Лабораторная работа No 1

Вариант 5

Задание 1. Для создания нового проекта выполните следующие действия (см. Приложение 1):

1. Создайте новый проект: File ► New Project, либо Сreate Project в окне Start Page

2. В окне New project в левой части выберите Visual C# Projects, в правой – пункт Console Application

3. В поле Name введите имя проекта, в поле Location – место его сохранения на диске

4. Ознакомьтесь с основными окнами среды.

5. Рассмотрите каждую строку заготовки программы.

6. Наберите приведенный пример программы (Листинг 1). Вставьте свои значения соответствующих типов в пропущенных местах операторов.

using System;

namespace matemat1

{

class Program

{

public static void Main(string[] args)

{

// После знака "=" вставь свои значения

int x = 5;

double y = 3.14;

double b = 2.71;

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

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

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

}

}

}

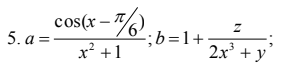
7. Сохраните весь проект на диске: File ► Save All

8. Для выполнения программы: Debug ► Start Without Debugging



Задание 2. Создайте новое консольное приложение для решения задачи.

Введите вещественные числа x, у, z из области допустимых значений исходных данных. Для преобразования к числовой форме используйте класс Convert и метод Parse. Вычислите а, b. Результаты выведите на экран с использованием формата и шаблонов



using System;

class Program

{

static void Main()

{

Console.Write("Введите x: ");

double x = Convert.ToDouble(Console.ReadLine());

Console.Write("Введите y: ");

double y = Convert.ToDouble(Console.ReadLine());

Console.Write("Введите z: ");

double z = Convert.ToDouble(Console.ReadLine());

if (x \* x + 1 == 0 || 2 \* x \* x \* x + y == 0)

{

Console.WriteLine("Ошибка: деление на ноль недопустимо.");

return;

}

double a = Math.Cos(x - Math.PI / 6) / (x \* x + 1);

double b = 1 + z / (2 \* x \* x \* x + y);

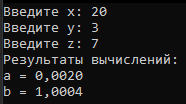
Console.WriteLine($"Результаты вычислений:");

Console.WriteLine($"a = {a:F4}");

Console.WriteLine($"b = {b:F4}");

}

}



Контрольные вопросы:

**1. Основные принципы технологии .NET**

.NET — это платформа для разработки приложений. Основные принципы:

* **Многоплатформенность**: поддержка Windows, macOS, Linux, iOS, Android.
* **Управляемый код**: используется CLR (Common Language Runtime) для выполнения программ, безопасности, сборки мусора.
* **Поддержка нескольких языков программирования**: C#, VB.NET, F#.
* **Большая стандартная библиотека классов**: предоставляет множество встроенных функций и инструментов.

**2. Что представляет собой платформа Visual Studio.NET**

Visual Studio.NET — это среда разработки, предоставляющая:

* Удобный редактор кода с подсветкой синтаксиса.
* Инструменты для отладки и тестирования.
* Возможность создания проектов различных типов (консольные приложения, веб-приложения и др.).
* Интеграцию с системами контроля версий, такими как Git.

**3. Как создать консольное приложение**

1. Откройте Visual Studio.
2. Выберите "Создать новый проект".
3. В списке шаблонов выберите "Консольное приложение (.NET)".
4. Укажите имя и место сохранения проекта.
5. Нажмите "Создать" и начните писать код в файле Program.cs.

**4. Принципы объектно-ориентированного программирования**

Объектно-ориентированное программирование (ООП) базируется на 4 принципах:

* **Инкапсуляция**: скрытие деталей реализации через свойства и методы.
* **Наследование**: возможность одного класса унаследовать свойства и методы другого.
* **Полиморфизм**: возможность методов принимать разные формы (перегрузка, переопределение).
* **Абстракция**: упрощение сложности через выделение важных деталей.

**5. Литералы. Как определяются типы литералов**

Литералы — это фиксированные значения, записанные в коде. Тип определяется:

* **Целочисленные**: 42, 0b101 (двоичные), 0x2A (шестнадцатеричные).
* **Вещественные**: 3.14, 2.5f.
* **Строковые**: "Привет".
* **Булевы**: true, false.
* **Символьные**: 'A'.

**6. Какие типы относятся к встроенным**

Встроенные типы в C#:

* **Целочисленные**: byte, sbyte, short, ushort, int, uint, long, ulong.
* **Вещественные**: float, double, decimal.
* **Символьные**: char.
* **Строковые**: string.
* **Логические**: bool.

**7. Чем отличаются типы-значения и ссылочные типы**

* **Типы-значения** (value types): данные хранятся непосредственно в переменной. Например, int, float.
* **Ссылочные типы** (reference types): переменная содержит ссылку на объект в памяти. Например, string, class.

**8. Какие типы числовых переменных имеются**

* **Целые числа**: byte, sbyte, short, ushort, int, uint, long, ulong.
* **Дробные числа**: float, double, decimal.

**9. Что такое объявление и инициализация**

* **Объявление**: создание переменной с указанием типа (например, int x;).
* **Инициализация**: присвоение значения переменной (например, int x = 10;).

**10. Для чего используется упаковка и распаковка**

* **Упаковка** (boxing): преобразование значения типа-значения в объект, чтобы сохранить его в виде ссылочного типа.
* **Распаковка** (unboxing): обратный процесс — извлечение значения из объекта.

**11. Как в C# выполняется преобразование типа**

* **Неявное преобразование**: автоматическое, без потери данных (например, int → float).
* **Явное преобразование**: с использованием оператора (type) (например, (int)).

**12. Как осуществляется консольный ввод**

Для ввода данных используется метод Console.ReadLine(), который возвращает введенную строку:

csharp

string input = Console.ReadLine();

**13. Чем отличаются методы Read и ReadLine**

* Read: считывает **один символ** в виде числа.
* ReadLine: считывает **строку**.

**14. Как обеспечить вывод данных на экран**

Используйте метод Console.WriteLine():

csharp

Console.WriteLine("Привет, мир!");

**15. Для чего предназначен и как используется форматный вывод данных**

Форматный вывод позволяет вставлять переменные в строку. Пример:

csharp

int x = 5;

Console.WriteLine($"Значение x: {x}");

**16. Каковы основные правила использования стандартных функций**

* Правильно указывайте параметры функции.
* Убедитесь, что возвращаемый тип совпадает с типом переменной.
* Используйте документацию для понимания функций.

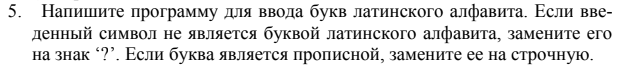
**17. Основные приемы работы в среде разработки Visual Studio.NET**

* **Как создать консольное приложение?** Выберите "Консольное приложение" при создании нового проекта.
* **Как сохранить проект с заданным именем?** Выберите "Файл" → "Сохранить как".
* **Как загрузить проект?** Нажмите "Открыть проект" и выберите файл .sln.
* **Как выполнить отладку программы?** Нажмите F5 или кнопку "Отладка".
* **Как откомпилировать и выполнить программу?** Нажмите Ctrl+Shift+B для сборки, а затем F5 для выполнения.
* **Как просмотреть результаты выполнения программы?** Результат появится в консоли или соответствующем окне.

Лабораторная работа No 2.

