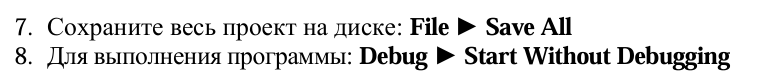
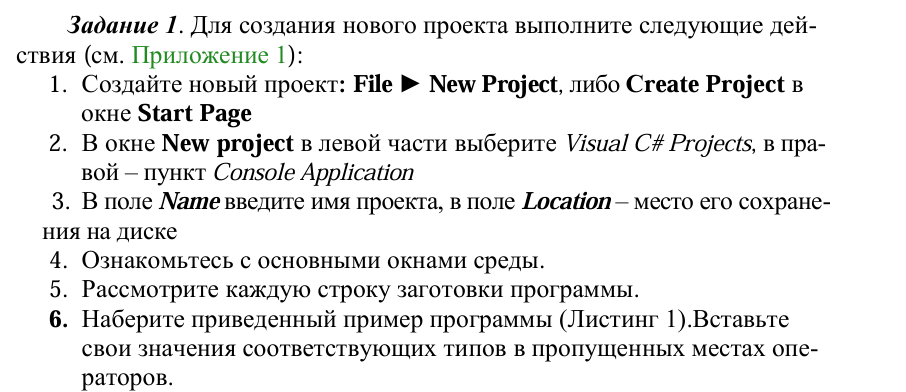
**Лабораторная работа №1**

«СОЗДАНИЕ КОНСОЛЬНОГО ПРИЛОЖЕНИЯ. ВВОД И ВЫВОД ДАННЫХ»

Вариант 3

using System;

namespace lr\_1

{

class Program

{

public static void Main(string[] args)

{

int x = 1;

Console.WriteLine("x: " + x);

double y = 2;

Console.WriteLine("y: " + y);

double z = 3;

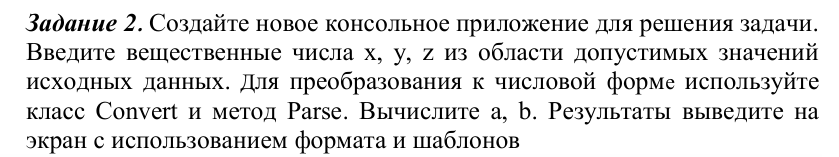
Console.WriteLine("z: " + z);

Console.ReadLine();

}

}

}



using System;

namespace lr\_2

{

internal class Program

{

static void Main(string[] args)

{

double x = GetDoubleFromUser();

double y = GetDoubleFromUser();

double z = GetDoubleFromUser();

double a, b;

a = Math.Pow(1 + y, 2) \* (Math.Pow(x, 2) + 4) / (Math.Pow(Math.E, -x) + Math.Pow(x, 2) + 4);

b = 1 / (Math.Pow(x, 4) / 2 + Math.Pow(Math.Sin(z), 4) + 1);

Console.WriteLine($"a = {a}\nb = {b}");

Console.ReadLine();

}

private static double GetDoubleFromUser() => double.Parse(Console.ReadLine());

}

}

**Контрольные вопросы**

1. Основные принципы технологии .NET.

Кроссплатформенность: Возможность разработки и запуска приложений на разных операционных системах.Управляемый код: Код, исполняемый в среде выполнения CLR (Common Language Runtime), что обеспечивает защиту памяти и автоматическое управление памятью.Язык независимости: Поддержка множества языков программирования, которые могут взаимодействовать друг с другом.Богатая библиотека классов: Набор библиотек, предоставляющих готовые решения для различных задач.

1. Что представляет собой платформа Visual Studio.NET?

Visual Studio.NET — это интегрированная среда разработки (IDE), предоставляющая инструменты для разработки, отладки и развертывания приложений на .NET. Она включает редактор кода, инструменты для управления проектами, отладчик, а также множество библиотек и компонентов для упрощения разработки.

1. Как создать консольное приложение?

В Visual Studio:Откройте Visual Studio.Выберите "Создать новый проект".Выберите шаблон "Консольное приложение" (.NET Core или .NET Framework).Укажите имя проекта и его расположение.Нажмите "Создать".

1. Принципы объектно-ориентированного программирования.

Инкапсуляция: Скрытие внутренней реализации объекта и предоставление доступа к его данным только через методы.Наследование: Возможность создавать новые классы на основе существующих, унаследовав их свойства и методы.Полиморфизм: Возможность использования одного интерфейса для работы с объектами разных классов.Абстракция: Выделение общих характеристик и поведения объектов, создание абстрактных классов и интерфейсов.

1. Литералы. Как определяются типы литералов?

Литералы — это фиксированные значения, используемые в коде. Тип литерала определяется на основании контекста, в котором он используется, или явным образом через суффиксы (например, 123 — это int, 123.45 — это double, 'A' — char, "Hello" — string).

1. Какие типы относятся к встроенным?

К встроенным типам относятся примитивные типы данных, такие как:int, long, short, bytefloat, double, decimalchar, string, bool

1. Чем отличаются типы-значения и ссылочные типы?

Типы-значения: Хранят данные непосредственно, например, int, float, bool. Они находятся в стеке и имеют фиксированный размер.Ссылочные типы: Хранят ссылку на объект в памяти, например, string, class, array. Они находятся в куче и могут занимать переменный размер.

1. Какие типы числовых переменных имеются?

int, long, short, byte (целочисленные типы)float, double, decimal (числа с плавающей запятой)

1. Что такое объявление и инициализация?

Объявление — это создание переменной с указанием ее типа. Например: int x;Инициализация — это присвоение значения переменной. Например: x = 5;

1. Для чего используется упаковка и распаковка?

Упаковка (boxing) — это процесс преобразования типа-значения в тип-объект (например, преобразование int в object).Распаковка (unboxing) — это процесс извлечения типа-значения из типа-объекта.Эти операции необходимы для работы с коллекциями и другими структурами данных, которые принимают ссылочные типы.

1. Как в С# выполняется преобразование типа?

Преобразование может быть явным (используя оператор приведения) или неявным (для совместимых типов). Например:Явное: int x = (int)3.14;Неявное: double d = 5;

1. Как осуществляется консольный ввод?

