

Министерство науки и высшего образования РФ  
ФГАОУ ВПО  
Национальный исследовательский технологический университет «МИСиС»

---

Институт Информационных технологий и компьютерных наук (ИТКН)

Кафедра Инфокоммуникационных технологий (ИКТ)

**Отчет по контрольной работе №2**  
по дисциплине «Программирование и Алгоритмизация»

Выполнил:  
студент группы БИВТ-24-5

Черных Богдан

Проверил:  
Стучилин В. В.

Москва, 2024

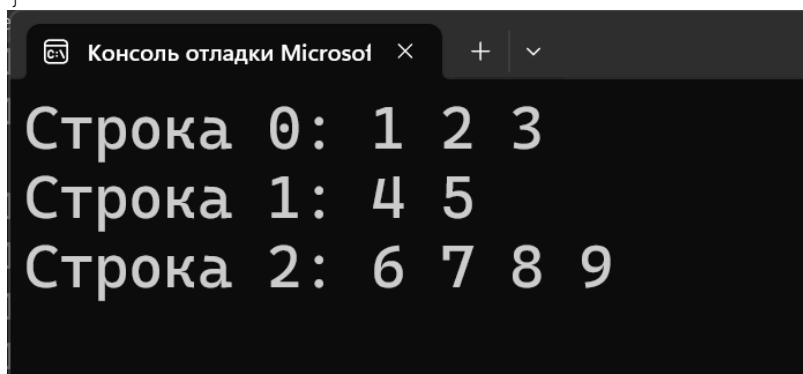
## ВАРИАНТ 5

### Задание №1.

**Ступенчатые массивы (jagged arrays)** в C# — это массивы массивов, где каждый элемент основного массива является отдельным массивом. Особенность ступенчатых массивов заключается в том, что вложенные массивы могут иметь разную длину, что позволяет гибко управлять количеством элементов в каждой "строке" массива.

Пример:

```
using System; //Нужно для корректной работы кода
class Program // Создаем программу
{
    static void Main() // функция Main()
    {
        // Создаем ступенчатый массив, где каждая строка может иметь разное
        // количество элементов
        int[][] jaggedArray = new int[3][];
        // Инициализируем первую строку массива с тремя элементами
        jaggedArray[0] = new int[] { 1, 2, 3 };
        // Инициализируем вторую строку массива с двумя элементами
        jaggedArray[1] = new int[] { 4, 5 };
        // Инициализируем третью строку массива с четырьмя элементами
        jaggedArray[2] = new int[] { 6, 7, 8, 9 };
        // Проходим по всем строкам массива
        for (int i = 0; i < jaggedArray.Length; i++)
        {
            // Для каждой строки выводим её элементы
            Console.Write($"Строка {i}: ");
            for (int j = 0; j < jaggedArray[i].Length; j++)
            {
                // Выводим каждый элемент строки
                Console.Write(jaggedArray[i][j] + " ");
            }
            // Переход на новую строку после завершения вывода элементов строки
            Console.WriteLine();
        }
    }
}
```



### Задание №2.

**Исключения** — это ситуации, которые нарушают нормальное выполнение программы. Исключения представляют собой ошибки, возникающие во

время выполнения программы, такие как деление на ноль, обращение к несуществующему элементу массива и другие.

В C# исключения представляют собой объекты, которые наследуются от класса `System.Exception`. Их обработка позволяет программе не завершаться аварийно, а продолжить работу.

Основные типы исключений:

`System.DivideByZeroException` — деление на ноль.

`System.IndexOutOfRangeException` — выход за пределы массива.

`System.NullReferenceException` — обращение к объекту, равному `null`.

`System.InvalidOperationException` — недопустимая операция.

`System.IO.IOException` — ошибки ввода-вывода.

`System.ArgumentException` — неправильный аргумент метода.

`System.ArgumentNullException` - возникает, если в метод передан `null`, а метод не принимает значения `null`.

`System.OutOfMemoryException` - возникает, если программе не хватает памяти для выполнения операции.

