**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ им. А.Н.Косыгина (ТЕХНОЛОГИИ, ДИЗАЙН, ИСКУССТВО)»**

**(ФГБОУ ВО «РГУ им. А.Н.Косыгина»)**

**Адаев Р.Б.**

**Технология разработки программного обеспечения.**

**Лабораторный практикум**

*Учебное пособие*

*Допущено к изданию редакционно-издательским*

*советом университета в качестве учебного пособия для подготовки бакалавров по направлениям 01.03.02 Прикладная математика и информатика, 09.03.01 Информатика и вычислительная техника, 09.03.02 Информационные системы и технологии*

Москва

ФГБОУ ВО «РГУ им.А.Н.Косыгина», 2026

УДК 519.254

А 34

А 34 Адаев Р.Б., Технология разработки программного обеспечения. Лабораторный практикум: учебное пособие – М.: ФГБОУ ВО «РГУ им. А.Н.Косыгина», 2024. – 109 с.

Рецензенты:

Исаев А.Н. - заместитель генерального директора ООО «Архитектор ИТ»;

Рыжкова Е.А. – заведующий кафедры Автоматики и промышленной электроники ФГБОУ ВО «РГУ им. А.Н.Косыгина», д-р техн.наук

Учебное пособие «Технология разработки программного обеспечения» посвящено изучению основ и продвинутых аспектов программирования на языке C#. В нем рассматриваются ключевые концепции объектно-ориентированного программирования и методы обработки данных с использованием современных инструментов и библиотек. Пособие включает в себя теоретические материалы и практические задания, охватывающие такие темы, как основы языка C# и его синтаксис, объектно-ориентированному программирование, полиморфизм, визуализации данных с использованием компонентов ListBox и DataGridView, работе с базами данных и BI-инструментами, такими как DataLens и PowerBI, использованию библиотеки LINQ для обработки и анализа данных, параллельной обработке данных и синхронизации потоков. Освоение материала пособия позволит студентам успешно применять полученные знания и навыки в профессиональной деятельности, связанной с разработкой программного обеспечения.

Учебное пособие предназначено для обучающихся по направлениям подготовки 01.03.02 Прикладная математика и информатика, 09.03.01 Информатика и вычислительная техника, 09.03.02 Информационные системы и технологии очной формы обучения и будет использовано при изучении дисциплины «Технологии программирования» и других дисциплин, связанных с программированием: «Теория множеств и алгоритмы на графах», «Информационные системы и базы данных».

УДК 519.254

Подготовлено к печати на кафедре

автоматизированных систем обработки

информации и управления

Печатается в авторской редакции

|  |  |
| --- | --- |
|  | © ФГБОУ ВО «РГУ им. А.Н. Косыгина», 2026 |
|  | © Адаев Р.Б., 2026 |

Оглавление

[Введение 11](#_Toc209005536)

[Тема 1. Основы C# 12](#_Toc209005537)

[1.1 Язык программирования C#, его преимущества 12](#_Toc209005538)

[1.2 Элементы управления 14](#_Toc209005539)

[1.2.1 Визуальные компоненты 14](#_Toc209005540)

[1.2.2 Перечень компонентов 15](#_Toc209005541)

[1.2.3 Обработка событий 17](#_Toc209005542)

[1.3 Компонент Кнопка 18](#_Toc209005543)

[1.4 Переключатели 20](#_Toc209005544)

[1.5 Компонент текстовое поле 22](#_Toc209005545)

[1.6 Цвета кнопок и форм 23](#_Toc209005546)

[1.7 Случайные числа в C# 25](#_Toc209005547)

[Пример 1.1. Работа кнопок 26](#_Toc209005548)

[Задания по теме 28](#_Toc209005549)

[Варианты заданий 28](#_Toc209005550)

[Тема 2. Основы объектно–ориентированного программирования 29](#_Toc209005551)

[2.1 Основы ООП 29](#_Toc209005552)

[2.2 Класс. Объекты класса 30](#_Toc209005553)

[2.3 Конструктор класса 31](#_Toc209005554)

[Пример 2.1. Пример класса, описывающего правильный треугольник 33](#_Toc209005555)

[2.4 Уровни доступа к членам класса 33](#_Toc209005556)

[2.5 Ссылки на объекты 34](#_Toc209005557)

[Задания по теме 35](#_Toc209005558)

[Варианты заданий 36](#_Toc209005559)

[Примеры расчета методов 39](#_Toc209005560)

[Тема 3. Статические члены классов. Свойства 40](#_Toc209005561)

[3.1 Статические члены классов 40](#_Toc209005562)

[3.2 GET/SET 42](#_Toc209005563)

[3.3 Генерация случайной даты 44](#_Toc209005564)

[Тема 4. Массивы. Преобразование типов 45](#_Toc209005565)

[4.1 Массивы в С# 45](#_Toc209005566)