Для чтения данных из консоли используется метод Console.ReadLine(), который считывает строку и возвращает её как string.

1. Чем отличаются методы Read и ReadLine?

Console.Read() считывает один символ и возвращает его как int, в то время как Console.ReadLine() считывает всю строку до нажатия клавиши Enter и возвращает её как string.

1. Как обеспечить вывод данных на экран?

Для вывода данных на экран используется метод Console.WriteLine(), который отображает строку и переводит курсор на новую строку.

1. Для чего предназначен и как используется форматный вывод дан ных?

Форматный вывод позволяет отображать данные в определённом формате. Используется метод Console.WriteLine() с форматными строками, например: Console.WriteLine("Value: {0}", variable);

1. Каковы основные правила использования стандартных функций?

Следует учитывать типы и количество передаваемых аргументов.Проверка на возможные исключения и ошибки.Чтение документации для понимания поведения функции.

17. Основные приемы работы в среде разработки Visual Studio.NET:

− Как создать консольное приложение?

− Как сохранить проект с заданным именем?

− Как загрузить проект?

− Как выполнить отладку программы?

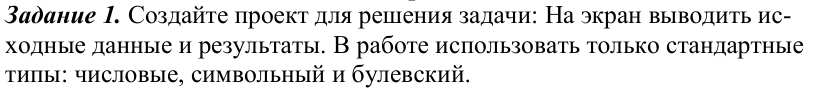
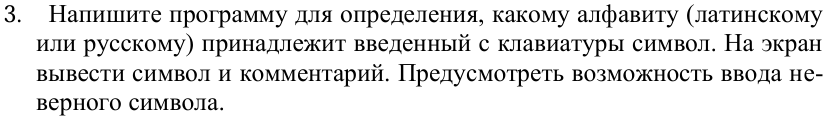
− Как откомпилировать и выполнить программу?

− Как просмотреть результаты выполнения программы?

Как создать консольное приложение: Смотри пункт 3.Как сохранить проект с заданным именем: Нажмите "Файл" > "Сохранить как..." и задайте имя.Как загрузить проект: Нажмите "Файл" > "Открыть" > "Проект" и выберите нужный проект.Как выполнить отладку программы: Установите точки останова и нажмите F5 или выберите "Отладка" > "Запустить отладку".Как откомпилировать и выполнить программу: Нажмите Ctrl+F5 или выберите "Отладка" > "Запустить без отладки".Как просмотреть результаты выполнения программы: Результаты будут отображены в консольном окне, если это консольное приложение, или в окне вывода, если это другой тип приложения.

**Лабораторная работа №2**

«ПРОСТЕЙШИЕ КЛАССЫ. ИНКАПСУЛЯЦИЯ И СВОЙСТВА»

Вариант 3

using System;

using System.Threading;

namespace lr2\_1

{

internal class Program

{

static void Main(string[] args)

{

Console.WriteLine("Введите символ");

char ui = (char)Console.Read();

if (!IsCorrectInput(ui))

{

Console.WriteLine("Некорректный ввод");

}

else

if (IsLatin(ui))

Console.WriteLine($"{ui} - Латинский");

else

Console.WriteLine($"{ui} - Русский");

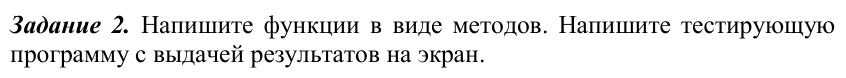
Thread.Sleep(3000);

}

private static bool IsLatin(char inp) => inp >= 'a' && inp <= 'z';

private static bool IsCorrectInput(char inp) => (inp >= 'а' && inp <= 'я' || inp == 'ё') || (inp >= 'a' && inp <= 'z');

}

}

using System;

using System.Collections.Generic;

namespace lr\_2\_2

{

internal class Program

{

static void Main(string[] args)

{

Console.WriteLine("Когда чисел хватит - введите 'ост'");

List<int> list = new List<int>();

while (true)

{

string inp = Console.ReadLine();

if (inp == "ост") break;

list.Add(int.Parse(inp));

}

RemoveAll1And3(list);

CheckForAll1And3Removed(list);

Console.ReadLine();

}

private static void CheckForAll1And3Removed(List<int> list)

{

if (list.Contains(1) || list.Contains(3))

Console.WriteLine("Значения 1 или 3 есть в списке");

else

Console.WriteLine("Нет значений 1 или 3");

}

private static void RemoveAll1And3(List<int> list) => list.RemoveAll(x => x == 1 || x == 3);

}

}

**Контрольные вопросы**

1. Выражением какого типа является условие в операторе if? Какие зна чения оно может принимать?

Условие в операторе `if` представляет собой булевое выражение, которое может принимать значения `true` (истина) или `false` (ложь).

1. Как работает оператор if, если отсутствует часть else ?

Если часть `else` отсутствует, то оператор `if` выполняет только блок кода, если условие истинно. Если условие ложно, выполняется ничего — управление передается на следующую строку кода после блока `if`.

1. В каких случаях используется оператор switch?

Оператор `switch` используется для выбора одного из нескольких блоков кода для выполнения, в зависимости от значения выражения.

1. Какого типа может быть в операторе switch?

Числовой, строковой и тп

1. В каком случае выполняется последовательность инструкций default ветви?

Последовательность инструкций в ветви `default` выполняется, если ни одно из условий в `case` не совпадает с значением выражения, на которое ссылается оператор `switch`.

1. В чем отличие операторов while и do ... while

Основное отличие операторов `while` и `do ... while` заключается в том, что в `while` проверка условия происходит перед выполнением тела цикла, а в `do ... while` — после выполнения тела, что гарантирует хотя бы одно выполнение тела цикла.

1. Что представляет собой элемент в операторе for ?

Элемент в операторе `for` представляет собой инициализацию переменной, условие продолжения выполнения цикла и шаг изменения переменной

1. Какого типа может быть элемент в цикле for?

Элемент в цикле `for` может быть целочисленного типа, типа `char` или любого другого, поддерживаемого языком, который позволяет использовать переменные в качестве управляющих.

1. Назначение управляющей переменной цикла for?