Объявление и определение методов

Задание 1. Создайте проект для решения задачи: На экран выводить исходные данные и результаты. В работе использовать только стандартные типы: числовые, символьный и булевский.



using System;

class Program

{

static void Main()

{

Console.Write("Введите символ: ");

char input = Console.ReadKey().KeyChar;

Console.WriteLine();

char result;

if ((input >= 'A' && input <= 'Z') || (input >= 'a' && input <= 'z'))

{

// заменяем на строчную

result = char.ToLower(input);

}

else

{

//не латинская буква заменяем на ?

result = '?';

}

Console.WriteLine($"Результат: {result}");

}

}



Задание 2. Напишите функции в виде методов. Напишите тестирующую программу с выдачей результатов на экран.



using System;

class Program

{

// Функция вычисляет sqrt(x^2 + 1)

static double Function1(double x)

{

return Math.Sqrt(x \* x + 1);

}

// Функция вычисляет sqrt(x^2 - 1), возвращает NaN, если подкоренное выражение отрицательное

static double Function2(double x)

{

if (x \* x - 1 < 0)

return double.NaN; // Возвращаем NaN если корень отрицательный

return Math.Sqrt(x \* x - 1);

}

static void Main()

{

// Ввод начального значения x

Console.Write("Введите начальное значение x: ");

double xStart = Convert.ToDouble(Console.ReadLine());

// Ввод конечного значения x

Console.Write("Введите конечное значение x: ");

double xEnd = Convert.ToDouble(Console.ReadLine());

// Ввод шага h

Console.Write("Введите шаг h: ");

double h = Convert.ToDouble(Console.ReadLine());

// Проверка на положительность шага

if (h <= 0)

{

Console.WriteLine("Ошибка: шаг должен быть положительным числом.");

return;

}

// Заголовок таблицы значений

Console.WriteLine("\nТаблица значений:");

Console.WriteLine("────────────────────────────────");

Console.WriteLine("| x | f1(x) = sqrt(x^2+1) | f2(x) = sqrt(x^2-1) |");

Console.WriteLine("────────────────────────────────");

// Цикл вычисления значений и вывода таблицы

for (double x = xStart; x <= xEnd; x += h)

{

double f1 = Function1(x); // Вычисляем f1(x)

double f2 = Function2(x); // Вычисляем f2(x)

// Печатаем строку таблицы

Console.WriteLine($"| {x,5:F2} | {f1,12:F4} | {f2,12:F4} |");

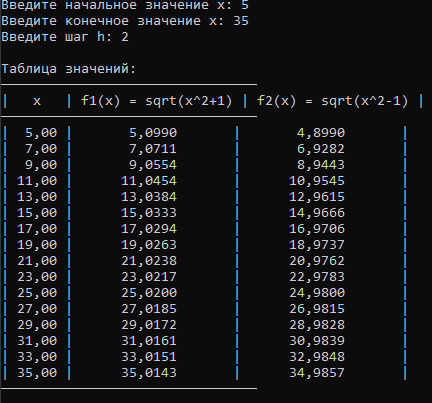
}

// Конец таблицы

Console.WriteLine("────────────────────────────────");

}

}



Контрольные вопросы:

**1. Выражением какого типа является условие в операторе if? Какие значения оно может принимать?**

Условие в операторе if является выражением типа bool. Оно может принимать значения:

* true (истина) — выполняется блок кода внутри if.
* false (ложь) — блок кода внутри if пропускается.

**2. Как работает оператор if, если отсутствует часть else?**

Если отсутствует часть else, то при выполнении условия (true) выполняется блок кода внутри if. Если условие равно false, программа просто продолжает выполнение следующего кода без выполнения блока if.

**3. В каких случаях используется оператор switch?**

Оператор switch используется для выбора одной из нескольких ветвей выполнения на основе значения выражения. Он удобен, когда есть множество условий, которые проверяются на равенство.

**4. Какого типа может быть <выражение> в операторе switch?**

Выражение в switch может быть следующих типов:

* Целочисленные (int, byte, char).
* Строковые (string).
* Перечисления (enum).

**5. В каком случае выполняется последовательность инструкций default-ветви?**

Ветвь default выполняется, если ни одно из значений в case не совпадает с выражением в switch.

**6. В чем отличие операторов while и do...while?**

* while: сначала проверяет условие, затем выполняет блок кода. Если условие false, код не выполняется ни разу.
* do...while: сначала выполняет блок кода, затем проверяет условие. Код выполняется хотя бы один раз.

**7. Что представляет собой элемент <инициализация> в операторе for?**

Элемент <инициализация> — это выражение, которое выполняется перед началом цикла. Обычно используется для объявления и инициализации управляющей переменной.

**8. Какого типа может быть элемент <условие> в цикле for?**

Элемент <условие> должен быть выражением типа bool. Оно определяет, будет ли выполняться очередная итерация цикла.

**9. Назначение управляющей переменной цикла for?**

Управляющая переменная используется для контроля количества итераций цикла. Обычно она изменяется в элементе <итерация>.

**10. Назначение управляющих операторов goto, break, continue, return**

* goto: передаёт управление на указанную метку.
* break: завершает выполнение текущего цикла или оператора switch.
* continue: пропускает оставшуюся часть текущей итерации цикла и переходит к следующей.
* return: завершает выполнение метода и возвращает значение (если метод не void).

**11. Как программируются циклические алгоритмы с явно заданным числом повторений цикла?**

Для этого используется цикл for, где:

* <инициализация> задаёт начальное значение переменной.
* <условие> определяет количество повторений.
* <итерация> изменяет значение переменной.

**12. Что представляют собой методы?**

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

**13. Как объявляется метод?**

Метод объявляется следующим образом:

csharp

[модификатор доступа] [тип возвращаемого значения] ИмяМетода([параметры])

{

// Тело метода

}

**14. Какова область действия параметров метода?**

Параметры метода доступны только внутри тела метода, где они объявлены.

**15. Как вызываются методы?**

Методы вызываются по имени, с передачей аргументов, если они требуются:

csharp

ИмяМетода(аргументы);

**16. Общие (статические) методы класса**

Статические методы принадлежат классу, а не объекту. Они вызываются через имя класса:

csharp

ИмяКласса.ИмяМетода();

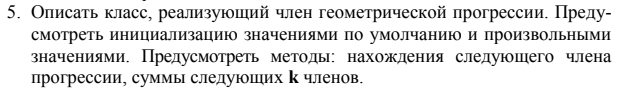
Статические методы используются для выполнения действий, которые не зависят от состояния объекта.

Лабораторная работа No 3.

Задание 1. Создайте проект, в котором опишите класс для решения задачи Вашего варианта.

Разрабатываемый класс должен содержать следующие элементы:

скрытые и открытые поля, конструкторы без параметров и с параметрами (имена некоторых полей должны совпадать с идентификаторами параметров), методы и свойства. Методы и свойства должны обеспечивать непротиворечивый и удобный интерфейс класса. В программе должна выполняться проверка всех разработанных элементов класса, вывод состояния объекта.



using System;

class GeometricProgression

{

private double firstTerm; // Первый член прогрессии

private double ratio; // Знаменатель прогрессии

// Свойства для доступа к приватным полям

public double FirstTerm

{

get { return firstTerm; }

set { firstTerm = value; }

}

public double Ratio

{

get { return ratio; }

set { ratio = value; }

}

// Конструктор по умолчанию (инициализация стандартных значений)

public GeometricProgression()

{

firstTerm = 1; // Первый член по умолчанию

ratio = 2; // Знаменатель по умолчанию

}

// Конструктор с параметрами

public GeometricProgression(double firstTerm, double ratio)

{

this.firstTerm = firstTerm; // Инициализация первого члена

this.ratio = ratio; // Инициализация знаменателя

}

// Метод вычисляет n-ый член прогрессии

public double NextTerm(int n)

{

return firstTerm \* Math.Pow(ratio, n); // Формула n-го члена

}

// Метод вычисления суммы следующих k членов прогрессии

public double SumOfNextKTerms(int k)

{

if (ratio == 1) // Обработка случая, если знаменатель равен 1

return firstTerm \* k;

// Формула суммы k членов геометрической прогрессии

return firstTerm \* (Math.Pow(ratio, k) - 1) / (ratio - 1);

}

// Метод для вывода текущих параметров прогрессии

public void PrintState()

{

Console.WriteLine($"Геометрическая прогрессия: a₁ = {firstTerm}, q = {ratio}");

}

}

class Program

{

static void Main()

{

// Создаём объект класса с параметрами по умолчанию

GeometricProgression gp1 = new GeometricProgression();

gp1.PrintState(); // Выводим параметры прогрессии

Console.WriteLine($"Следующий член (n=3): {gp1.NextTerm(3):F2}"); // n-ый член

Console.WriteLine($"Сумма следующих 4 членов: {gp1.SumOfNextKTerms(4):F2}"); // Сумма членов

Console.WriteLine("\n");