Исключения обрабатываются с помощью конструкции `try-catch-finally`:

`try` — блок кода, в котором может возникнуть исключение.

`catch` — блок для обработки исключения.

`finally` (опционально) — блок, который выполняется всегда, независимо от того, возникло исключение или нет.

Пример обработки нескольких типов исключений:

```
using System; // Подключение стандартного пространства имен

class Program // Создаем программу
{
    static void Main() // функция Main()
    {
        try // Проверяем на исключения
        {
            Console.WriteLine("Введите первое число:"); // Запрос первого числа
            int num1 = int.Parse(Console.ReadLine()); // Преобразование ввода в
число

            Console.WriteLine("Введите второе число:"); // Запрос второго числа
            int num2 = int.Parse(Console.ReadLine()); // Преобразование ввода в
число

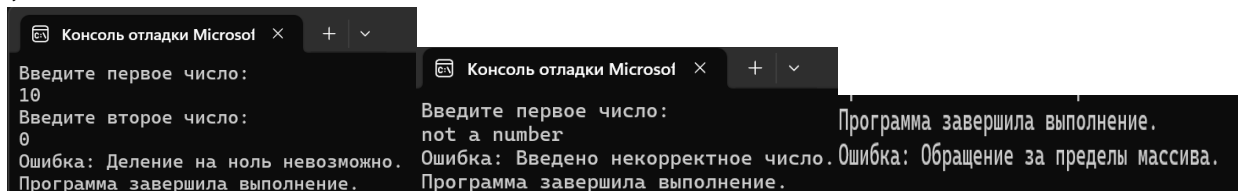
            // Попытка деления чисел
```

```

        int result = num1 / num2; // Деление чисел
        Console.WriteLine($"Результат деления: {result}"); // Вывод результата
    }
    catch (DivideByZeroException ex) // Обработка деления на ноль
    {
        Console.WriteLine("Ошибка: Деление на ноль невозможно."); // Сообщение
        об ошибке
    }
    catch (FormatException ex) // Обработка ошибки формата ввода
    {
        Console.WriteLine("Ошибка: Введено некорректное число."); // Сообщение
        об ошибке
    }
    catch (Exception ex) // Общая обработка всех остальных исключений
    {
        Console.WriteLine($"Произошла ошибка: {ex.Message}"); // Вывод общего
        сообщения об ошибке
    }
    finally
    {
        Console.WriteLine("Программа завершила выполнение."); // Финальное
        сообщение
    }

    // Второй пример — выход за пределы массива
    try
    {
        int[] numbers = { 1, 2, 3 }; // Объявление массива
        Console.WriteLine(numbers[5]); // Попытка обращения к несуществующему
        элементу
    }
    catch (IndexOutOfRangeException ex) // Обработка выхода за пределы массива
    {
        Console.WriteLine("Ошибка: Обращение за пределы массива."); //
        Сообщение об ошибке
    }
}
}

```



## Задание №3.

**Чтение данных из файла** — это одна из ключевых операций в программировании, позволяющая получать и обрабатывать сохранённую информацию. В C# работа с файлами осуществляется через пространство имён System.IO, которое предоставляет классы и методы для взаимодействия с файловой системой. В C# существует несколько способов чтения данных из файла. Чтение может быть выполнено с помощью методов:

**File.ReadAllText** — чтение всего содержимого файла в строку.

**File.ReadAllLines** — чтение файла построчно в массив строк.

**StreamReader** — поэтапное чтение файла.

Примеры программ для чтения данных файла:

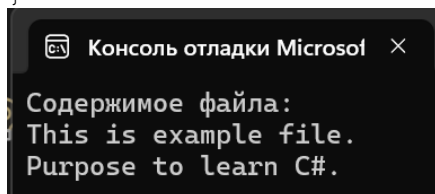
### 1) Чтение всего файла целиком

Используется, когда нужно загрузить всё содержимое файла в память.