[4.2 Изменение длины массива 46](#_Toc209005567)

[4.3 Обработка исключений 49](#_Toc209005568)

[4.4 Преобразование числовых типов 50](#_Toc209005569)

[4.4.1 Преобразование числовых типов в строковый тип 51](#_Toc209005570)

[4.4.2 Преобразование строкового типа в числовой тип 51](#_Toc209005571)

[4.4.3 Метод Convert 52](#_Toc209005572)

[Задание по теме 52](#_Toc209005573)

[Тема 5. Основы визуализации. Компоненты вывода данных в C# 52](#_Toc209005574)

[5.1 WPF и WinForms 52](#_Toc209005575)

[5.2 ListBox 55](#_Toc209005576)

[5.3 Datagridview 56](#_Toc209005577)

[5.4 Сравнение ListBox и Datagridview 57](#_Toc209005578)

[Пример реализации вывода данных для ListBox 58](#_Toc209005579)

[Пример реализации вывода данных для Datagridview 59](#_Toc209005580)

[5.5 Нежелательная реализация 60](#_Toc209005581)

[5.6 Datagridview и БД 61](#_Toc209005582)

[Задание по теме 67](#_Toc209005583)

[Тема 6. Структурные и ссылочные типы. Абстрактные класс и интерфейсы 67](#_Toc209005584)

[6.1 Структурные типы 67](#_Toc209005585)

[6.2 Ссылочные типы 68](#_Toc209005586)

[6.3 Сравнение работы оператора присваивания со структурными и со ссылочными типами 71](#_Toc209005587)

[6.4 Абстрактные классы 72](#_Toc209005588)

[6.5 Интерфейсы 73](#_Toc209005589)

[Пример 7.1. Интерфейс для вывода данных в консоль 74](#_Toc209005590)

[Задание по теме 76](#_Toc209005591)

[Тема 7. Наследование классов. Принципы ООП 76](#_Toc209005592)

[7.1 Наследование классов 76](#_Toc209005593)

[7.2 Виртуальные и невиртуальные методы 78](#_Toc209005594)

[7.3 Полиморфизм 79](#_Toc209005595)

[Пример 7.1. Пример классов 80](#_Toc209005596)

[7.4 Инкапсуляция 82](#_Toc209005597)

[7.5 Агрегация 84](#_Toc209005598)

[Задание по теме 86](#_Toc209005599)

[Варианты заданий 87](#_Toc209005600)

[Примеры расчета методов 93](#_Toc209005601)

[Тема 8. Рисование графических примитивов 96](#_Toc209005602)

[8.1 Рисование графических примитивов 96](#_Toc209005603)

[Пример 8.1 Рисование нескольких графических примитивов 98](#_Toc209005604)

[8.2 Компоненты-полотна 100](#_Toc209005605)

[8.3 Компонент PictureBox 101](#_Toc209005606)

[Пример 8.2 Пример использования PictureBox 103](#_Toc209005607)

[Пример 8.3 Рисование многоугольных фигур 104](#_Toc209005608)

[Пример 8.4 Рисование фигур с использованием ООП 105](#_Toc209005609)

[Пример 8.5 Рисование шестиугольника в цикле 110](#_Toc209005610)

[Задание по теме 112](#_Toc209005611)

[Тема 9. Коллекции 112](#_Toc209005612)

[9.1 Структуры данных и коллекции 112](#_Toc209005613)

[9.2 Нетипизированные и типизированные коллекции 114](#_Toc209005614)

[9.3 Списки 115](#_Toc209005615)

[Пример 9.1 Список 116](#_Toc209005616)

[9.4 Стек 117](#_Toc209005617)

[9.5 Очередь 118](#_Toc209005618)

[9.6 Преимущества и недостатки и реализация в программировании 119](#_Toc209005619)

[Пример 9.2 Очередь 120](#_Toc209005620)

[Пример 9.3 Работа с кольцевым списком 120](#_Toc209005621)

[Пример 9.4 Работа с очередью 125](#_Toc209005622)

[9.7 Словари Dictionary 129](#_Toc209005623)

[Пример 9.5 Использование Dictionary 130](#_Toc209005624)

[Задания по теме 132](#_Toc209005625)

[Тема 10. Linq 132](#_Toc209005626)

[10.1 Основы Linq 132](#_Toc209005627)

[10.2 LINQ для множеств 133](#_Toc209005628)

[10.3 Сортировка с Linq 134](#_Toc209005629)

[10.4 Методы Linq 134](#_Toc209005630)

[10.5 Лямбда-выражения 136](#_Toc209005631)

[Пример 10.1. Работа с Linq 139](#_Toc209005632)

[Задания по теме 143](#_Toc209005633)

[Тема 11. Визуализация данных 146](#_Toc209005634)

[11.1 Chart 146](#_Toc209005635)

[Пример 11.1 Столбчатая диаграмма 147](#_Toc209005636)