Управляющая переменная цикла `for` используется для контроля количества итераций цикла и для отслеживания текущего состояния цикла.

1. Назначение управляющих операторов goto, break, continue, return.

- `goto` — используется для безусловного перехода на другую строку кода.

- `break` — завершает выполнение ближайшего блока цикла или `switch`.

- `continue` — пропускает оставшуюся часть текущей итерации цикла и переходит к следующей.

- `return` — завершает выполнение метода и возвращает управление в место его вызова.

11. Как программируются циклические алгоритмы с явно заданным чис лом повторений цикла?

Циклические алгоритмы с явно заданным числом повторений программируются с помощью цикла `for`, где задается начальное значение, условие продолжения и шаг изменения переменной.

1. Что представляют собой методы?

Методы — это блоки кода, которые выполняют определенные действия и могут принимать параметры и возвращать значения.

1. Как объявляется метод?

Метод объявляется с указанием типа возвращаемого значения, имени метода и списка параметров в круглых скобках. Например:

public int sum(int a, int b) {

return a + b;

}

1. Какова область действия параметров метода?

Область действия параметров метода ограничена телом метода, внутри которого они объявлены. Они недоступны вне этого метода.

1. Как вызываются методы?

Методы вызываются по имени, следуя синтаксису, который включает в себя передачу необходимых аргументов в круглых скобках. Например:

int result = sum(5, 10);

1. Общие (статические) методы класса.

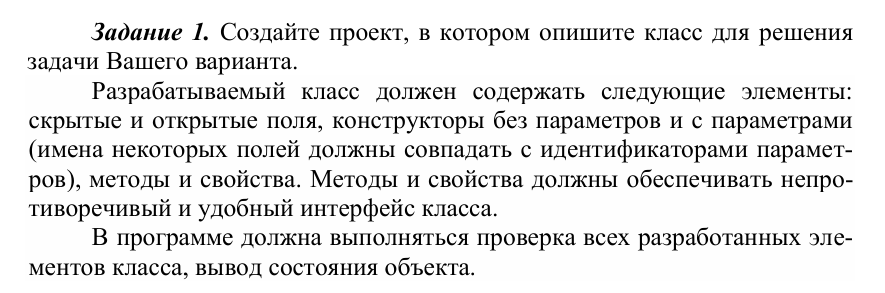
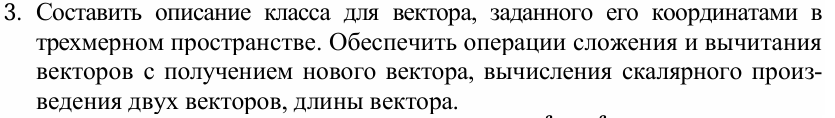
16. Общие (статические) методы класса — это методы, которые принадлежат самому классу, а не его экземплярам. Они могут вызываться без создания объекта класса. Например:

public static void printHello() {

System.out.println("Hello, world!");

**Лабораторная работа №3**

«ПРОСТЕЙШИЕ КЛАССЫ. ИНКАПСУЛЯЦИЯ И СВОЙСТВА»

Вариант 3

Задание 2. Включите в проект Задания 1обработку исключений.

using System;

namespace lr\_3\_1

{

public class Vector

{

public double Length { get => Math.Sqrt(x \* x + y \* y + z \* z); }

private double x, y, z;

public Vector() { }

public Vector(double x, double y, double z)

{

this.x = x;

this.y = y;

this.z = z;

}

public static Vector operator +(Vector a, Vector b) => new Vector(a.x + b.x, a.y + b.y, a.z + b.z);

public static Vector operator - (Vector a, Vector b) => new Vector(a.x - b.x, a.y - b.y, a.z - b.z);

public static double operator \*(Vector a, Vector b) => a.x \* b.x + a.y \* b.y + a.z \* b.z;

public override string ToString() => $"({x}, {y}, {z})";

}

class Program

{

static void Main()

{

Vector vector1 = new Vector(ReadDouble(), ReadDouble(), ReadDouble());

Vector vector2 = new Vector(ReadDouble(), ReadDouble(), ReadDouble());

Vector sum = vector1 + vector2;

Vector difference = vector1 - vector2;

double dotProduct = vector1 \* vector2;

Console.WriteLine($"Vector 1: {vector1}");

Console.WriteLine($"Vector 2: {vector2}");

Console.WriteLine($"Length of Vector 1: {vector1.Length}");

Console.WriteLine($"Length of Vector 2: {vector2.Length}");

Console.WriteLine($"Sum: {sum}");

Console.WriteLine($"Difference: {difference}");

Console.WriteLine($"Dot Product: {dotProduct}");

Vector vector = new Vector();

Console.WriteLine($"zero vector: {vector}");

Vector v1 = new Vector();

Vector v2 = new Vector();

Vector v3 = v1 + v2;

Console.ReadLine();

}

private static double ReadDouble()

{

while(true)

{

try

{

return double.Parse(Console.ReadLine());

}

catch (Exception)

{

Console.WriteLine("Введены некорректные данные, введите вещественное число");

}

}

}

}

}

**Котрольный вопросы**

1. Как описываются классы в C#? Классы в C# описываются с помощью ключевого слова class, за которым следует имя класса и тело класса, заключенное в фигурные скобки. Пример:

public class MyClass

{

// Члены класса

}

3. Что относится к членам класса? Членами класса являются поля, свойства, методы, события и конструкторы. Эти элементы определяют состояние (данные) и поведение (функции) класса.

3. Что такое статические члены класса? Статические члены класса принадлежат самому классу, а не его экземплярам. Они могут быть доступны без создания объекта класса. Для объявления статических членов используется ключевое слово static. Пример:

public class MyClass

{

public static int MyStaticField;

}

4. Данные: поля и константы. Поля — это переменные, которые хранят состояние объекта. Константы — это значения, которые не могут изменяться после их инициализации. Константы объявляются с использованием ключевого слова const. Пример:

public class MyClass

{

public int myField;

public const int myConstant = 10;

}

5. Спецификаторы полей и констант класса. Поля и константы могут иметь различные спецификаторы доступа: public, private, protected, internal и комбинации из этих модификаторов. Это определяет видимость и доступность членов класса.