// Запрашиваем у пользователя первый член и знаменатель

Console.Write("Введите первый член прогрессии: ");

double firstTerm = Convert.ToDouble(Console.ReadLine());

Console.Write("Введите знаменатель прогрессии: ");

double ratio = Convert.ToDouble(Console.ReadLine());

// Создаём объект класса с пользовательскими значениями

GeometricProgression gp2 = new GeometricProgression(firstTerm, ratio);

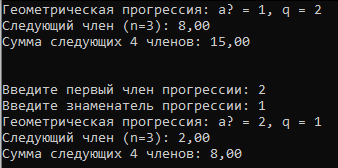
gp2.PrintState(); // Выводим параметры прогрессии

Console.WriteLine($"Следующий член (n=3): {gp2.NextTerm(3):F2}"); // n-ый член

Console.WriteLine($"Сумма следующих 4 членов: {gp2.SumOfNextKTerms(4):F2}"); // Сумма членов

}

}

  
Задание 2. Включите в проект Задания 1обработку исключений.

using System;

class GeometricProgression

{

// Закрытые поля для хранения первого члена и знаменателя прогрессии

private double firstTerm; // Первый член прогрессии

private double ratio; // Знаменатель прогрессии

// Свойства для безопасного доступа и изменения полей

public double FirstTerm

{

get { return firstTerm; } // Получение значения поля

set { firstTerm = value; } // Установка значения поля

}

public double Ratio

{

get { return ratio; } // Получение значения знаменателя

set

{

if (value == 0) // Проверка на недопустимое значение (равное 0)

throw new ArgumentException("Знаменатель прогрессии не может быть равен 0.");

ratio = value; // Установка значения знаменателя

}

}

// Конструктор по умолчанию, задаёт стандартные значения

public GeometricProgression()

{

firstTerm = 1; // Первый член прогрессии по умолчанию

ratio = 2; // Знаменатель прогрессии по умолчанию

}

// Конструктор с пользовательскими параметрами

public GeometricProgression(double firstTerm, double ratio)

{

if (ratio == 0) // Проверка на недопустимое значение знаменателя

throw new ArgumentException("Знаменатель прогрессии не может быть равен 0.");

this.firstTerm = firstTerm; // Инициализация первого члена

this.ratio = ratio; // Инициализация знаменателя

}

// Метод вычисления следующего члена прогрессии

public double NextTerm(int n)

{

if (n < 0) // Проверка на отрицательный номер члена

throw new ArgumentException("Номер члена прогрессии должен быть неотрицательным.");

return firstTerm \* Math.Pow(ratio, n); // Формула n-го члена прогрессии

}

// Метод вычисления суммы следующих k членов прогрессии

public double SumOfNextKTerms(int k)

{

if (k <= 0) // Проверка на недопустимое значение количества членов

throw new ArgumentException("Количество членов суммы должно быть положительным.");

if (ratio == 1) // Если знаменатель равен 1, сумма - это просто повторение первого члена

return firstTerm \* k;

// Формула суммы геометрической прогрессии

return firstTerm \* (Math.Pow(ratio, k) - 1) / (ratio - 1);

}

// Метод вывода текущего состояния прогрессии

public void PrintState()

{

Console.WriteLine($"Геометрическая прогрессия: a₁ = {firstTerm}, q = {ratio}");

}

}

class Program

{

static void Main()

{

try

{

// Создание объекта с параметрами по умолчанию

GeometricProgression gp1 = new GeometricProgression();

gp1.PrintState(); // Вывод текущего состояния прогрессии

Console.WriteLine($"Следующий член (n=3): {gp1.NextTerm(3):F2}"); // Вычисление n-го члена

Console.WriteLine($"Сумма следующих 4 членов: {gp1.SumOfNextKTerms(4):F2}"); // Вычисление суммы

Console.WriteLine("\n");

// Ввод данных от пользователя с обработкой ошибок

Console.Write("Введите первый член прогрессии: ");

double firstTerm = Convert.ToDouble(Console.ReadLine());

Console.Write("Введите знаменатель прогрессии: ");

double ratio = Convert.ToDouble(Console.ReadLine());

// Создание объекта с пользовательскими параметрами

GeometricProgression gp2 = new GeometricProgression(firstTerm, ratio);

gp2.PrintState(); // Вывод состояния прогрессии

Console.Write("Введите номер следующего члена прогрессии (n): ");

int n = Convert.ToInt32(Console.ReadLine());

Console.WriteLine($"Следующий член (n={n}): {gp2.NextTerm(n):F2}"); // Вычисление n-го члена

Console.Write("Введите количество членов для суммирования (k): ");

int k = Convert.ToInt32(Console.ReadLine());

Console.WriteLine($"Сумма следующих {k} членов: {gp2.SumOfNextKTerms(k):F2}"); // Вычисление суммы

}

catch (FormatException) // Обработка ошибок ввода

{

Console.WriteLine("Ошибка: введено некорректное число.");

}

catch (ArgumentException ex) // Обработка ошибок аргументов

{

Console.WriteLine($"Ошибка: {ex.Message}");

}

catch (Exception ex) // Обработка неизвестных ошибок

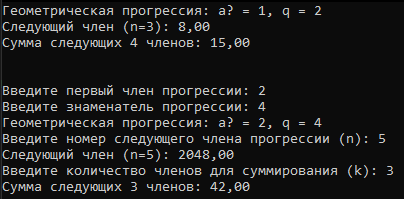
{

Console.WriteLine($"Неизвестная ошибка: {ex.Message}");

}

}

}



Контрольные вопросы:

**1. Как описываются классы в C#?**

Классы в C# описываются с использованием ключевого слова class. Они содержат данные (поля) и методы для работы с этими данными. Пример:

csharp

class MyClass

{

public int Field; // Поле

public void Method() // Метод

{

Console.WriteLine("Метод класса");

}

}

**2. Что относится к членам класса?**

Члены класса включают:

* **Поля**: переменные, хранящие данные.
* **Методы**: функции, выполняющие действия.
* **Свойства**: доступ к данным с контролем.
* **Конструкторы**: инициализация объектов.
* **События**: уведомления о действиях.
* **Индексаторы**: доступ к элементам по индексу.

**3. Что такое статические члены класса?**

Статические члены класса принадлежат самому классу, а не его объектам. Они вызываются через имя класса:

csharp

class MyClass

{

public static int StaticField;

public static void StaticMethod()

{

Console.WriteLine("Статический метод");

}

}

MyClass.StaticMethod();

**4. Данные: поля и константы**

* **Поля**: переменные, хранящие данные объекта или класса.
* **Константы**: неизменяемые значения, задаются с ключевым словом const.

**5. Спецификаторы полей и констант класса**

Спецификаторы доступа определяют видимость:

* **public**: доступен всем.
* **private**: доступен только внутри класса.
* **protected**: доступен в классе и его наследниках.
* **internal**: доступен внутри сборки.

**6. Как передаются параметры в методы?**

Параметры передаются:

* **По значению** (по умолчанию): создаётся копия значения.
* **По ссылке**: с использованием ключевого слова ref или out.

**7. Для чего предназначен параметр params?**

Параметр params позволяет передавать переменное количество аргументов одного типа:

csharp

void PrintNumbers(params int[] numbers)

{

foreach (int number in numbers)

Console.WriteLine(number);

}

PrintNumbers(1, 2, 3);

**8. Что представляет собой конструктор? Для чего он используется?**

Конструктор — это специальный метод, который вызывается при создании объекта. Он используется для инициализации полей класса.

**9. Какие бывают конструкторы?**

* **Конструктор по умолчанию**: без параметров.
* **Параметризованный конструктор**: принимает параметры.
* **Статический конструктор**: инициализирует статические члены.
* **Конструктор копирования**: создаёт копию объекта.

**10. Может ли класс не иметь конструктора?**

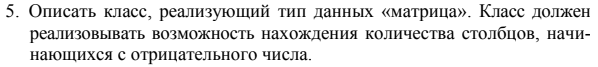
Да, если конструктор не определён явно, компилятор автоматически создаёт конструктор по умолчанию.