[11.2 ProgressBar 148](#_Toc209005637)

[Пример 11.2 ProgressBar 149](#_Toc209005638)

[Пример 11.3 PictureBox и Chart 152](#_Toc209005639)

[Задания по теме 158](#_Toc209005640)

[Варианты заданий 158](#_Toc209005641)

[Тема 12. Делегаты 159](#_Toc209005642)

[12.1 Императивное программирование 159](#_Toc209005643)

[12.2 Декларативное программирование 160](#_Toc209005644)

[12.3 Сравнение императивного и декларативного 161](#_Toc209005645)

[12.4 Перечень языков 161](#_Toc209005646)

[12.5 Делегаты 162](#_Toc209005647)

[Пример 12.1. Делегаты 162](#_Toc209005648)

[12.6 Преимущества использования делегатов 163](#_Toc209005649)

[Пример 12.2 Построение рисунков с помощью делегатов 164](#_Toc209005650)

[Задание по теме 169](#_Toc209005651)

[Тема 13. Потоки. Параллелизм 171](#_Toc209005652)

[13.1 Теория 171](#_Toc209005653)

[Пример 13.1. Потоки 172](#_Toc209005654)

[Пример 13.2 Пример со счетами 173](#_Toc209005655)

[Пример 13.3. Пример с вычислениями 177](#_Toc209005656)

[13.2 Параллелизм 180](#_Toc209005657)

[13.3 Декомпозиция 181](#_Toc209005658)

[Задания по теме 183](#_Toc209005659)

[Тема 14. Работа с файлами 185](#_Toc209005660)

[10.1 Чтение информации из файлов 185](#_Toc209005661)

[Примеры чтения текстовых файлов 185](#_Toc209005662)

[10.2 Запись строк в файл 187](#_Toc209005663)

[10.3 Методы С# для работы с файлами 188](#_Toc209005664)

[Примеры: 188](#_Toc209005665)

[Пример 10.1. Перенос всех файлов из одной папки в другую 189](#_Toc209005666)

[10.4 Подсчет времени 189](#_Toc209005667)

[1. Stopwatch (самый распространенный и простой): 189](#_Toc209005668)

[2. DateTime (менее точный, но подходит для простых случаев): 190](#_Toc209005669)

[Задания для выполнения на базовом и углубленном уровне 191](#_Toc209005670)

[Тема 11. Синхронизация потоков 191](#_Toc209005671)

[11.1 Ошибки при отсутствии синхронизации потоков 191](#_Toc209005672)

[11.1.1 Состояние гонки (Race Condition): 191](#_Toc209005673)

[11.1.2. Неживая блокировка (Deadlock): 192](#_Toc209005674)

[11.1.3. Условие конкуренции (Contention): 192](#_Toc209005675)

[11.1.4. Некорректная работа с общими ресурсами: 192](#_Toc209005676)

[11.1.5. Неопределенное поведение: 193](#_Toc209005677)

[11.2 Методы синхронизации потоков 193](#_Toc209005678)

[11.2.1. Блокировка (Lock) 193](#_Toc209005679)

[11.2.2. Mutex (Взаимная исключающая блокировка) 194](#_Toc209005680)

[11.2.3. Semaphore (Семафор) 194](#_Toc209005681)

[11.2.4. Monitor (Монитор) 194](#_Toc209005682)

[Пример 11.1. Синхронизация для программы счетов 195](#_Toc209005683)

[11.3. Сигнальные методы синхронизации потоков 198](#_Toc209005684)

[11.3.1. ManualResetEvent 199](#_Toc209005685)

[11.3.2. AutoResetEvent 199](#_Toc209005686)

[11.3.3. ManualResetEventSlim 200](#_Toc209005687)

[Задания для выполнения на базовом и углубленном уровне (ЛР 6) 201](#_Toc209005688)

[Варианты заданий 205](#_Toc209005689)

[Тема 12. Алгоритмы поиска 206](#_Toc209005690)

[12.1 Временная сложность алгоритмов 206](#_Toc209005691)

[12.2 Алгоритмы поиска 206](#_Toc209005692)

[12.2.1 Алгоритм линейного поиска 206](#_Toc209005693)

[12.2.2 Алгоритм дихотомического поиска 208](#_Toc209005694)

[13.2.3 Алгоритм интерполирующего поиска 209](#_Toc209005695)

[12.3 Решето Эратосфена 212](#_Toc209005696)

[Тема 13. Задача о максимальном потоке 213](#_Toc209005697)

[13.1 Понятие максимального потока 213](#_Toc209005698)

[13.2 Алгоритм Форда-Фалкерсона 214](#_Toc209005699)

[13.3 Алгоритм Диница 215](#_Toc209005700)

[13.4 Пример расчёта максимального потока 216](#_Toc209005701)

[13.5 Программная реализация 218](#_Toc209005702)