6. Как передаются параметры в методы? Параметры в методы могут передаваться по значению (по умолчанию) или по ссылке (с использованием ключевого слова ref или out). При передаче по значению создается копия аргумента, а при передаче по ссылке метод может изменять значение переменной, переданной в качестве параметра.

7. Для чего предназначен параметр params? Параметр params позволяет передавать переменное количество аргументов в метод. Он должен быть последним параметром в списке параметров метода и позволяет передавать массив значений.

public void MyMethod(params int[] numbers) { }

8. Что представляет собой конструктор? Для чего он используется? Конструктор — это специальный метод, который вызывается при создании объекта класса. Он используется для инициализации полей и выполнения других необходимых действий при создании экземпляра класса.

9. Какие бывают конструкторы? В C# бывают следующие типы конструкторов:

Конструктор по умолчанию — без параметров.

Конструктор с параметрами — позволяет передавать значения для инициализации полей.

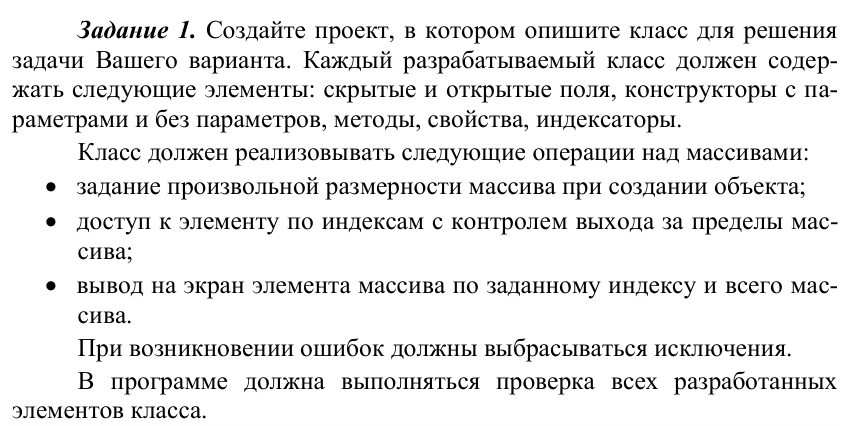
Статический конструктор — используется для инициализации статических членов класса и вызывается перед первым обращением к классу.

10. Может ли класс не иметь конструктора? Да, класс может не иметь явного конструктора. В этом случае компилятор автоматически создает конструктор по умолчанию, который инициализирует поля значениями по умолчанию.

11. Для чего предназначена система сбора мусора? Система сборки мусора (Garbage Collection) в C# автоматически управляет памятью, освобождая неиспользуемые объекты и предотвращая утечки памяти. Это позволяет разработчикам сосредоточиться на логике приложения, не беспокоясь о ручном управлении памятью.

**Лабораторная работа №4**

«МАССИВЫ. ИНДЕКСАТОРЫ»

Вариант 3

3. Описать класс для работы с одномерным массивом целых чисел. Обеспечить возможность нахождения суммы элементов, стоящих после введенного с клавиатуры значения и вывода элементов, стоящих после введенного с клавиатуры значения

using System;

using System.Collections.Generic;

using System.Linq;

namespace lr\_4\_1

{

internal class Program

{

static void Main(string[] args)

{

ArrayAdder a = new ArrayAdder();

//a.ShowElement(10);

//a.ShowElement(-1);

a.ShowArrayState();

a.ShowElement(2);

Console.WriteLine("Введите элемент для вывода и суммирования элементов после него");

int el = int.Parse(Console.ReadLine());

a.ShowElementsAfter(el);

Console.WriteLine($"Сумма последовательности элементов: {a.SumElementsAfter(el)}");

Console.ReadLine();

}

}

class ArrayAdder

{

private int[] arr;

public int this[int i]

{

get

{

if (arr.Length <= i || i < 0)

throw new Exception("Данное число выходит за границы индексов массива");

return arr[i];

}

set

{

if (arr.Contains(value))

throw new Exception("Данное значение уже имеется");

arr[i] = value;

}

}

public ArrayAdder()

{

Random rnd = new Random();

arr = new int[10];

HashSet<int> set = new HashSet<int>();

while (set.Count < 10)

set.Add(rnd.Next(1, 11));

arr = set.ToArray();

}

public ArrayAdder(int n)

{

arr = new int[n];

}

public void ShowElement(int i) => Console.WriteLine(this[i]);

public void ShowArrayState() => Console.WriteLine(string.Join(" ", arr));

public void ShowElementsAfter(int el)

{

string res = "";

int i = GetElementIndex(el) + 1;

for (int j = i; j < arr.Length; j++)

res += $"{arr[j]} ";

Console.WriteLine(res);

}

public int SumElementsAfter(int el)

{

int i = GetElementIndex(el) + 1;

int sum = 0;

for (int j = i; j < arr.Length; j++)

sum += arr[j];

return sum;

}

private int GetElementIndex(int el)

{

for (int i = 0; i < arr.Length; i++)

if (arr[i] == el)

return i;

throw new Exception("Массив не имеет данного элемента");

}

}

}

**Котрольный вопросы**

1. Что понимается под массивом? Массив в C# — это структура данных, которая позволяет хранить фиксированное количество элементов одного типа. Массивы могут быть одномерными и многомерными.

2. Каковы возможные способы описания массивов (одномерных и многомерных)? Одномерный массив можно объявить следующим образом:

int[] array = new int[5]; // Массив из 5 целых чисел

Для многомерного массива, например, двумерного, используется следующий синтаксис:

int[,] matrix = new int[3, 4]; // Двумерный массив 3x4

3. В каких случаях целесообразно описывать двумерный массив с помощью одномерных? Двумерный массив можно описать с помощью одномерных, когда необходимо экономить память или когда доступ к элементам осуществляется по одному измерению за раз. Например, можно использовать одномерный массив и вычислять индексы для доступа ко элементам:

int[] oneDimArray = new int[12];

int rows = 3;

int cols = 4;

int index = row \* cols + col; // Доступ к элементу по двум индексам

4. Какие типы допустимы для описания индексов массивов? Индексами массивов могут быть только целые числа (типы int, uint, long, ulong и т. д.). В C# индексы массивов начинаются с нуля.