Подходит для небольших файлов, так как потребляет значительное количество памяти при больших объёмах данных. Методы, такие как `File.ReadAllText` и `File.ReadAllLines`, удобны и просты в использовании.

```
using System; // Подключение стандартного пространства имен
using System.IO; // Подключение для работы с файлами

class Program // Создаем программу
{
    static void Main() // функция Main()
    {
        try
        {
            // Читаем весь файл как одну строку
            string content = File.ReadAllText("example.txt"); // Чтение
            содержимого файла
            Console.WriteLine("Содержимое файла:\n" + content); // Вывод
            содержимого на экран
        }
        catch (FileNotFoundException) // Обработка ошибки, если файл не найден
        {
            Console.WriteLine("Ошибка: Файл не найден."); // Вывод ошибки
        }
        catch (Exception ex) // Общая обработка других исключений
        {
            Console.WriteLine($"Произошла ошибка: {ex.Message}"); // Вывод ошибки
        }
    }
}
```



### 2) Чтение построчно

Применяется для работы с текстовыми файлами, где данные структурированы по строкам. Этот метод удобен для обработки данных, например, логов или таблиц, где каждая строка представляет собой отдельную запись.

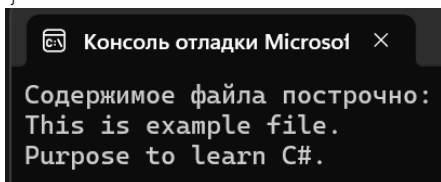
```
using System; // Подключение стандартного пространства имен
using System.IO; // Подключение пространства имен для работы с файлами

class Program // Создаем программу
{
    static void Main() // функция Main()
    {
        try
        {
            // Читаем файл построчно в массив строк
            string[] lines = File.ReadAllLines("example.txt"); // Чтение всех
            строк из файла
        }
    }
}
```

```

        Console.WriteLine("Содержимое файла построчно:"); // Вывод информации
о начале построчного чтения
        foreach (string line in lines) // Перебираем каждую строку в массиве
строк
        {
            Console.WriteLine(line); // Выводим текущую строку на экран
        }
    }
    catch (FileNotFoundException) // Ловим ошибку, если файл не найден
    {
        Console.WriteLine("Ошибка: Файл не найден."); // Выводим сообщение об
ошибке
    }
    catch (Exception ex) // Ловим все остальные возможные ошибки
    {
        Console.WriteLine($"Произошла ошибка: {ex.Message}"); // Выводим
сообщение об ошибке
    }
}
}

```



### 3) Чтение данных поблочно

Используется для больших файлов, которые нельзя загрузить в память целиком. Такой подход позволяет обрабатывать данные постепенно, без перегрузки системы. Для этого часто применяют классы `StreamReader` или `FileStream`. Рассмотрим пример `StreamReader`.

```

using System; // Подключение стандартного пространства имен
using System.IO; // Подключение пространства имен для работы с файлами

class Program // Создаем программу
{
    static void Main() // функция Main()
    {
        try
        {
            // Открываем файл для чтения с помощью StreamReader
            using (StreamReader reader = new StreamReader("example.txt")) //
Создаем объект StreamReader для чтения файла
            {
                Console.WriteLine("Содержимое файла (чтение по строкам):"); //
Вывод информации о начале чтения

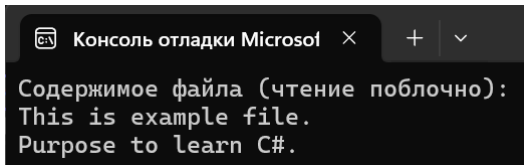
                string line; // Переменная для хранения текущей строки
                while ((line = reader.ReadLine()) != null) // Читаем файл
построчно, пока строки не закончатся
                {
                    Console.WriteLine(line); // Выводим текущую строку на экран
                }
            } // Завершаем использование StreamReader (ресурсы автоматически
освобождаются)
        }
        catch (FileNotFoundException) // Ловим ошибку, если файл не найден
        {
            Console.WriteLine("Ошибка: Файл не найден."); // Выводим сообщение об
ошибке
        }
    }
}