[Задания для выполнения на базовом и углубленном уровне (ЛР 7) 226](#_Toc209005703)

[Тема 14. Алгоритмы сортировки 227](#_Toc209005704)

[14.1 Сортировка пузырьком (Bubble Sort) 228](#_Toc209005705)

[14.2 Сортировка выбором (Selection Sort) 228](#_Toc209005706)

[14.3. Сортировка вставками (Insertion Sort) 228](#_Toc209005707)

[14.4. Сортировка слиянием (Merge Sort) 228](#_Toc209005708)

[14.5. Быстрая сортировка (Quick Sort) 229](#_Toc209005709)

[14.6. Сортировка с помощью ведер (Bucket Sort) 229](#_Toc209005710)

[14.7. Сортировка по подсчету (Counting Sort) 229](#_Toc209005711)

[14.8. Сортировка Шелла (Shell Sort) 229](#_Toc209005712)

[Задания для выполнения на базовом и углубленном уровне (ЛР 8) 230](#_Toc209005713)

[Тема 15. Триангуляция 230](#_Toc209005714)

[15.1 Триангуляция 230](#_Toc209005715)

[15.2 Триангуляция Делоне 232](#_Toc209005716)

[15.2.1 Алгоритмы построения 232](#_Toc209005717)

[15.3 Программный код триангуляции 233](#_Toc209005718)

[Ссылки: 236](#_Toc209005719)

# Введение

Данный практикум посвящен дисциплине Технологии разработки программного обеспечения магистров.

Разработан Адаевым Р. Б. в 2024 г.

В магистратуре для студентов, обучающихся на базовом уровне предусмотрено освоение тем 1-13, выполнение практических работ 1-5 и выполнение лабораторных работ 1-8

Для студентов, обучающихся на углубленном уровне предусмотрено повторение тем 1-5, освоение тем 6-15 и выполнение лабораторных работ 1-8.

Для группы ИТА-123 курс изменен, часть материалов вынесено на лекции

Тема 1. Основы С#

ЛР 1 Кнопки

Тема 2. Основы ООП

ЛР 2 ООП

Тема 3. Статич.члены, свойства

Тема 4. Массивы

ЛР 3.1 Массивы

Тема 5. Основы визуализации. Компоненты вывода данных: Листбокс, таблицы

ЛР 3.2 Компоненты вывода данных

Тема 6. Структурные и ссылочные типы, абстрактные классы

ПР 1. Отличия структурных и ссылочных типов

Тема 7. Наследование. Принципы ООП

ЛР 4. Наследование

Тема 8. Рисование графических примитивов

ПР 2. Нарисовать круг, шестиугольник

Тема 9. Коллекции

ЛР 5. Рисование структур данных (использовать 2 структуры и 2 способа рисования: 1) фигуры должны рисоваться по определенным координатам; 2) фигуры рисуются друг за другом

Необходимо выбрать две структуры данных (списки, очередь, стек) и реализовать визуализацию

Тема 10. Linq

ЛР 6. Linq

Тема 11. Визуализация данных

ЛР 7. Визуализация сложных данных

Тема 12. Делегаты

ЛР 8. Делегаты

Тема 13. Потоки, параллелизм

ЛР 9.1 Потоки

Тема 14. Работа с файлами

Тема 15. Синхронизаиия потоков

ЛР 9.2 Синхронизация потоков

Тема 16. Алгоритмы поиска

Тема 17. Алгоритм Решето Эратосфена

Тема 18. Задача о максимальном потоке

Тема 19. Триангуляция

Тема 20. Алгоритмы сортировки

Тема 21. Перегрузка операторов

Для выполнения заданий на компьютере нужно установить   среду визуального программирования либо *Microsoft Visual Studio.NET, либо SharpDevelop (более простой вариант).*

# Тема 1. Основы C#

## 1.1 Язык программирования C#, его преимущества

C# (произносится «си шарп») — это объектно–ориентированный язык программирования, разработанный Microsoft. Он является мощным и гибким инструментом для создания различных приложений, от веб–сайтов и мобильных приложений до игр и настольных программ.

Преимущества C#:

* Объектно–ориентированный: C# основан на принципах объектно–ориентированного программирования, что делает код более структурированным, понятным и удобным для расширения.
* Универсальность: C# подходит для разработки практически любых типов приложений, от веб–приложений до мобильных игр.
* Поддержка Microsoft: C# активно поддерживается Microsoft, что гарантирует его стабильность, развитие и доступность широкого спектра инструментов.
* Богатая библиотека: C# обладает богатой библиотекой классов, которая предоставляет готовые решения для выполнения различных задач.
* Совместимость с .NET: C# интегрируется с платформой .NET, которая обеспечивает доступ к широкому спектру инструментов и библиотек для разработки.

Основные понятия:

1. Типы данных: C# использует различные типы данных для представления информации, например, целые числа (int), числа с плавающей точкой (float), строки (string) и логические значения (bool).