5. Какие типы могут использоваться в качестве базовых для описания массивов? В C# массивы могут быть созданы для любых типов данных, включая примитивные типы (например, int, double, char) и ссылочные типы (например, объекты пользовательских классов).

6. Как осуществляется ввод и вывод массивов? Ввод и вывод массивов можно осуществлять с помощью циклов. Например:

// Ввод

for (int i = 0; i < array.Length; i++)

{

array[i] = Convert.ToInt32(Console.ReadLine());

}

// Вывод

foreach (var item in array)

{

Console.WriteLine(item);

}

7. Для чего предназначен цикл foreach? Цикл foreach предназначен для перебора элементов коллекции или массива без необходимости управления индексами. Это делает код более читаемым и уменьшает вероятность ошибок.

8. Можно ли использовать цикл foreach для ввода элементов массива? Да, цикл foreach может быть использован для вывода элементов массива, но для ввода элементов он не подходит, так как не позволяет изменять значения элементов. Для ввода лучше использовать обычный цикл for.

9. Как определяется базовый тип индексатора? Базовый тип индексатора определяется в определении индексатора в классе. Это тип, который будет использоваться для хранения значений, например:

public int this[int index] // индексатор

{

get { return array[index]; }

set { array[index] = value; }

}

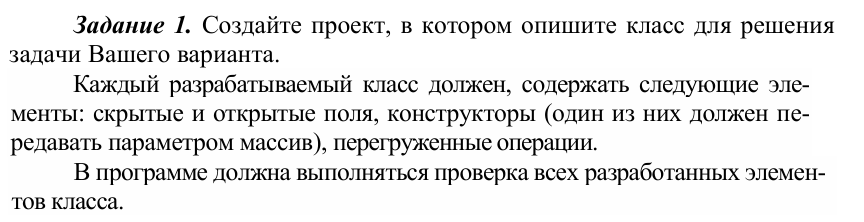
10. Что записывается в качестве имени индексатора? В C# индексаторы не имеют имени в традиционном смысле; вместо этого они используются с ключевым словом this, и к ним можно обращаться через квадратные скобки.

11. Что содержит список параметров индексатора? Список параметров индексатора содержит параметры, которые используются для доступа к элементам массива или коллекции. Например, в случае одномерного массива это может быть один параметр типа int, а для двумерного массива — два параметра:

public int this[int row, int col]

**Лабораторная работа №5**

«ОПЕРАЦИИ КЛАССА. СТРОКИ.»

Вариант 3

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace lr\_5\_1

{

internal class Program

{

static void Main(string[] args)

{

StringSequence firstSeq = new StringSequence(new string[] { "hey", "bye" });

StringSequence secondSeq = new StringSequence(new string[] { "hello", "bye" });

Console.WriteLine($"{firstSeq == secondSeq} {firstSeq != secondSeq}");

firstSeq.Seq = secondSeq.Seq;

Console.WriteLine($"{firstSeq == secondSeq} {firstSeq != secondSeq}");

Console.ReadLine();

}

}

class StringSequence

{

private string[] arr;

public string[] Seq

{

get => arr;

set => arr = value;

}

public StringSequence(string[] arr)

{

this.arr = arr;

}

public StringSequence() { }

public static bool operator ==(StringSequence left, StringSequence right)

{

bool res = true;

if (left.arr.Length != right.arr.Length)

return false;

for (int i = 0; i < left.arr.Length; i++)

res &= left.arr[i] == right.arr[i];

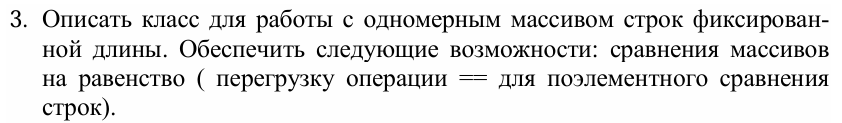
return res;

}

public static bool operator !=(StringSequence left, StringSequence right) => !(left == right);

}

}



**Котрольный вопросы**

1. Что представляет собой перегрузка методов?

Перегрузка методов — это возможность объявить несколько методов с одним и тем же именем, но с различными параметрами (разное количество, порядок или типы параметров) в одном классе. Это позволяет вызывать один и тот же метод с разными параметрами, что повышает читаемость и удобство использования кода.

2. Что представляет собой перегрузка операций?

Перегрузка операций — это возможность определить новые значения для стандартных операторов (таких как `+`, `-`, `\*`, и т.д.) для пользовательских типов данных. Это позволяет использовать операторы с экземплярами пользовательских классов так же, как с примитивными типами данных.

3. Формат описания операции класса.

Операции в классе описываются с использованием ключевого слова `operator`, за которым следует символ операции. Например, для перегрузки оператора сложения `+` это будет выглядеть так:

public static MyClass operator +(MyClass a, MyClass b) {

// реализация операции

}

4. Какие операции нельзя перегружать?

В C# нельзя перегружать следующие операции:

- Оператор `new` (выделение памяти)

- Оператор `is` (проверка типа)

- Оператор `as` (приведение типа)

5. Что является результатом перегрузки унарных операций?

Результатом перегрузки унарных операций является новый экземпляр пользовательского класса или измененный экземпляр класса, в зависимости от логики, реализованной в перегруженном методе оператора.

6. Какие параметры могут быть у бинарных операций класса?

Бинарные операции принимают два параметра, оба из которых должны быть экземплярами пользовательского класса или одного из них может быть примитивным типом. Например:

public static MyClass operator +(MyClass a, MyClass b) {

// реализация операции

}

7. Как выполняется перегрузка операций отношения?

Перегрузка операций отношения (например, `==`, `!=`, `<`, `>`, `<=`, `>=`) выполняется аналогично другим операциям, с использованием ключевого слова `operator`. Например:

public static bool operator ==(MyClass a, MyClass b) {

// реализация сравнения для ==

}

public static bool operator !=(MyClass a, MyClass b) {

// реализация сравнения для !=

}

8. Чем являются строки в C#?