```

```

        catch (Exception ex) // Ловим все остальные возможные ошибки
        {
            Console.WriteLine($"Произошла ошибка: {ex.Message}"); // Выводим
            // сообщение об ошибке
        }
    }
}

```



## Задание №4.

**Специальные символы в C#** — это управляющие последовательности, которые позволяют добавлять в строки символы, не поддерживаемые напрямую, или изменять их форматирование. Они представляют собой комбинацию обратного слэша (\) с определённой буквой или символом. Такие последовательности особенно полезны, когда требуется включить в текст строки управляющие символы, как перенос строки, отступы, кавычки, и другие.

### Основные специальные символы в C#

\n — перенос строки.

\t — табуляция.

\" — двойные кавычки внутри строки.

\\ — обратный слэш.

\r — возврат каретки (обычно используется в Windows).

\b — символ возврата на один символ назад.

\' — одинарная кавычка.

\f — перевод формата страницы.

\v — вертикальная табуляция.

### Пример программы с использованием специальных символов:

```

using System; // Подключение стандартного пространства имен

class Program // Создаем программу
{
    static void Main() // функция Main()
    {
        // Пример использования переноса строки (\n)
        Console.WriteLine("Приветствую всех!\nЭто пример строки с переносом."); // Вывод двух строк с переносом строки

        // Пример использования табуляции (\t)
        Console.WriteLine("Список товаров:\n\t1. Хлеб\n\t2. Молоко\n\t3. Пицца"); // Вывод списка с табуляцией

        // Пример использования двойных кавычек (")
        Console.WriteLine("Тренер мне сказал: \"Работай усерднее!\""); // Вставка
        // двойных кавычек в строку
    }
}

```

```

// Пример использования обратного слэша (\\)
Console.WriteLine("Путь к файлу: C:\\Program Files\\MyApp"); // Вывод пути
к файлу с обратными слэшами

// Пример использования возврата каретки (\\r)
Console.WriteLine("Начало строки\\rКонец"); // Текст "Конец" перезаписывает
"Начало строки"

// Пример использования одинарной кавычки (\\')
Console.WriteLine("Я сказал: \\Привет!\\', и ушёл как обычно."); // Вставка
одинарной кавычки в строку

// Пример использования возврата на один символ назад (\\b)
Console.WriteLine("Символ\\b удалён."); // Удаляет последний символ перед
\\b

// Пример использования перевода формата страницы (\\f)
Console.WriteLine("Текст до перевода формата\\fТекст после перевода
формата."); // Вставка символа перевода формата страницы

// Пример использования вертикальной табуляции (\\v)
Console.WriteLine("Элемент A\\vЭлемент B\\vЭлемент C"); // Вывод строк с
вертикальной табуляцией
}
}

```

```

Консоль отладки Microsoft
Приветствую всех!
Это пример строки с переносом.
Список товаров:
  1. Хлеб
  2. Молоко
  3. Пицца
Тренер мне сказал: "Работай усерднее!"
Путь к файлу: C:\Program Files\MyApp
Конец строки
Я сказал: 'Привет!', и ушёл как обычно.
Символ удалён.
Текст до перевода формата
Текст после перевода формата.
Элемент A
Элемент B
Элемент C

```

## Задание №5.

**Поля класса** — это переменные, которые хранят данные или состояние объекта. Они определяются внутри класса, но вне методов, конструкторов и других членов. Поля являются важной частью объекта, так как содержат информацию, связанной с этим объектом.

Основные характеристики полей класса

1. Модификаторы доступа:

`public` — доступно из любого места в коде.

`private` — доступно только внутри самого класса.

`protected` — доступно внутри класса и его производных классов.

`internal` — доступно только внутри текущей сборки.

2. Типы полей:



Экземплярные поля: принадлежат конкретному объекту.

Статические поля: принадлежат самому классу, а не его объектам.

3. Инициализация полей: Поля могут быть инициализированы сразу при объявлении или в конструкторе класса.