2. Переменные: Переменные — это контейнеры для хранения данных. В C# переменные объявляются с помощью ключевого слова var и названия, например, var name = "Иван";.

3. Операторы: Операторы используются для выполнения операций над данными, например, арифметические операторы (+, –, , /), сравнительные операторы (==, !=, >, <) и логические операторы (&&, ||, !).

Условные операторы: Условные операторы (if, else, else if) используются для выполнения кода в зависимости от условия.

Циклы: Циклы (for, while, do–while) используются для выполнения кода несколько раз.

Функции: Функции — это блоки кода, которые выполняют определенную задачу. Они объявляются с помощью ключевого слова void или типа возвращаемого значения, названия и списка параметров.

Пример кода:

// Объявление переменной

string name = "Иван";

// Вывод на консоль

Console.WriteLine("Привет, " + name + "!");

// Условный оператор

if (name == "Иван")

{

Console.WriteLine("Тебя зовут Иван!");

}

else

{

Console.WriteLine("Тебя зовут не Иван!");

}

// Цикл

for (int i = 0; i < 5; i++)

{

Console.WriteLine(i);

}

## 1.2 Элементы управления

### 1.2.1 Визуальные компоненты

Визуальные компоненты (например, кнопки, текстовые поля, изображения) в приложениях на C# (WinForms) имеют свойства, определяющие их местоположение и размеры на форме. Имя компонента используется в коде для обращения к нему, а программное имя — это имя, используемое в коде, которое может отличаться от пользовательского имени.

Свойства местоположения и размеров:

Location: содержит координаты (X, Y) верхнего левого угла компонента относительно формы.

Size: содержит размеры (ширина, высота) компонента.

Left: Координата X верхнего левого угла.

Top: Координата Y верхнего левого угла.

Width: Ширина компонента.

Height:Высота компонента.

Anchor: фиксирует компонент к определённым сторонам контейнера.

В C# компоненты имеют программное и пользовательское имя.

Программное имя — это имя, используемое в коде для прямого обращения к компоненту. Обычно это свойство Name.

Пользовательское имя — это имя, которое отображается пользователю в визуальном редакторе (например, в дизайнере). Это свойство Text (для некоторых компонентов). Важно помнить, что пользовательское имя не обязательно совпадает с программным.

Как изменить свойства:

1.В конструкторе

2.В коде

Пример:

button1.Text = "Новая надпись"; // Изменение текста

3. В обработчиках событий: Изменение свойств часто происходит в обработчике событий (например, при нажатии кнопки).

### 1.2.2 Перечень компонентов

Элементы управления для отображения информации:

* Label: отображает текст.
* TextBox: позволяет пользователю вводить и редактировать текст.
* RichTextBox: обеспечивает форматированный текст, включая изображения, таблицы и другие объекты.
* PictureBox: отображает изображения.
* ListView: отображает список элементов с возможностью сортировки и фильтрации.
* TreeView: представляет иерархические данные в виде дерева.
* DataGridView: отображает данные в табличном виде.

Элементы управления для взаимодействия с пользователем:

* Button: Кнопка, которая вызывает действие при нажатии.
* CheckBox: Флажок для выбора варианта "включено" или "выключено".
* RadioButton: Радиокнопка для выбора одного варианта из нескольких.
* ComboBox: Комбинированный список для выбора значения из списка.
* ListBox: Список для выбора одного или нескольких значений.
* ProgressBar: Полоса прогресса, показывающая выполнение задачи.
* MenuStrip: Строка меню для размещения пунктов меню.
* ContextMenuStrip: Контекстное меню, появляющееся при правом клике мыши.
* TabControl: Вкладки для группировки связанных элементов управления.

Другие элементы управления:

* Panel: Панель для группировки элементов управления.
* GroupBox: Группа, которая группирует элементы управления и предоставляет заголовок.
* ToolTip: Инструментальная подсказка, которая появляется при наведении курсора мыши.
* ErrorProvider: отображает ошибки валидации для элементов управления.
* Timer: Таймер для периодического выполнения кода.

Пример использования элементов управления:

// Создать форму

Form form = new Form();

form.Text = "Пример формы";

// Создать кнопку

Button button = new Button();

button.Text = "Обычная кнопка";

button.Location = new Point(10, 10);

form.Controls.Add(button);

// Создать поле ввода текста

TextBox textBox = new TextBox();

textBox.Location = new Point(10, 50);

form.Controls.Add(textBox);

// Обработчик события нажатия на кнопку

button.Click += (sender, e) => {

MessageBox.Show($"Вы ввели: {textBox.Text}");

};

// Запустить форму

Application.Run(form);

Также можно создавать компоненты с помощью вкладки Дизайн.