В C# строки представляют собой объекты класса `string`, который является неизменяемым (immutable) типом данных. Это означает, что после создания строки ее содержимое нельзя изменить.

9. Какие операции определены для строк?

Для строк в C# определены различные операции, такие как:

- Конкатенация (`+` и `+=`)

- Сравнение (`==` и `!=`)

- Индексация (для доступа к символам строки)

- Методы для поиска, замены, разбиения и форматирования строк (например, `Substring()`, `Replace()`, `Split()`, `ToUpper()`, `ToLower()` и т.д.)

10. Как создаются строки?

Строки в C# могут создаваться несколькими способами:

- Литералы строк:

string str = "Hello, World!";

- Конструкторы класса `string`:

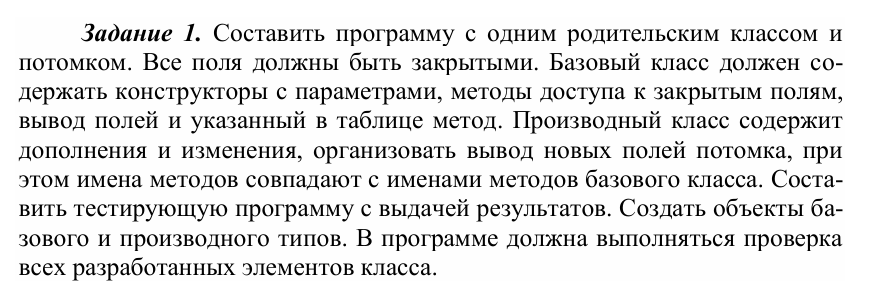
string str = new string('a', 5); // "aaaaa"

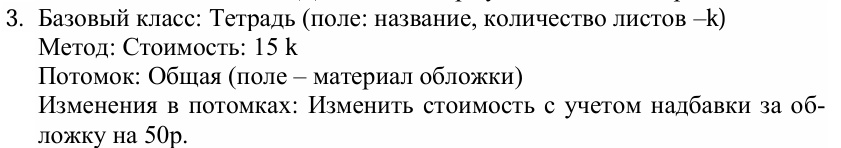
11. Можноли изменять значение строки?

Нет, строки в C# являются неизменяемыми. Когда вы выполняете операцию, которая изменяет строку (например, конкатенацию), создается новая строка, а оригинальная остается неизменной.

**Лабораторная работа №6**

«ИЕРАРХИИ КЛАССОВ. НАСЛЕДОВАНИЕ.»

Вариант 3



using System;

namespace lr\_6\_1

{

internal class Program

{

static void Main(string[] args)

{

Notebook notebook1 = new Notebook("Обычная в клетку", 12);

CoveredNotebook notebook2 = new CoveredNotebook("Буржуйная в клетку", 24, "Картон");

Console.WriteLine(notebook1);

Console.WriteLine(notebook2);

Console.ReadLine();

}

}

class Notebook

{

private string name;

public string Name => name;

private int k;

public int Sheets => k;

public virtual int Price => k \* 15;

public Notebook(string name, int k)

{

this.name = name;

this.k = k;

}

public override string ToString() => $"Информация о тетради\n\tНазвание: {name}\n\tКол-во листов: {k}\n\tЦена: {Price}";

}

class CoveredNotebook : Notebook

{

public override int Price => base.Price + 50;

private string coverMaterial;

public CoveredNotebook(string name, int k, string material) : base(name, k)

{

coverMaterial = material;

}

public override string ToString() => base.ToString() + $"\n\tМатериал обложки: {coverMaterial}";

}

}

**Контрольные вопросы**

1. Принцип наследования:

Наследование — это механизм объектно-ориентированного программирования (ООП), который позволяет создавать новый класс на основе существующего. Новый класс (производный или дочерний) наследует члены (свойства и методы) базового класса (родительского), что позволяет повторно использовать код и расширять его функциональность.

2. Члены класса, которые наследуются:

В C# наследуются следующие члены класса:

- Публичные (`public`) члены.

- Защищенные (`protected`) члены.

- Защищенные внутренние (`protected internal`) члены.

Члены, помеченные как `private`, не наследуются.

3. Защищенный доступ:

Защищенный доступ (`protected`) позволяет членам класса быть доступными только в пределах самого класса и в производных классах. Это означает, что дочерние классы могут обращаться к защищенным членам родительского класса, но не могут обращаться к ним из других классов, которые не являются потомками.

4. Вызов конструкторов базового класса:

Конструкторы базового класса вызываются в дочернем классе с помощью ключевого слова `base`. Если в дочернем классе не указан вызов конструктора базового класса, то будет вызван его конструктор без параметров (если он доступен).

class BaseClass

{

public BaseClass(int value) { }

}

class DerivedClass : BaseClass

{

public DerivedClass(int value) : base(value) // Вызов конструктора базового класса

{ }

}

5. Сокрытие имен при наследовании:

Сокрытие имен происходит, когда дочерний класс определяет члена с тем же именем, что и в базовом классе. В этом случае член дочернего класса "скрывает" член базового класса, и при обращении к этому члену из экземпляров дочернего класса будет использоваться версия из дочернего класса. Для явного указания на использование базового члена можно использовать ключевое слово `base`.

6. Получение доступа к сокрытому члену базового класса:

Чтобы получить доступ к сокрытому члену базового класса, можно использовать ключевое слово `base`. Например:

class BaseClass

{

public void Display() { Console.WriteLine("Base Display"); }

}

class DerivedClass : BaseClass

{

public new void Display() // Сокрытие члена

{

Console.WriteLine("Derived Display");

}

public void ShowBaseDisplay()

{

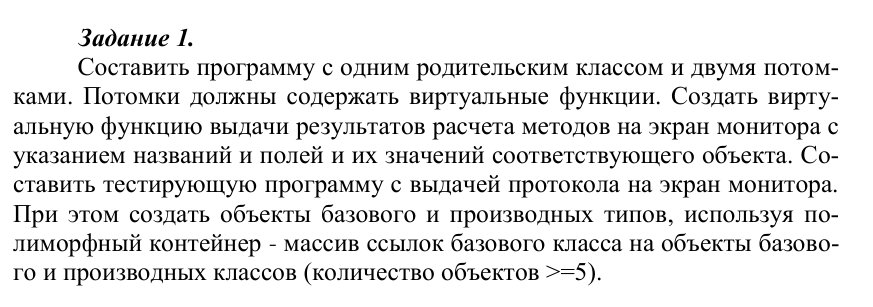
base.Display(); // Вызов метода из базового класса