**11. Для чего предназначена система сбора мусора?**

Система сбора мусора (Garbage Collector) автоматически освобождает память, занятую объектами, которые больше не используются. Это помогает избежать утечек памяти и упрощает управление ресурсами.

Лабораторная работа No 4.

Задание 1. Создайте проект, в котором опишите класс для решения задачи Вашего варианта. Каждый разрабатываемый класс должен содержать следующие элементы: скрытые и открытые поля, конструкторы с параметрами и без параметров, методы, свойства, индексаторы. Класс должен реализовывать следующие операции над массивами: задание произвольной размерности массива при создании объекта; доступ к элементу по индексам с контролем выхода за пределы массива; вывод на экран элемента массива по заданному индексу и всего массива. При возникновении ошибок должны выбрасываться исключения. В программе должна выполняться проверка всех разработанных элементов класса.



using System;

class Matrix

{

private int[,] data; // Двумерный массив для хранения элементов матрицы

// Свойство для получения количества строк

public int Rows => data.GetLength(0);

// Свойство для получения количества столбцов

public int Columns => data.GetLength(1);

// Конструктор по умолчанию (создаёт матрицу 3x3)

public Matrix()

{

data = new int[3, 3];

}

// Конструктор с параметрами (создаёт матрицу указанного размера)

public Matrix(int rows, int columns)

{

if (rows <= 0 || columns <= 0) // Проверяем корректность размеров

throw new ArgumentException("Размеры матрицы должны быть положительными числами.");

data = new int[rows, columns]; // Инициализация матрицы

}

// Индексатор для доступа к элементам матрицы

public int this[int row, int col]

{

get

{

if (row < 0 || row >= Rows || col < 0 || col >= Columns) // Проверка индексов

throw new IndexOutOfRangeException("Индексы выходят за пределы матрицы.");

return data[row, col]; // Возвращаем элемент

}

set

{

if (row < 0 || row >= Rows || col < 0 || col >= Columns) // Проверка индексов

throw new IndexOutOfRangeException("Индексы выходят за пределы матрицы.");

data[row, col] = value; // Устанавливаем элемент

}

}

// Метод для вывода всей матрицы на экран

public void PrintMatrix()

{

Console.WriteLine("Матрица:");

for (int i = 0; i < Rows; i++) // Проходим по строкам

{

for (int j = 0; j < Columns; j++) // Проходим по столбцам

{

Console.Write($"{data[i, j],4} "); // Вывод элемента с форматированием

}

Console.WriteLine(); // Переход на следующую строку

}

}

// Метод для вывода одного элемента матрицы

public void PrintElement(int row, int col)

{

try

{

Console.WriteLine($"Элемент ({row}, {col}): {this[row, col]}"); // Вывод указанного элемента

}

catch (IndexOutOfRangeException ex) // Обработка выхода за пределы

{

Console.WriteLine($"Ошибка: {ex.Message}");

}

}

// Метод для подсчёта столбцов, начинающихся с отрицательного числа

public int CountColumnsWithNegativeStart()

{

int count = 0; // Счётчик столбцов

for (int j = 0; j < Columns; j++) // Проходим по всем столбцам

{

if (data[0, j] < 0) // Проверяем первый элемент столбца

count++;

}

return count; // Возвращаем количество подходящих столбцов

}

}

class Program

{

static void Main()

{

try

{

// Ввод количества строк и столбцов с клавиатуры

Console.Write("Введите количество строк: ");

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

Console.Write("Введите количество столбцов: ");

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

// Создаём матрицу с указанными размерами

Matrix matrix = new Matrix(rows, cols);

// Ввод элементов матрицы

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < cols; j++)

{

Console.Write($"Введите элемент ({i},{j}): ");

matrix[i, j] = int.Parse(Console.ReadLine());

}

}

// Вывод всей матрицы

matrix.PrintMatrix();

// Проверяем индексатор (получаем элемент)

Console.Write("\nВведите индекс строки: ");

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

Console.Write("Введите индекс столбца: ");

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

matrix.PrintElement(rowIndex, colIndex); // Вывод указанного элемента

// Подсчёт столбцов, начинающихся с отрицательного числа

Console.WriteLine($"Количество столбцов, начинающихся с отрицательного числа: {matrix.CountColumnsWithNegativeStart()}");

}

catch (FormatException) // Обработка ошибки ввода

{

Console.WriteLine("Ошибка: введено некорректное значение.");

}

catch (ArgumentException ex) // Обработка ошибки аргументов

{

Console.WriteLine($"Ошибка: {ex.Message}");

}

catch (Exception ex) // Обработка неизвестных ошибок

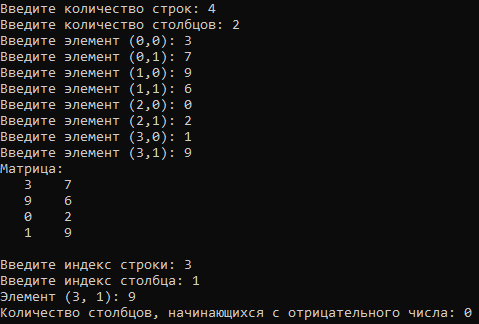
{

Console.WriteLine($"Неизвестная ошибка: {ex.Message}");

}

}

}



Контрольные вопросы:

**1. Что понимается под массивом?**

Массив — это структура данных, которая хранит набор элементов одного типа. Элементы массива имеют индексы, позволяющие к ним обращаться.

**2. Каковы возможные способы описания массивов (одномерных и многомерных)?**

* **Одномерные массивы**: описываются как int[] array = new int[5];.
* **Многомерные массивы**: описываются как int[,] matrix = new int[3, 3];.
* **Зубчатые массивы**: массив массивов, например: int[][] jaggedArray = new int[3][];.

**3. В каких случаях целесообразно описывать двумерный массив с помощью одномерных?**

Использование одномерного массива для представления двумерного может быть целесообразным, если требуется оптимизация памяти или упрощение доступа к элементам. Например, двумерный массив можно представить как одномерный с вычислением индекса: array[row \* columns + col].

**4. Какие типы допустимы для описания индексов массивов?**

Индексы массивов должны быть целочисленными типами, такими как int, long, short, byte.

**5. Какие типы могут использоваться в качестве базовых для описания массивов?**

В качестве базовых типов могут использоваться любые типы данных: примитивные (int, float, char), ссылочные (string, object), пользовательские классы и структуры.

**6. Как осуществляется ввод и вывод массивов?**

* **Ввод**: элементы массива вводятся с помощью цикла:

csharp

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

{

array[i] = int.Parse(Console.ReadLine());

}

* **Вывод**: элементы массива выводятся с помощью цикла:

csharp

foreach (int item in array)

{

Console.WriteLine(item);

}

**7. Для чего предназначен цикл foreach?**

Цикл foreach предназначен для перебора всех элементов коллекции или массива без необходимости использования индексов.

**8. Можно ли использовать цикл foreach для ввода элементов массива?**

Нет, цикл foreach не подходит для ввода элементов массива, так как он предоставляет доступ только для чтения к элементам.

**9. Как определяется базовый тип индексатора?**

Базовый тип индексатора определяется типом данных, который возвращается или устанавливается через индексатор.

**10. Что записывается в качестве имени индексатора?**

Индексатор не имеет имени. Вместо имени используется ключевое слово this.

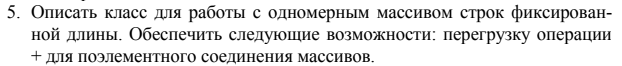
**11. Что содержит список параметров индексатора?**

Список параметров индексатора содержит типы и имена параметров, которые используются для доступа к элементам объекта. Например, this[int index].

Лабораторная работа No 5.