### 1.2.3 Обработка событий

Обработчики событий – это функции, которые выполняются при наступлении этих событий. Существует несколько способов создания обработчиков событий. Для создания обработчика события можно, например, сначала на форме выделить компонент, с которым это событие должно быть связано, затем перейти на вкладку *Свойства*, открыть страничку событий (*Events*), выбрать имя события из списка и выполнить двойной щелчок справа от него. При этом откроется окно *Редактора кода*, в котором уже будет сгенерирована средой заготовка нужной функции, разработчику останется только вписать в фигурные скобки операторы, задающие реакцию приложения на выбранное событие.

В C#, обработка событий часто реализуется с использованием делегатов.

Обработка событий позволяет реагировать на действия, обрабатывать определённые события, таких как нажатия кнопок, перемещение мыши, изменения текста и многие другие.

Событие — это действие (или состояние), на которое пользователь может реагировать. События могут произойти в результате пользовательских действий (например, нажатие клавиш) или в результате программной логики (например, завершение загрузки данных).

Делегат — это тип, который представляет ссылку на метод. Делегаты используются для определения методов, которые могут обрабатывать события.

Подписка на события — это процесс, при котором вы указываете, какой метод должен вызываться в ответ на событие.

Простейший пример обработки события в C# (Обработка события нажатия кнопки):

Добавим обработчик события для кнопки, чтобы, когда пользователь нажимает на кнопку, текст в текстовом поле менялся.

В конструкторе формы (Form1) мы подписываемся на событие Click кнопки button1, используя оператор +=, который добавляет обработчик события Button1\_Click.

using System;

using System.Windows.Forms;

namespace EventHandlingExample

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

// Подписка на событие Click кнопки

button1.Click += new EventHandler(Button1\_Click);

// EventHandler — это стандартный делегат в C#, который принимает два параметра: объект, вызвавший событие (sender), и аргументы события (e).

}

// Обработчик события Click

private void Button1\_Click(object sender, EventArgs e)

{

textBox1.Text = "Кнопка нажата!"; // Изменяем текст в текстовом поле

}

}

}

В методе Button1\_Click мы изменяем текст в текстовом поле, когда кнопка нажата.

## 1.3 Компонент Кнопка

Одним из самых используемых управляющих компонентов является кнопка – визуальный компонент *Button.*

Данный компонент имеет довольно много свойств, но рассмотрим всего шесть из них. Отметим, что этими свойствами обладают почти все визуальные компоненты, а свойством *Name* вообще, все компоненты. Свойство *Name* задает имя компонента. По умолчанию имя первой кнопки будет *button1*, второй кнопки – *button2*, и т.д. Свойство *Name* можно изменить. Для этого надо выделить кнопку на форме и открыть вкладку *Свойства (Properties)*, найти название свойства и в поле ввода, расположенном справа от названия, ввести новое имя компонента. Свойство *Name* – это идентификатор, и строится оно по правилам построения идентификаторов.

Свойство *Text* – надпись на компоненте, оно имеет тип s*tring*, Данное свойство также можно изменить. При задании значения свойства *Text* можно использовать русские буквы и другие символы, запрещенные при определении идентификаторов.

Свойства *Left, Top*, задают координаты левого верхнего угла кнопки в системе координат компонента–контейнера. Компонент–контейнер – это компонент, на котором располагается кнопка. Каждый визуальный компонент имеет свою систему координат, в этой системе координат точка (0,0) располагается в верхнем левом углу, ось Х направлена слева на право, ось У сверху вниз. Свойства *Width, Height* задают соответственно ширину и высоту компонента. Свойства *Left, Top, Width, Height* имеют тип *int*. Значения всех перечисленных свойств, кроме *Name*, могут быть изменены во время работы программы. Например, для изменения надписи на кнопке можно использовать следующий код:

button1.Text=”Первая кнопка”;

а для изменения положения кнопки на форме следующие команды:

button1. Left= button1. Left+10;

button1. Top = button1. Top +10;

При выполнении последних двух команд, кнопка сдвинется вправо и вниз на 10 пикселей.

При обращении к свойствам формы в обработчиках событий этой формы можно использовать служебное слово *this*. Например, команды *this. Width, this. Height* будут обращениями, соответственно, к ширине и высоте текущей формы. Команда *this. Height =450;* изменит высоту формы, а строка

if (this. Height < button1. Top+25) button1. Top=100;

переместит кнопку *button1*, если она находится на нижнем краю формы.

## 1.4 Переключатели

Компоненты *CheckBox* и *RadioButton* используются, для того чтобы пользователь мог включать и выключать какие-то режимы работы приложения или индикации состояния.

*CheckBox*  называют флажком. Это прямоугольник с надписью, содержание которой определяется свойством *Text*. В прямоугольнике может находиться галочка, в этом случае, говорят “флажок поднят”, при отсутствии галочки говорят «флажок снят». Щелчок мышки поднимает или снимает флажок. Поднятие и снятие флажка обуславливается свойством *Checked*. Это свойство имеет булев тип. Значение *true* – соответствует поднятому флажку, *false* – опущенному флажку.