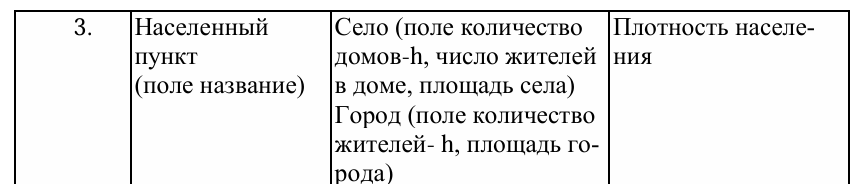
}

}

**Лабораторная работа №7**

«ИЕРАРХИИ КЛАССОВ. НАСЛЕДОВАНИЕ.»

Вариант 3



using System;

namespace lr\_7\_1

{

internal class Program

{

static void Main(string[] args)

{

LivingPlace[] livingPlaces = new LivingPlace[]

{

new City(2\_000\_000, 25000),

new Village(2, 1000, 5),

new City(1\_000\_000, 10000),

new Village(20, 10000, 7),

new Village(5, 10, 100)

};

foreach (var lp in livingPlaces)

Console.WriteLine($"Плотность населения: {lp.GetPopulationDensity()}");

Console.WriteLine(livingPlaces.Select(x => x.GetPopulationDensity()).Min());

Console.ReadLine();

}

}

abstract class LivingPlace

{

protected int h;

protected double square;

protected LivingPlace(int h, double square)

{

this.h = h;

this.square = square;

}

public virtual double GetPopulationDensity() { throw new NotImplementedException(); }

}

class Village : LivingPlace

{

public int Homes => h;

public readonly double AveragePeopleInHouse;

public Village(int homesCount, double square, double averagePeopleInHouse) : base(homesCount, square)

{

AveragePeopleInHouse = averagePeopleInHouse;

}

public override double GetPopulationDensity() => (Homes \* AveragePeopleInHouse) / square;

}

class City : LivingPlace

{

public int Population => h;

public City(int population, double square) : base(population, square) { }

public override double GetPopulationDensity() => Population / square;

}

}

**Контрольные вопросы**

1. Принцип полиморфизма в C# (и в объектно-ориентированном программировании в целом) означает возможность использования объектов разных классов через общий интерфейс. Это позволяет объектам разных типов обрабатывать одни и те же операции, что приводит к более гибкому и расширяемому коду. Полиморфизм достигается через наследование и интерфейсы.

2. Позднее связывание используется для определения того, какой метод будет вызван в момент выполнения, а не в момент компиляции. Это позволяет создавать более гибкие и расширяемые системы, где конкретный тип объекта может быть определен только во время выполнения программы. Позднее связывание особенно полезно в случае работы с базовыми классами и интерфейсами, когда точный тип объекта может быть неизвестен до выполнения кода.

3. Виртуальные методы используются в C# для реализации полиморфизма. Они позволяют подклассам переопределять поведение метода базового класса. Это особенно полезно, когда у вас есть базовый класс с общими методами, и вам нужно изменить или расширить функциональность в производных классах.

4. При переопределении виртуального метода необходимо соблюдать несколько условий:

- Метод должен иметь одинаковое имя, тип возвращаемого значения и параметры.

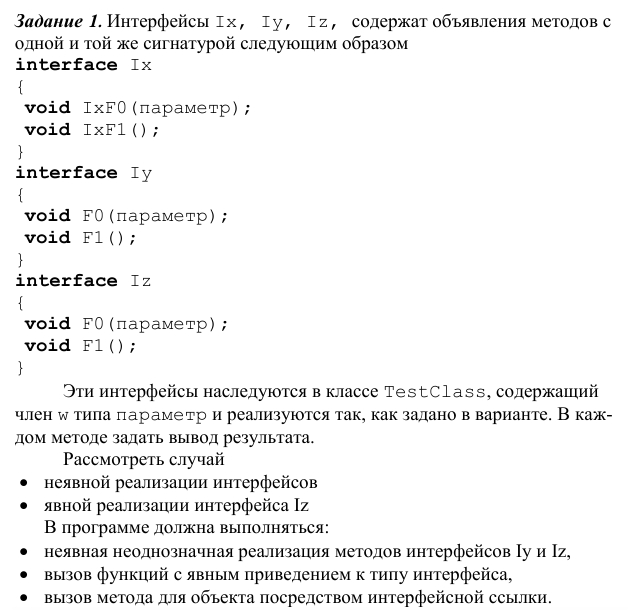
- В производном классе метод должен быть объявлен с ключевым словом `override`.

- Если базовый метод помечен как `virtual`, `abstract` или `override`, его переопределение в производном классе обязательно требует использования ключевого слова `override`.

5. Абстрактные классы — это классы, которые не могут быть инстанцированы напрямую и могут содержать как абстрактные (без реализации), так и неабстрактные (с реализацией) методы. Они предназначены для определения общего интерфейса для производных классов, которые должны реализовать абстрактные методы. Абстрактные классы часто используются как базовые классы для создания иерархий классов, где определяются общие свойства и методы.

6. Да, в абстрактном классе могут быть неабстрактные методы. Это позволяет иметь как абстрактные методы, которые должны быть реализованы в производных классах, так и методы с уже реализованной логикой, которые могут использоваться производными классами. Это дает возможность создать базовый функционал, который может быть использован наследниками.