Задание 1. Создайте проект, в котором опишите класс для решения

задачи Вашего варианта. Каждый разрабатываемый класс должен, содержать следующие элементы: скрытые и открытые поля, конструкторы (один из них должен передавать параметром массив), перегруженные операции. В программе должна выполняться проверка всех разработанных элементов класса.



using System;

class StringArray

{

private string[] data; // Закрытый массив строк

// Свойство для получения размера массива

public int Length => data.Length;

// Конструктор по умолчанию

public StringArray(int size)

{

if (size <= 0)

throw new ArgumentException("Размер массива должен быть положительным числом.");

data = new string[size];

}

// Конструктор с передачей массива строк

public StringArray(string[] values)

{

if (values == null || values.Length == 0)

throw new ArgumentException("Массив не может быть пустым.");

data = new string[values.Length];

Array.Copy(values, data, values.Length);

}

// Индексатор для доступа к элементам массива

public string this[int index]

{

get

{

if (index < 0 || index >= Length)

throw new IndexOutOfRangeException("Индекс выходит за границы массива.");

return data[index];

}

set

{

if (index < 0 || index >= Length)

throw new IndexOutOfRangeException("Индекс выходит за границы массива.");

data[index] = value;

}

}

// Перегрузка оператора + для поэлементного соединения массивов

public static StringArray operator +(StringArray arr1, StringArray arr2)

{

if (arr1.Length != arr2.Length)

throw new InvalidOperationException("Массивы должны быть одной длины.");

string[] result = new string[arr1.Length];

for (int i = 0; i < arr1.Length; i++)

{

result[i] = arr1[i] + arr2[i]; // Поэлементное соединение

}

return new StringArray(result);

}

// Метод для заполнения массива пользователем

public void FillFromUser()

{

Console.WriteLine($"Введите {Length} строк:");

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

{

Console.Write($"Элемент {i + 1}: ");

data[i] = Console.ReadLine();

}

}

// Метод для вывода массива

public void Print()

{

Console.WriteLine("Содержимое массива:");

foreach (string s in data)

{

Console.WriteLine(s);

}

}

}

class Program

{

static void Main()

{

try

{

// Ввод размеров массивов

Console.Write("Введите размер массивов: ");

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

// Создаем два массива с пользовательским вводом

StringArray array1 = new StringArray(size);

StringArray array2 = new StringArray(size);

Console.WriteLine("\nЗаполнение первого массива:");

array1.FillFromUser();

Console.WriteLine("\nЗаполнение второго массива:");

array2.FillFromUser();

// Вывод введенных данных

Console.WriteLine("\nПервый массив:");

array1.Print();

Console.WriteLine("\nВторой массив:");

array2.Print();

// Соединяем массивы

StringArray resultArray = array1 + array2;

Console.WriteLine("\nРезультат поэлементного соединения:");

resultArray.Print();

}

catch (Exception ex)

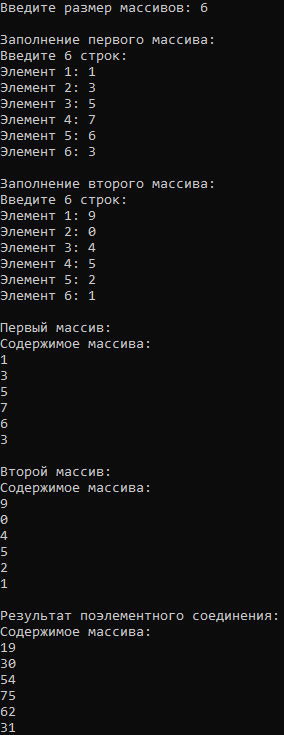
{

Console.WriteLine($"Ошибка: {ex.Message}");

}

}

}



Контрольные вопросы:

1. **Перегрузка методов** — это возможность создавать несколько методов с одинаковым именем, но разной сигнатурой (различие в типах, количестве или порядке параметров). Это позволяет использовать один метод для разных типов данных.
2. **Перегрузка операций** — это процесс определения пользовательской логики для стандартных операторов (например, +, -, ==) в пользовательских классах. Это делается с помощью ключевого слова operator.
3. **Формат описания операции класса** включает использование ключевого слова operator, указание типа возвращаемого значения и параметров. Например:

Csharp

public static Тип operator Оператор(Тип параметр) { // Логика }

1. **Операции, которые нельзя перегружать**: нельзя перегружать операции sizeof, typeof, new, is, as, а также операции присваивания.
2. **Результат перегрузки унарных операций** — это возможность изменить поведение таких операций, как +, -, !, ~, ++, -- для пользовательских классов. Например, можно определить, как объект класса будет увеличиваться или уменьшаться.
3. **Параметры бинарных операций класса**: бинарные операции принимают два параметра — один из них должен быть объектом класса, в котором перегружается операция.
4. **Перегрузка операций отношения** выполняется путем определения логики для операторов сравнения (<, >, <=, >=, ==, !=). Обычно рекомендуется перегружать их парами (например, == и !=) для согласованности.
5. **Строки в C#** — это объекты класса String, которые представляют текст. Они неизменяемы, то есть после создания их значение нельзя изменить.
6. **Операции для строк** включают конкатенацию (+), сравнение (==, !=), поиск подстроки (IndexOf, Contains), форматирование и интерполяцию строк.
7. **Создание строк**: строки можно создавать с помощью строковых литералов, конструктора String, методов форматирования или интерполяции. Например:

Csharp

string str = "Привет, мир!";

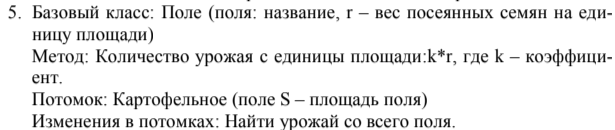
string formatted = string.Format("Число: {0}", 42);

string interpolated = $"Число: {42}";

1. **Изменение значения строки** невозможно, так как строки неизменяемы. Любая операция, которая кажется изменяющей строку, фактически создает новую строку

Лабораторная работа No 6. Наследование

Задание 1. Составить программу с одним родительским классом и потомком. Все поля должны быть закрытыми. Базовый класс должен содержать конструкторы с параметрами, методы доступа к закрытым полям, вывод полей и указанный в таблице метод. Производный класс содержит дополнения и изменения, организовать вывод новых полей потомка, при этом имена методов совпадают с именами методов базового класса. Составить тестирующую программу с выдачей результатов. Создать объекты базового и производного типов. В программе должна выполняться проверка всех разработанных элементов класса.



using System;

class Поле

{

private string название;

private double r; // вес посеянных семян на единицу площади

// Конструктор с параметрами

public Поле(string название, double r)

{

this.название = название;

this.r = r;

}

public string ПолучитьНазвание() => название;

public double ПолучитьВесСемян() => r;

public void УстановитьВесСемян(double новоеR)

{

if (новоеR > 0)

r = новоеR;

else

Console.WriteLine("Ошибка: Вес семян должен быть положительным!");

}

// Метод вычисления урожая с единицы площади

public virtual double КоличествоУрожая(double k)

{

return k \* r;

}

// Вывод информации о поле

public virtual void ВывестиИнформацию()

{

Console.WriteLine($"Поле: {название}, Вес семян: {r}");

}

}

// Производный класс "Картофельное"

class Картофельное : Поле

{

private double S;

public Картофельное(string название, double r, double S) : base(название, r)

{

this.S = S;

}

// Метод для вычисления урожая со всего поля

public override double КоличествоУрожая(double k)

{

return base.КоличествоУрожая(k) \* S;

}

// Вывод информации о картофельном поле

public override void ВывестиИнформацию()

{

base.ВывестиИнформацию();

Console.WriteLine($"Площадь поля: {S}");

}

}

class Program

{

static void Main()

{

Console.Write("Введите название поля: ");

string название = Console.ReadLine();

Console.Write("Введите вес семян на единицу площади: ");

double r = Convert.ToDouble(Console.ReadLine());

Console.Write("Введите коэффициент урожайности: ");

double k = Convert.ToDouble(Console.ReadLine());

// Создание объекта базового класса

Поле обычноеПоле = new Поле(название, r);

обычноеПоле.ВывестиИнформацию();

Console.WriteLine($"Урожай с единицы площади: {обычноеПоле.КоличествоУрожая(k)}");

Console.Write("\nВведите площадь картофельного поля: ");

double S = Convert.ToDouble(Console.ReadLine());

// Создание объекта производного класса

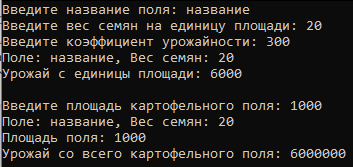
Картофельное картофельноеПоле = new Картофельное(название, r, S);