4. Константы и поля readonly:

const — значение задаётся на этапе компиляции и не изменяется.

readonly — значение может быть задано только при создании объекта или в конструкторе.

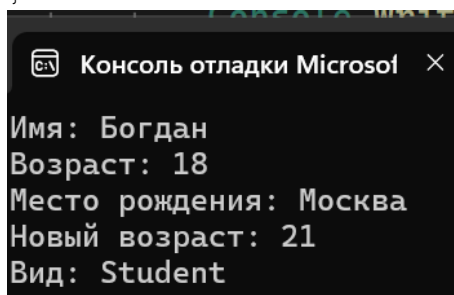
Пример программы с полями класса:

```
using System; // Подключение стандартного пространства имен
// Определение класса Person
class Person
{
    public string Name; // Публичное поле для хранения имени человека
    private int age; // Приватное поле для хранения возраста человека
    public static string Species = "Student"; // Статическое поле для хранения
вида, общее для всех объектов класса
    public readonly string Birthplace; // Поле только для чтения, задаётся при
создании объекта и не изменяется
    // Конструктор для инициализации полей
    public Person(string name, int age, string birthplace)
    {
        Name = name; // Инициализация поля Name переданным значением name
        this.age = age; // Инициализация приватного поля age переданным значением
age
        Birthplace = birthplace; // Инициализация поля только для чтения
Birthplace переданным значением birthplace
    }
    // Метод для получения значения приватного поля age
    public int GetAge()
    {
        return age; // Возвращает значение приватного поля age
    }
    // Метод для изменения значения приватного поля age
    public void SetAge(int newAge)
    {
        if (newAge > 0) // Проверка: новое значение возраста должно быть
положительным
        {
            age = newAge; // Установка нового значения возраста, если условие
выполнено
        }
    }
}
// Основной класс программы
class Program
{
    static void Main()
    {
        // Создание объекта класса Person с указанием имени, возраста и места
рождения
        Person person1 = new Person("Богдан", 18, "Москва"); // Создаём объект
person1
        // Работа с публичным полем Name
        Console.WriteLine($"Имя: {person1.Name}"); // Выводим значение публичного
поля Name
        // Работа с приватным полем age через метод GetAge
        Console.WriteLine($"Возраст: {person1.GetAge()}"); // Выводим значение
возраста через метод GetAge
    }
}
```

```

        // Работа с полем только для чтения Birthplace
        Console.WriteLine($"Место рождения: {person1.Birthplace}"); // Выводим
значение поля Birthplace
        // Изменение значения возраста через метод SetAge
        person1.SetAge(21); // Вызываем метод SetAge для изменения возраста
        Console.WriteLine($"Новый возраст: {person1.GetAge()}"); // Выводим
обновлённое значение возраста через метод GetAge
        // Работа со статическим полем Species
        Console.WriteLine($"Вид: {Person.Species}"); // Выводим значение
статического поля Species, доступного через имя класса
    }
}
}
// Основной класс программы
class Program
{
    static void Main()
    {
        // Создание объекта класса Person с указанием имени, возраста и места
        рождения
        Person person1 = new Person("Иван", 25, "Москва"); // Создаём объект
        person1
        // Работа с публичным полем Name
        Console.WriteLine($"Имя: {person1.Name}"); // Выводим значение публичного
поля Name
        // Работа с приватным полем age через метод GetAge
        Console.WriteLine($"Возраст: {person1.GetAge()}"); // Выводим значение
возраста через метод GetAge
        // Работа с полем только для чтения Birthplace
        Console.WriteLine($"Место рождения: {person1.Birthplace}"); // Выводим
значение поля Birthplace
        // Изменение значения возраста через метод SetAge
        person1.SetAge(30); // Вызываем метод SetAge для изменения возраста
        Console.WriteLine($"Новый возраст: {person1.GetAge()}"); // Выводим
обновлённое значение возраста через метод GetAge
        // Работа со статическим полем Species
        Console.WriteLine($"Вид: {Person.Species}"); // Выводим значение
статического поля Species, доступного через имя класса
    }
}

```



## Задание №6.

В C# классы и структуры используются для создания пользовательских типов данных. Оба конструкта позволяют группировать данные и действия (методы) вместе, но между ними есть существенные отличия. Выбор между классом и структурой зависит от характера данных и поведения, которое требуется для приложения.