**Лабораторная работа №8**

«ИНТЕРФЕЙСЫ И СТРУКТУРНЫЕ ТИПЫ»

using System;

namespace lr\_8\_1

{

internal class Program

{

static void Main(string[] args)

{

Iy iy = new TestClass();

iy.F0(2);

((Iz)iy).F1();

Console.ReadLine();

}

}

interface Ix

{

void IxF0(int n);

void IxF1();

}

interface Iy

{

void F0(int n);

void F1();

}

interface Iz

{

void F0(int n);

void F1();

}

class TestClass : Ix, Iy, Iz

{

void Iz.F0(int w) => Console.WriteLine(7 \* w - 2);

void Iz.F1() => Console.WriteLine(-2);

public void F0(int w) => Console.WriteLine(Math.Pow(w, 3));

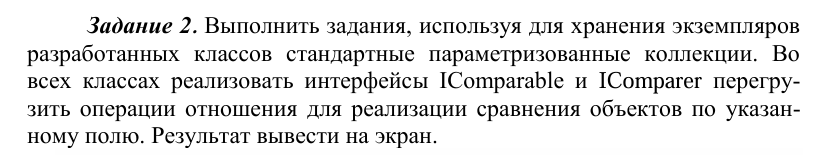
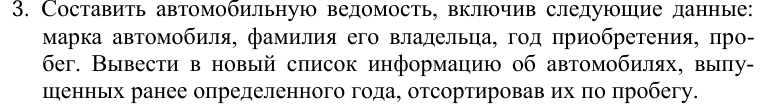
public void F1() => Console.WriteLine(0);

public void IxF0(int w) => Console.WriteLine(w + 5);

public void IxF1() => Console.WriteLine(5);

}

}



using System;

namespace lr\_8\_2

{

using System;

using System.Collections.Generic;

using System.Linq;

class Program

{

public class Car : IComparable<Car>, IComparer<Car>

{

public string Brand { get; set; }

public string Owner { get; set; }

public int YearOfPurchase { get; set; }

public int Mileage { get; set; }

public Car(string brand, string owner, int yearOfPurchase, int mileage)

{

Brand = brand;

Owner = owner;

YearOfPurchase = yearOfPurchase;

Mileage = mileage;

}

public int CompareTo(Car other) => Mileage.CompareTo(other.Mileage);

int IComparer<Car>.Compare(Car x, Car y)

{

if (x.Mileage < y.Mileage) return -1;

if (x.Mileage > y.Mileage) return 1;

return 0;

}

}

static void Main(string[] args)

{

List<Car> cars = new List<Car>

{

new Car("Toyota", "Иванов", 2010, 150000),

new Car("Ford", "Петров", 2005, 200000),

new Car("BMW", "Сидоров", 2015, 50000),

new Car("Audi", "Кузнецов", 2000, 300000),

new Car("Nissan", "Смирнов", 2008, 120000)

};

Console.WriteLine("Машины");

foreach (var car in cars)

{

Console.WriteLine($"Марка: {car.Brand}, Владелец: {car.Owner}, Год приобретения: {car.YearOfPurchase}, Пробег: {car.Mileage}");

}

int filterYear = int.Parse(Console.ReadLine());

var filteredCars = cars

.Where(car => car.YearOfPurchase < filterYear)

.ToList();

filteredCars.Sort();

Console.WriteLine("Автомобили, выпущенные ранее " + filterYear + " года, отсортированные по пробегу:");

foreach (var car in filteredCars)

{

Console.WriteLine($"Марка: {car.Brand}, Владелец: {car.Owner}, Год приобретения: {car.YearOfPurchase}, Пробег: {car.Mileage}");

}

Console.ReadLine();

}

}

}

Вариант 3

**Контрольные вопросы**

1. Как описывается интерфейс? Его назначение. Интерфейс в C# описывается с помощью ключевого слова interface. Его основное назначение – определять контракт, который классы должны реализовать. Интерфейс содержит объявления методов, свойств, событий и индексаторов, но не содержит их реализации. Это позволяет создавать более гибкие и расширяемые архитектуры, где разные классы могут реализовать одни и те же интерфейсы по-своему.

2. Какие члены может содержать интерфейс? Интерфейс может содержать:

Методы

Свойства

События

Индексаторы Все члены интерфейса по умолчанию являются абстрактными и публичными, и не могут иметь реализации (до C# 8.0, где появились "default interface methods").

3. Какие спецификаторы допустимы у методов, реализующих интерфейс? Методы, реализующие интерфейс, должны быть публичными. Спецификаторы доступа, такие как private, protected или internal, недопустимы для методов интерфейса. Однако, при реализации интерфейса в классе, можно использовать любые спецификаторы доступа для методов, реализующих интерфейс.

4. В каких случаях используется явная реализация интерфейса? Явная реализация интерфейса используется, когда класс реализует несколько интерфейсов, которые содержат методы с одинаковыми именами. Это позволяет избежать конфликтов и четко указать, какой метод из какого интерфейса реализуется. Явная реализация осуществляется с помощью указания имени интерфейса перед именем метода.

5. Как осуществляется наследование интерфейсов? Интерфейсы могут наследовать другие интерфейсы с помощью двоеточия (:). При этом класс, реализующий интерфейс-потомок, должен реализовать все члены всех интерфейсов, от которых он наследует.

6. Можно ли явно реализованные методы объявлять виртуальными? Нет, явно реализованные методы не могут быть объявлены виртуальными. Они должны иметь фиксированную реализацию, и виртуальные методы не соответствуют концепции явной реализации.

7. Можно ли повторно реализовать интерфейс, указав его имя в списке предков класса наряду с классом-предком? Да, можно. Класс может наследовать от базового класса и одновременно реализовывать интерфейсы. В этом случае нужно будет реализовать все члены интерфейса.

8. Какие стандартные интерфейсы используются для работы с коллекциями? В C# для работы с коллекциями используются следующие стандартные интерфейсы:

IEnumerable<T> и IEnumerable для итерации по элементам.

ICollection<T> и ICollection для работы с коллекциями, которые могут изменяться.

IList<T> и IList для работы с индексируемыми коллекциями.

IReadOnlyCollection<T> и IReadOnlyList<T> для работы с коллекциями только для чтения.

9. Чем отличаются интерфейсы IComparable и IComparer?

IComparable используется для сравнения объектов одного типа. Он определяет метод CompareTo, который позволяет определить, меньше, больше или равен текущий объект другому объекту.

IComparer используется для сравнения объектов различных типов и определяет метод Compare, который принимает два объекта и возвращает целое число, указывающее их относительный порядок.