картофельноеПоле.ВывестиИнформацию();

Console.WriteLine($"Урожай со всего картофельного поля: {картофельноеПоле.КоличествоУрожая(k)}");

}

}



Контрольные вопросы:

* 1. **Принцип наследования** заключается в том, что один класс (производный) может использовать функциональность другого класса (базового). Это позволяет повторно использовать код, расширять его и добавлять новые возможности.
  2. **Члены класса, которые наследуются**: производный класс наследует все открытые (public) и защищенные (protected) члены базового класса. Приватные (private) члены не наследуются напрямую, но могут быть доступны через методы базового класса.
  3. **Защищенный доступ** (protected) позволяет членам класса быть доступными только внутри самого класса и его производных классов. Это полезно для создания логики, которая должна быть доступна только наследникам.
  4. **Вызов конструкторов базового класса** происходит автоматически при создании объекта производного класса. Если базовый класс имеет конструктор с параметрами, его нужно явно вызвать в конструкторе производного класса с помощью ключевого слова base. Например:

csharp

public class BaseClass {

public BaseClass(int value) { }

}

public class DerivedClass : BaseClass {

public DerivedClass(int value) : base(value) { }

}

* 1. **Сокрытие имен при наследовании** происходит, когда производный класс определяет член с тем же именем, что и член базового класса. Для явного сокрытия используется ключевое слово new. Например:

csharp

public class BaseClass {

public void Method() { }

}

public class DerivedClass : BaseClass {

public new void Method() { }

}

* 1. **Доступ к сокрытому члену базового класса** можно получить через объект базового класса или с помощью ключевого слова base внутри производного класса. Например:

csharp

public class BaseClass {

public void Method() { }

}

public class DerivedClass : BaseClass {

public new void Method() {

base.Method(); // Вызов метода базового класса

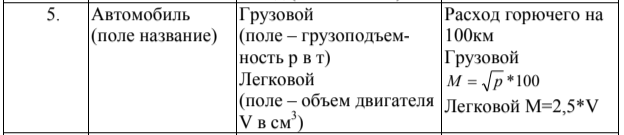
}

}

Лабораторная работа No 7.

Полиморфизм. Виртуальные методы

Задание 1. Составить программу с одним родительским классом и двумя потомками. Потомки должны содержать виртуальные функции. Создать виртуальную функцию выдачи результатов расчета методов на экран монитора с указанием названий и полей, и их значений соответствующего объекта. Составить тестирующую программу с выдачей протокола на экран монитора. При этом создать объекты базового и производных типов, используя полиморфный контейнер - массив ссылок базового класса на объекты базового и производных классов (количество объектов >=5).



using System;

using System.Collections.Generic;

// Абстрактный класс, представляющий автомобиль

abstract class Автомобиль

{

protected string название; // Поле для хранения названия автомобиля

public Автомобиль(string название) // Конструктор, устанавливающий название автомобиля

{

this.название = название;

}

public abstract double РасходТоплива(); // Абстрактный метод для расчета расхода топлива

public virtual void ВывестиИнформацию() // Виртуальный метод для отображения информации об автомобиле

{

Console.WriteLine($"Автомобиль: {название}");

}

}

// Класс, представляющий грузовой автомобиль

class Грузовой : Автомобиль

{

private double грузоподъемность; // Поле для хранения грузоподъемности автомобиля

public Грузовой(string название, double p) : base(название) // Конструктор с вызовом конструктора базового класса

{

грузоподъемность = p;

}

public override double РасходТоплива() // Реализация расчета расхода топлива для грузовых автомобилей

{

return Math.Sqrt(грузоподъемность) \* 100;

}

public override void ВывестиИнформацию() // Переопределение метода для отображения информации

{

base.ВывестиИнформацию(); // Вызов метода базового класса

Console.WriteLine($"Тип: Грузовой\nГрузоподъемность: {грузоподъемность} т");

Console.WriteLine($"Расход топлива на 100 км: {РасходТоплива()} л\n");

}

}

// Класс, представляющий легковой автомобиль

class Легковой : Автомобиль

{

private double объемДвигателя; // Поле для хранения объема двигателя

public Легковой(string название, double V) : base(название) // Конструктор с вызовом конструктора базового класса

{

объемДвигателя = V;

}

public override double РасходТоплива() // Реализация расчета расхода топлива для легковых автомобилей

{

return 2.5 \* объемДвигателя;

}

public override void ВывестиИнформацию() // Переопределение метода для отображения информации

{

base.ВывестиИнформацию(); // Вызов метода базового класса

Console.WriteLine($"Тип: Легковой\nОбъем двигателя: {объемДвигателя} см³");

Console.WriteLine($"Расход топлива на 100 км: {РасходТоплива()} л\n");

}

}

// Основной класс программы

class Program

{

static void Main()

{

List<Автомобиль> автомобили = new List<Автомобиль>(); // Список для хранения автомобилей

Console.Write("Введите количество автомобилей: ");

int n;

while (!int.TryParse(Console.ReadLine(), out n) || n <= 0) // Проверка корректности ввода количества автомобилей

{

Console.Write("Ошибка! Введите положительное число: ");

}

for (int i = 0; i < n; i++) // Цикл для добавления автомобилей в список

{

Console.Write("\nВыберите тип автомобиля (1 - Грузовой, 2 - Легковой): ");

int тип;

while (!int.TryParse(Console.ReadLine(), out тип) || (тип != 1 && тип != 2)) // Проверка ввода типа автомобиля

{

Console.Write("Ошибка! Введите 1 (Грузовой) или 2 (Легковой): ");

}

Console.Write("Введите название автомобиля: ");

string название = Console.ReadLine();

if (тип == 1) // Если выбран грузовой автомобиль

{

Console.Write("Введите грузоподъемность (т): ");

double p;

while (!double.TryParse(Console.ReadLine(), out p) || p <= 0) // Проверка ввода грузоподъемности

{

Console.Write("Ошибка! Введите положительное число: ");

}

автомобили.Add(new Грузовой(название, p)); // Добавление грузового автомобиля в список

}

else // Если выбран легковой автомобиль

{

Console.Write("Введите объем двигателя (л): ");

double v;

while (!double.TryParse(Console.ReadLine(), out v) || v <= 0) // Проверка ввода объема двигателя

{

Console.Write("Ошибка! Введите положительное число: ");

}

автомобили.Add(new Легковой(название, v)); // Добавление легкового автомобиля в список

}

}

Console.WriteLine("\n\*\*\* Информация обо всех автомобилях \*\*\*");

foreach (var авто in автомобили) // Вывод информации обо всех автомобилях

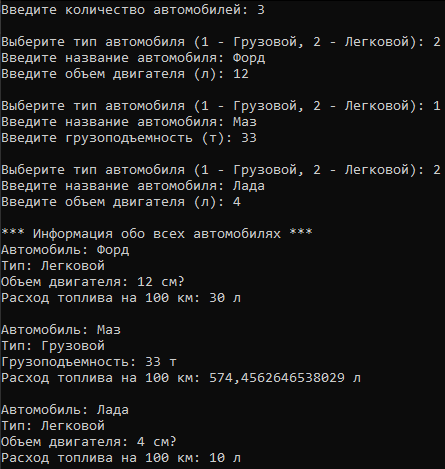
{

авто.ВывестиИнформацию();

}

}

}



Задание 2. Составить программу с абстрактным родительским классом и двумя объектами - потомками. Для этого модифицировать задание 1. Составить тестирующую программу с выдачей протокола на экран монитора. В ней нужно реализовать циклический вывод параметров объектов, используя полиморфный контейнер - массив объектов базового класса (количество объектов >=5).



using System;

using System.Collections.Generic;

// Абстрактный класс, представляющий общий тип автомобиля

abstract class Автомобиль

{

protected string название; // Поле для хранения названия автомобиля

public Автомобиль(string название) // Конструктор, инициализирующий название автомобиля

{

this.название = название;

}

public abstract double РасходТоплива(); // Абстрактный метод для расчета расхода топлива (определяется в подклассах)

public virtual void ВывестиИнформацию() // Виртуальный метод для вывода информации об автомобиле