Компонент *RadioButton* в единственном числе очень похож на компонент *CheckBox.* Отличие заключается в том, что вместо прямоугольника на экран выводится кружочек, а вместо галочки – точка. Появление или исчезновение точки тоже связно со свойством *Checked*. Но компоненты *RadioButton* могут объединяться в функциональные группы. В группу попадают все компоненты *RadioButton*, расположенные в одном контейнере (контейнером называется компонент, на котором могут располагаться другие компоненты, например форма или панели). В группе может быть выбран только один компонент *RadioButton*. Компоненты *CheckBox*, расположенные в одном контейнере, в группу не объединяются, они работают независимо друг от друга.

RadioButton (радиокнопка) — это элемент управления, который позволяет пользователю выбрать одинвариант из нескольких.

Основные свойства:

-Text:Текст, который отображается рядом с радиокнопкой.

-Checked: Логическое значение, указывающее, выбрана ли радиокнопка.

-CheckedChange: Событие, которое возникает при изменении состояния радиокнопки.

Использование:

//Создание:

RadioButton radioButton1 = new RadioButton();

radioButton1.Text = "Вариант 1";

radioButton1.Location = new Point(10, 10);

Controls.Add(radioButton1);

//Установка значения по умолчанию:

radioButton1.Checked = true;

//Обработка события CheckedChanged:

radioButton1.CheckedChanged += (sender, e) =>

{

if (radioButton1.Checked)

{

MessageBox.Show("Выбран вариант 1");

}

};

Пример:

using System;

using System.Windows.Forms;

public class RadioButtonForm : Form

{

public RadioButtonForm()

{

// Создать радиокнопки

RadioButton radioButton1 = new RadioButton();

radioButton1.Text = "Вариант 1";

radioButton1.Location = new Point(10, 10);

Controls.Add(radioButton1);

RadioButton radioButton2 = new RadioButton();

radioButton2.Text = "Вариант 2";

radioButton2.Location = new Point(10, 40);

Controls.Add(radioButton2);

// Обработчик события для первой радиокнопки

radioButton1.CheckedChanged += (sender, e) =>

{

if (radioButton1.Checked)

{

MessageBox.Show("Выбран вариант 1");

}

};

// Обработчик события для второй радиокнопки

radioButton2.CheckedChanged += (sender, e) =>

{

if (radioButton2.Checked)

{

MessageBox.Show("Выбран вариант 2");

}

};

}

// Точка входа в приложение

static void Main(string[] args)

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new RadioButtonForm());

}

}

Результат: при запуске этого кода вы увидите форму с двумя радиокнопками. Пользователь может выбрать только один вариант. При изменении выбора появится соответствующее сообщение.

Радиокнопки могут быть сгруппированы для выбора только одного варианта. Для этого можно использовать элемент управления GroupBox.

Автоматическое отключение: если одна радиокнопка в группе выбрана, остальные автоматически отключаются.

## 1.5 Компонент текстовое поле

В Windows Forms для создания текстового поля используется компонент TextBox.

Пример кода, демонстрирующий его использование (этот код создаст простую форму с одним текстовым полем):

using System;

using System.Windows.Forms; //импорт необходимых библиотек

public class Form1 : Form

{

private TextBox textBox1; // объявляет переменную textBox1 типа TextBox, которая будет хранить ссылку на созданное текстовое поле.

public Form1()

{

// Инициализация формы

InitializeComponent();

}

private void InitializeComponent()

// вызывается для инициализации формы и добавления элементов управления

{

// Создание текстового поля

textBox1 = new TextBox();

textBox1.Location = new Point(10, 10);

// задает координаты верхнего левого угла текстового поля на форме (10 пикселей от левого края и 10 пикселей от верхнего края)

textBox1.Size = new Size(100, 20); // Размер т.поля задает координаты верхнего левого угла текстового поля на форме (10 пикселей от левого края и 10 пикселей от верхнего края)

textBox1.Text = "Введите текст"; // Начальный текст

Controls.Add(textBox1); // Добавление поля на форму

}

}

Еще о свойствах и методах:

–ReadOnly: свойство, которое делает текстовое поле доступным только для чтения (текст нельзя редактировать).

–Multiline: свойство, позволяющее вводить текст в несколько строк.

–MaxLength: свойство, которое устанавливает максимальное количество символов, которое можно ввести в текстовое поле.

–PasswordChar: свойство, которое устанавливает символ, используемый для отображения введенных символов (например, ", чтобы скрыть пароль).

–Text: свойство, которое возвращает или устанавливает текст текстового поля.

–TextChanged: событие, которое происходит, когда текст текстового поля изменяется.

Clear: метод, который очищает текст текстового поля.

## Цвета кнопок и форм

В C# (и .NET Framework) для управления цветом элементов интерфейса (кнопки, формы, элементы управления) используются свойства объектов, отвечающих за внешний вид.

