x4 ===");
    int[,] matrix4 = {
    { 1, 0, 0, 0 },
    { 0, 1, 0, 0 },
    { 0, 0, 1, 0 },
    { 0, 0, 0, 1 }
    };
    PrintMatrix(matrix4);
    bool result4 = IsSymmetric(matrix4);
    Console.WriteLine($"Симметрична: {(result4 ? "Да" : "Нет")}");
```

```
    // Метод проверки симметричности матрицы
    static bool IsSymmetric(int[,] matrix)
    {
        int rows = matrix.GetLength(0);  // количество строк
        int cols = matrix.GetLength(1);  // количество столбцов

        // Проверка: матрица должна быть квадратной
        if (rows != cols)
        {
            return false;
        }

        // Проверка симметричности
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                // Проверяем, что элемент [i,j] равен элементу [j,i]
                if (matrix[i, j] != matrix[j, i])
                {
                    return false;
                }
            }
        }

        return true;
    }

    // Метод вывода матрицы
    static void PrintMatrix(int[,] matrix)
    {
        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);

        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                Console.Write($"{matrix[i, j],4}");
            }
            Console.WriteLine();
        }
    }
}
```

### 5.8 Работа с null

```
//Ошибка, типу int нельзя присвоить null
int nl = null;
```

Nullable value types используют структуру Nullable<T>. Тип int? означает, что переменной можно присвоить или значение типа int или null.

```
int? nl_number = 42;
int? nl_nullNumber = null;

if (nl_number.HasValue)
{
    //присваивается значение int (42)
    int nl_value = nl_number.Value;
    Console.WriteLine(nl_value);
}
```

Null-coalescing оператор ?? появился в C# 2. Оператор ?? присваивает правое значение если левое null.

```
int nl_result1 = nl_number ?? -1; // 42
int nl_result2 = nl_nullNumber ?? -1; // -1
Console.WriteLine($"{nl_number} {nl_nullNumber} {nl_result1}
{nl_result2}");
```

Nullable reference types — это ссылочные типы (например, string), которые явно помечены как допускающие null.

```
string? nl_str1 = "Строка1";
string? nl_str2 = null;
if(nl_str2 == null) Console.WriteLine(nl_str1);
```

Null-conditional оператор ?. появился в C# 6. Если nl_str2 == null, то возвращается null, ошибка не возникает.

```
int ? nl_length1 = nl_str1?.Length;
int? nl_length2 = nl_str2?.Length;
Console.WriteLine($"{nl_length1} {nl_length2}");
```

Null-coalescing assignment (присваивание) ??= появилось начиная с C#8. Присваивание сработает, только если nl_str2 == null.

```
nl_str2 ??= "!!!";
Console.WriteLine(nl_str2);
```

Для проверки на null можно использовать выражения

```
// старый стиль
if (obj == null) { }
// современный стиль начиная с C# 7
if (obj is null) { }
if (obj is not null) { }
```

## ЗАДАНИЕ

**Напишите программу на C#, которая демонстрирует различные способы работы с null-значениями на примере массива nullable целых чисел.**

```
int?[] numbers = { 10, null, 25, null,
                   30, 15, null };
```

1. **Вывести исходный массив (для null выводить слово "null")**
2. **Подсчитать количество null-значений, используя проверку is null**
3. **Подсчитать количество не-null значений, используя проверку is not null**
4. **Вычислить сумму всех ненулевых чисел, используя свойство HasValue и Value**
5. **Создать новый массив, где все null заменены на -1, используя оператор ??**
6. **Вывести длину строкового представления каждого числа (используя оператор ?. для вызова метода ToString())**
7. **Заменить все null-значения в исходном массиве на 0, используя оператор ??=**
8. **Вывести итоговый массив после замены null-значений**

РЕШЕНИЕ:

```
static void example_null()
{
    // Исходные данные
    int?[] numbers = { 10, null, 25, null, 30, 15, null };