{

Console.WriteLine($"Автомобиль: {название}");

}

}

// Класс для грузовых автомобилей, наследующий абстрактный класс Автомобиль

class Грузовой : Автомобиль

{

private double грузоподъемность; // Поле для хранения грузоподъемности грузового автомобиля

public Грузовой(string название, double p) : base(название) // Конструктор с вызовом конструктора базового класса

{

грузоподъемность = p;

}

public override double РасходТоплива() // Реализация расчета расхода топлива для грузовых автомобилей

{

return Math.Sqrt(грузоподъемность) \* 100;

}

public override void ВывестиИнформацию() // Переопределение метода для вывода информации о грузовых автомобилях

{

base.ВывестиИнформацию(); // Вызов метода базового класса

Console.WriteLine($"Тип: Грузовой\nГрузоподъемность: {грузоподъемность} т");

Console.WriteLine($"Расход топлива на 100 км: {РасходТоплива()} л\n");

}

}

// Класс для легковых автомобилей, наследующий абстрактный класс Автомобиль

class Легковой : Автомобиль

{

private double объемДвигателя; // Поле для хранения объема двигателя легкового автомобиля

public Легковой(string название, double V) : base(название) // Конструктор с вызовом конструктора базового класса

{

объемДвигателя = V;

}

public override double РасходТоплива() // Реализация расчета расхода топлива для легковых автомобилей

{

return 2.5 \* объемДвигателя;

}

public override void ВывестиИнформацию() // Переопределение метода для вывода информации о легковых автомобилях

{

base.ВывестиИнформацию(); // Вызов метода базового класса

Console.WriteLine($"Тип: Легковой\nОбъем двигателя: {объемДвигателя} л");

Console.WriteLine($"Расход топлива на 100 км: {РасходТоплива()} л\n");

}

}

// Основной класс программы для управления вводом данных и отображением результатов

class Program

{

static void Main()

{

List<Автомобиль> автомобили = new List<Автомобиль>(); // Список для хранения экземпляров автомобилей

Console.Write("Введите количество автомобилей (>=5): ");

int n;

while (!int.TryParse(Console.ReadLine(), out n) || n < 5) // Ввод и проверка корректности количества автомобилей

{

Console.Write("Ошибка! Введите число >= 5: ");

}

for (int i = 0; i < n; i++) // Цикл для ввода данных о каждом автомобиле

{

Console.Write("\nВыберите тип автомобиля (1 - Грузовой, 2 - Легковой): ");

int тип;

while (!int.TryParse(Console.ReadLine(), out тип) || (тип != 1 && тип != 2)) // Ввод и проверка типа автомобиля

{

Console.Write("Ошибка! Введите 1 (Грузовой) или 2 (Легковой): ");

}

Console.Write("Введите название автомобиля: ");

string название = Console.ReadLine(); // Ввод названия автомобиля

if (тип == 1) // Если выбран грузовой автомобиль

{

Console.Write("Введите грузоподъемность (т): ");

double p;

while (!double.TryParse(Console.ReadLine(), out p) || p <= 0) // Ввод и проверка грузоподъемности

{

Console.Write("Ошибка! Введите положительное число: ");

}

автомобили.Add(new Грузовой(название, p)); // Создание объекта грузового автомобиля и добавление в список

}

else // Если выбран легковой автомобиль

{

Console.Write("Введите объем двигателя (л): ");

double v;

while (!double.TryParse(Console.ReadLine(), out v) || v <= 0) // Ввод и проверка объема двигателя

{

Console.Write("Ошибка! Введите положительное число: ");

}

автомобили.Add(new Легковой(название, v)); // Создание объекта легкового автомобиля и добавление в список

}

}

double суммарныйРасход = 0; // Переменная для хранения суммарного расхода топлива

Console.WriteLine("\n\*\*\* Информация обо всех автомобилях \*\*\*");

foreach (var авто in автомобили) // Цикл для вывода информации о каждом автомобиле

{

авто.ВывестиИнформацию();

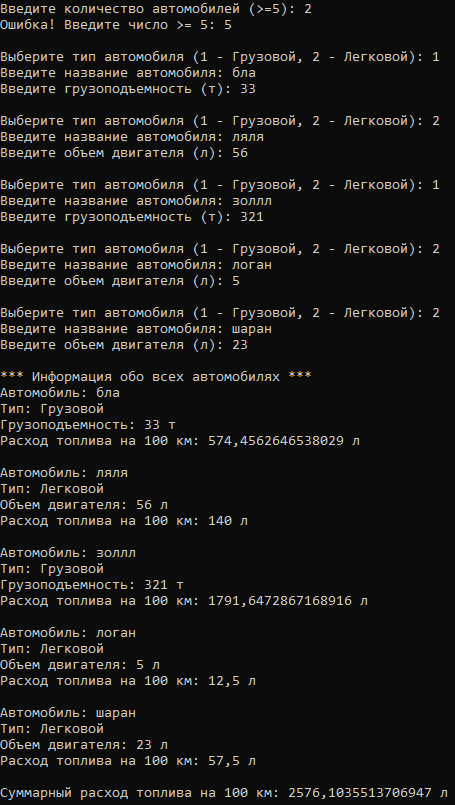
суммарныйРасход += авто.РасходТоплива(); // Суммирование расхода топлива

}

Console.WriteLine($"Суммарный расход топлива на 100 км: {суммарныйРасход} л"); // Вывод суммарного расхода топлива

}

}

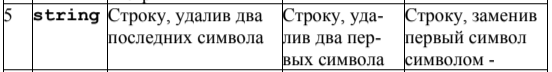


Контрольные вопросы:

* 1. **Принцип полиморфизма** заключается в способности объектов разных классов, имеющих общий базовый класс, обрабатываться единообразно. Это позволяет использовать один интерфейс для работы с объектами разных типов, что упрощает код и повышает его гибкость.
  2. **Позднее связывание** используется для вызова методов, которые определяются во время выполнения программы, а не на этапе компиляции. Это позволяет вызывать переопределенные методы производных классов через ссылки на базовый класс, реализуя полиморфизм.
  3. **Виртуальные методы** используются, когда требуется, чтобы производные классы могли переопределять поведение методов базового класса. Это полезно для создания гибкой и расширяемой архитектуры программ.
  4. **Условия для переопределения виртуального метода**:
* Метод в базовом классе должен быть объявлен с ключевым словом virtual.
* Метод в производном классе должен быть переопределен с использованием ключевого слова override.
* Сигнатура метода (имя, параметры и возвращаемый тип) должна совпадать с сигнатурой метода базового класса.
  1. **Абстрактные классы** — это классы, которые не могут быть созданы напрямую (нельзя создать их экземпляр). Они предназначены для определения общей структуры и поведения для производных классов. Абстрактные классы могут содержать как абстрактные методы (без реализации), так и методы с реализацией.
  2. **Неабстрактные методы в абстрактном классе** могут быть. Они используются для предоставления общей реализации, которая может быть унаследована производными классами. Например, абстрактный класс может содержать методы, которые не требуют переопределения в производных классах.

Лабораторная работа No 8 Интерфейсы.