### Основные отличия классов и структур

1. Тип хранения данных:
  - Классы — ссылочные типы, хранятся в куче (heap). Переменная класса содержит ссылку на объект, а не сам объект.
  - Структуры — значимые типы, хранятся в стеке (stack). Переменная структуры содержит сам объект.
2. Создание объектов:
  - Классы создаются с использованием ключевого слова new, и всегда передаются по ссылке.
  - Структуры могут быть созданы без использования new. Они передаются по значению, копируя содержимое.
3. Изменяемость:
  - Классы могут быть изменяемыми (mutable).
  - Структуры рекомендуется делать неизменяемыми (immutable), так как копирование больших структур по значению может повлиять на производительность.
4. Наследование:
  - Классы поддерживают наследование.
  - Структуры не поддерживают наследование, но могут реализовывать интерфейсы.
5. Работа с null:
  - Классы могут быть равны null (ссылка на объект отсутствует).
  - Структуры не могут быть равны null, если не использовать Nullable<T>.
6. Размер и производительность:
  - Классы эффективнее для крупных объектов, так как передаются по ссылке.
  - Структуры подходят для небольших объектов с коротким временем жизни, так как копирование данных в стеке быстрее.

## Пример для иллюстрации отличий:

```
using System; // Подключение стандартного пространства имен

// Определение класса
class MyClass
{
    public int Value; // Поле класса для хранения значения
}
// Определение структуры
struct MyStruct
{
    public int Value; // Поле структуры для хранения значения
}
class Program // Создаем программу
{
    static void Main() // функция Main()
    {
        // Работа с классом
        MyClass class1 = new MyClass(); // Создаём объект класса MyClass
        class1.Value = 10; // Устанавливаем значение поля Value
        MyClass class2 = class1; // class2 получает ссылку на тот же объект, что и
class1
        class2.Value = 20; // Изменяем значение через class2
        Console.WriteLine($"Класс: class1.Value = {class1.Value}, class2.Value =
{class2.Value}"); // class1 и class2 ссылаются на один объект

        // Работа со структурой
        MyStruct struct1 = new MyStruct(); // Создаём объект структуры MyStruct
        struct1.Value = 10; // Устанавливаем значение поля Value
        MyStruct struct2 = struct1; // struct2 получает копию значений struct1
```

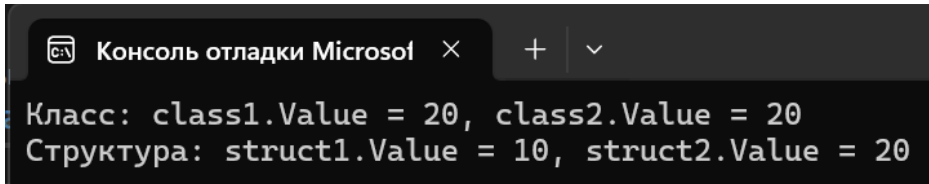
```
        struct2.Value = 20; // Изменяем значение через struct2
        Console.WriteLine($"Структура: struct1.Value = {struct1.Value},
struct2.Value = {struct2.Value}"); // struct1 и struct2 независимы
    }
}
```

Класс (MyClass) - class1 и class2 ссылаются на один и тот же объект.

Изменение class2.Value влияет на значение class1.Value, так как они оба указывают на один объект в куче.

Структура (MyStruct) - struct1 и struct2 хранят независимые копии данных.

Изменение struct2.Value не влияет на struct1.Value, так как данные находятся в стеке.



```
Консоль отладки Microsoft  ×  +  ▾
Класс: class1.Value = 20, class2.Value = 20
Структура: struct1.Value = 10, struct2.Value = 20
```

Конец работы.