    // 1. Вывод исходного массива
    Console.WriteLine("=== 1. Исходный массив ===");
    for (int i = 0; i < numbers.Length; i++)
    {
        // Используем ?. для безопасного вызова ToString()
        string display = numbers[i]?.ToString() ?? "null";
        Console.WriteLine($"numbers[{i}] = {display}");
    }
    Console.WriteLine();

    // 2. Подсчет null-значений (оператор is null)
    Console.WriteLine("=== 2. Подсчет null-значений (is null) ===");
    int nullCount = 0;
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] is null)
        {
            nullCount++;
        }
    }
    Console.WriteLine($"Количество null-значений: {nullCount}");
    Console.WriteLine();

    // 3. Подсчет не-null значений (оператор is not null)
    Console.WriteLine("=== 3. Подсчет не-null значений (is not null)
===");
    int notNullCount = 0;
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] is not null)
        {
            notNullCount++;
        }
    }
    Console.WriteLine($"Количество не-null значений: {notNullCount}");
    Console.WriteLine();

    // 4. Сумма ненулевых чисел (HasValue и Value)
    Console.WriteLine("=== 4. Сумма ненулевых чисел (HasValue и Value)
===");
    int sum = 0;
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i].HasValue)
        {
            sum += numbers[i].Value;
            Console.WriteLine($"Добавляем numbers[{i}] =
{numbers[i].Value}, текущая сумма = {sum}");
```

```
        }
    }
    Console.WriteLine($"Итоговая сумма: {sum}");
    Console.WriteLine();

    // 5. Замена null на -1 (оператор ??)
    Console.WriteLine("=== 5. Создание массива с заменой null на -1
(оператор ??) ===");
    int[] numbersWithDefault = new int[numbers.Length];
    for (int i = 0; i < numbers.Length; i++)
    {
        numbersWithDefault[i] = numbers[i] ?? -1;
        Console.WriteLine($"numbersWithDefault[{i}] =
{numbersWithDefault[i]}");
    }
    Console.WriteLine();

    // 6. Длина строкового представления (оператор ?.)
    Console.WriteLine("=== 6. Длина строкового представления (оператор ?.)
===");
    for (int i = 0; i < numbers.Length; i++)
    {
        // numbers[i]?.ToString() вернет null если numbers[i] == null
        // затем ?.Length вернет null если ToString() вернул null
        // затем ?? 0 заменит null на 0
        int? length = numbers[i]?.ToString()?.Length ?? 0;
        Console.WriteLine($"Длина представления numbers[{i}] = {length}");
    }
    Console.WriteLine();

    // 7. Замена null на 0 в исходном массиве (оператор ??=)
    Console.WriteLine("=== 7. Замена null на 0 в исходном массиве
(оператор ??=) ===");
    Console.WriteLine("До замены: null-значений = " + nullCount);
    for (int i = 0; i < numbers.Length; i++)
    {
        // Присваивание произойдет только если numbers[i] == null
        numbers[i] ??= 0;
    }

    // Проверка что null-значений больше нет
    int nullCountAfter = 0;
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] is null)
        {
            nullCountAfter++;
        }
    }
    Console.WriteLine("После замены: null-значений = " + nullCountAfter);
    Console.WriteLine();

    // 8. Вывод итогового массива
    Console.WriteLine("=== 8. Итоговый массив после замены ===");
    for (int i = 0; i < numbers.Length; i++)
```

```
    {
        Console.WriteLine($"numbers[{i}] = {numbers[i]}");
    }
    Console.WriteLine();

    // Дополнительная демонстрация
    Console.WriteLine("=== ДОПОЛНИТЕЛЬНО: Сравнение способов проверки
===");
    int? testValue = null;

    // Старый стиль
    if (testValue == null)
    {
        Console.WriteLine("testValue == null (старый стиль): true");
    }

    // Современный стиль
    if (testValue is null)
    {
        Console.WriteLine("testValue is null (современный стиль): true");
    }

    testValue = 42;
    if (testValue is not null)
    {
        Console.WriteLine($"testValue is not null: true, значение =
{testValue}");
    }
}
```