Задание1



using System;

interface Ix

{

void IxF0(string str);

void IxF1();

}

interface Iy

{

void F0(string str);

void F1();

}

interface Iz

{

void F0(string str);

void F1();

}

class TestClass : Ix, Iy, Iz

{

private string w;

public TestClass(string value)

{

w = value;

}

public void IxF0(string str)

{

w = str.Length > 2 ? str.Substring(0, str.Length - 2) : "";

Console.WriteLine($"IxF0: {w}");

}

public void IxF1()

{

Console.WriteLine($"IxF1: {w}");

}

// Неявная реализация Iy (удаление двух первых символов)

public void F0(string str)

{

w = str.Length > 2 ? str.Substring(2) : "";

Console.WriteLine($"Iy.F0: {w}");

}

public void F1()

{

Console.WriteLine($"Iy.F1: {w}");

}

// Явная реализация Iz (замена первого символа на '-')

void Iz.F0(string str)

{

w = str.Length > 0 ? "-" + str.Substring(1) : "-";

Console.WriteLine($"Iz.F0: {w}");

}

void Iz.F1()

{

Console.WriteLine($"Iz.F1: {w}");

}

}

class Program

{

static void Main()

{

Console.Write("Введите строку: ");

string input = Console.ReadLine();

TestClass obj = new TestClass(input);

// Демонстрация вызова методов Ix

Console.WriteLine("\nВызов методов Ix:");

obj.IxF0(input);

obj.IxF1();

// Демонстрация вызова методов Iy (неявная реализация)

Console.WriteLine("\nВызов методов Iy:");

obj.F0(input);

obj.F1();

// Вызов методов Iz через явное приведение

Console.WriteLine("\nВызов методов Iz (явное приведение):");

Iz izRef = obj;

izRef.F0(input);

izRef.F1();

// Вызов метода через интерфейсную ссылку

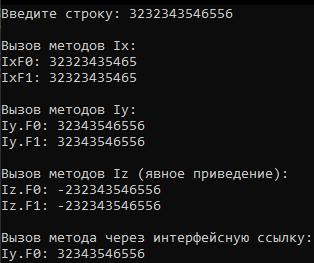
Console.WriteLine("\nВызов метода через интерфейсную ссылку:");

Iy iyRef = obj;

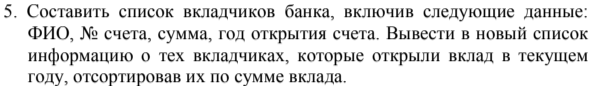
iyRef.F0(input);

}

}



Задание 2. Выполнить задания, используя для хранения экземпляров разработанных классов стандартные параметризованные коллекции. Во всех классах реализовать интерфейсы IComparable и IComparer перегрузить операции отношения для реализации сравнения объектов по указанному полю. Результат вывести на экран.



using System;

using System.Collections.Generic;

// Класс, представляющий вкладчика

class Depositor : IComparable<Depositor>

{

private string fullName; // ФИО вкладчика

private string accountNumber; // Номер счета

private decimal depositAmount; // Сумма вклада

private int openYear; // Год открытия счета

// Конструктор для инициализации всех полей

public Depositor(string fullName, string accountNumber, decimal depositAmount, int openYear)

{

this.fullName = fullName;

this.accountNumber = accountNumber;

this.depositAmount = depositAmount;

this.openYear = openYear;

}

// Свойства для получения значений полей

public string FullName => fullName;

public string AccountNumber => accountNumber;

public decimal DepositAmount => depositAmount;

public int OpenYear => openYear;

// Реализация интерфейса IComparable для сравнения по сумме вклада

public int CompareTo(Depositor other)

{

return depositAmount.CompareTo(other.depositAmount);

}

// Переопределение метода ToString для удобного отображения информации о вкладчике

public override string ToString()

{

return $"{fullName}, Счет: {accountNumber}, Вклад: {depositAmount} руб., Год открытия: {openYear}";

}

}

// Основной класс программы

class Program

{

static void Main()

{

List<Depositor> depositors = new List<Depositor>(); // Список для хранения вкладчиков

Console.Write("Введите количество вкладчиков: ");

int count;

while (!int.TryParse(Console.ReadLine(), out count) || count <= 0) // Проверка ввода количества вкладчиков

{

Console.Write("Ошибка! Введите корректное количество вкладчиков: ");

}

for (int i = 0; i < count; i++) // Цикл для ввода данных о каждом вкладчике

{

Console.WriteLine($"\nВведите данные для вкладчика {i + 1}:");

Console.Write("ФИО: ");

string fullName = Console.ReadLine(); // Ввод ФИО вкладчика

Console.Write("Номер счета: ");

string accountNumber = Console.ReadLine(); // Ввод номера счета

decimal depositAmount;

Console.Write("Сумма вклада: ");

while (!decimal.TryParse(Console.ReadLine(), out depositAmount) || depositAmount <= 0) // Проверка суммы вклада

{

Console.Write("Ошибка! Введите корректную сумму вклада: ");

}

int openYear;

Console.Write("Год открытия счета: ");

while (!int.TryParse(Console.ReadLine(), out openYear) || openYear < 1900 || openYear > DateTime.Now.Year) // Проверка года открытия

{

Console.Write("Ошибка! Введите корректный год открытия счета: ");

}

depositors.Add(new Depositor(fullName, accountNumber, depositAmount, openYear)); // Создание нового вкладчика и добавление в список

}

int currentYear = DateTime.Now.Year; // Текущий год

// Список вкладчиков, открывших счет в текущем году

List<Depositor> currentYearDepositors = depositors.FindAll(d => d.OpenYear == currentYear);

currentYearDepositors.Sort(); // Сортировка списка по сумме вклада

Console.WriteLine("\nВкладчики, открывшие вклад в текущем году (отсортированные по сумме вклада):");

if (currentYearDepositors.Count == 0) // Проверка на отсутствие таких вкладчиков

{

Console.WriteLine("Нет вкладчиков, открывших вклад в текущем году.");

}

else

{

foreach (var depositor in currentYearDepositors) // Вывод информации о каждом вкладчике

{

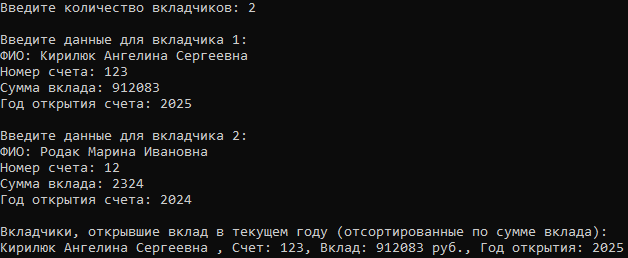
Console.WriteLine(depositor);

}

}

}

}



Контрольные вопросы:

* 1. **Описание интерфейса и его назначение**: Интерфейс в C# описывается с помощью ключевого слова interface. Он представляет собой контракт, который определяет набор методов, свойств, событий и индексаторов, которые должны быть реализованы классами или структурами, использующими этот интерфейс. Интерфейсы используются для обеспечения гибкости и возможности реализации множественного поведения, так как C# не поддерживает множественное наследование классов2.
  2. **Члены интерфейса**: Интерфейс может содержать методы, свойства, события, индексаторы, статические поля, константы и статические методы. Начиная с C# 8.0, интерфейсы могут содержать реализацию по умолчанию для своих членов3.
  3. **Спецификаторы методов, реализующих интерфейс**: Методы, реализующие интерфейс, должны быть открытыми (public) и не статическими. Это связано с тем, что члены интерфейса по умолчанию являются публичными.
  4. **Явная реализация интерфейса**: Явная реализация используется, когда необходимо скрыть реализацию интерфейса от внешнего доступа или избежать конфликта имен между членами интерфейсов. Для явной реализации используется синтаксис <ИмяИнтерфейса>.<ИмяМетода>.
  5. **Наследование интерфейсов**: Интерфейсы могут наследовать другие интерфейсы, создавая цепочку наследования. Класс, реализующий интерфейс, должен реализовать все члены как текущего интерфейса, так и его предков.
  6. **Явно реализованные методы и виртуальность**: Явно реализованные методы не могут быть объявлены виртуальными, так как они не являются частью публичного интерфейса класса.
  7. **Повторная реализация интерфейса**: Интерфейс можно повторно указать в списке предков класса, но это не приведет к повторной реализации его членов. Класс должен реализовать интерфейс только один раз.
  8. **Стандартные интерфейсы для работы с коллекциями**: Среди стандартных интерфейсов для работы с коллекциями выделяются IEnumerable, ICollection, IList, IDictionary, IReadOnlyCollection, IReadOnlyList и другие.
  9. **Различия между IComparable и IComparer**:
* IComparable используется для сравнения объектов одного типа с самим собой. Класс, реализующий этот интерфейс, должен определить метод CompareTo, который сравнивает текущий объект с другим объектом того же типа.
* IComparer используется для сравнения двух объектов. Этот интерфейс позволяет создавать пользовательскую логику сравнения, которая может быть использована, например, при сортировке коллекций.