Цвет фона определяет цвет заливки элемента (формы, кнопки и т.д.). Цвет переднего плана определяет цвет текста, отображаемого на элементе.

Цвет границы определяет цвет рамки вокруг элемента.

Пример работы с цветами:

using System;

using System.Drawing;

using System.Windows.Forms;

public class MyForm : Form

{

public MyForm()

{

// Установка цвета фона формы

this.BackColor = Color.LightBlue;

// Создание кнопки

Button myButton = new Button();

myButton.Text = "Нажми меня!";

myButton.Location = new Point(50, 50);

myButton.Size = new Size(100, 30);

// Установка цвета фона кнопки

myButton.BackColor = Color.Green;

// Установка цвета текста кнопки

myButton.ForeColor = Color.White;

// Добавление кнопки на форму

this.Controls.Add(myButton);

}

public static void Main()

{

Application.EnableVisualStyles();

Application.Run(new MyForm());

}

}

Класс «Color» из пространства имен «System.Drawing» предоставляет множество предопределенных цветов, таких как Color.Red, Color.Blue, Color.Green и т.д.

Свойство BackColor устанавливает цвет фона.

Свойство ForeColor устанавливает цвет текста.

Метод Color.FromArgb позволяет создавать цвет, задав его RGB–компоненты (от 0 до 255).

Компонент ColorDialog позволяет пользователю выбрать цвет из диалогового окна.

Свойство FlatStyle устанавливает стиль кнопки (стандартная, плоская, системная).

Пример использования системного цвета:

myButton.BackColor = SystemColors.Control;

Пример использования Color.FromArgb:

myButton.BackColor = Color.FromArgb(255, 128, 0); // Оранжевый цвет

Пример использования ColorDialog:

// Создание диалогового окна выбора цвета

ColorDialog colorDialog = new ColorDialog();

// Отображение диалогового окна

if (colorDialog.ShowDialog() == DialogResult.OK)

{

// Получение выбранного цвета

Color selectedColor = colorDialog.Color;

// Установка цвета кнопки

myButton.BackColor = selectedColor;

}

Для работы с графикой в C# рекомендуется использовать библиотеку System.Drawing.

## 1.7 Случайные числа в C#

В C# для генерации случайных чисел используется класс Random.

Есть несколько основных способов его использования:

1. Создание экземпляра Random:

Random random = new Random();

Этот код создает новый объект Random.

2. Генерация случайных целых чисел:

«Next()»: возвращает случайное целое число в диапазоне от 0 (включительно) до int.MaxValue (исключительно).

int randomNumber = random.Next();

Next(int maxValue): возвращает случайное целое число в диапазоне от 0 (включительно) до maxValue (исключительно).

int randomNumber = random.Next(10); // Случайное число от 0 до 9

Next(int minValue, int maxValue): возвращает случайное целое число в диапазоне от minValue (включительно) до maxValue (исключительно).

int randomNumber = random.Next(5, 15); // Случайное число от 5 до 14

3. Генерация случайных чисел с плавающей точкой:

NextDouble(): возвращает случайное число с плавающей точкой в диапазоне от 0 (включительно) до 1 (исключительно).

double randomNumber = random.NextDouble();

4. Управление «случайностью»:

Random по умолчанию использует текущее время как «seed /семя» для генерации случайных чисел. Это означает, что, если вы создадите несколько экземпляров Random в короткий промежуток времени, они могут генерировать одинаковые последовательности.

Чтобы получить более «случайные» числа, можно задать «семя» вручную:

Random random = new Random(12345); // 12345 – произвольное "семя"

## Пример 1.1. Работа кнопок

На форме 3 кнопки А, В, С.

a) По щелчку по кнопке А кнопка В перемещается вверх на 6 пикселей, достигнув верхнего края, она перескакивает вниз формы.

b) По щелчку по кнопке С кнопка В перемещается вниз на 6 пикселей, на краю формы останавливается.

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Windows.Forms;

namespace Lab1\_9

{

public partial class MainForm : Form {

public MainForm()

{

InitializeComponent();

}

void Button1Click(object sender, EventArgs e)

{

if (button2.Top-6 >0)

button2.Top=button2.Top-6;

else {

button2.Top=this.ClientSize.Height-button2.ClientSize.Height;

}}

void Button3Click(object sender, EventArgs e)

{

if (button2.Top+button2.ClientSize.Height+6 <this.ClientSize.Height)

button2.Top=button2.Top+6;

} }}

Результат выполнения программы представлен на рисунках 1-2.

Изображение выглядит как текст, снимок экрана, Прямоугольник, дисплей

Автоматически созданное описание

1. Перемещение кнопки

Изображение выглядит как текст, снимок экрана, Прямоугольник, дисплей

Автоматически созданное описание

1. Перемещение кнопки

## Задания по теме

ЛР 1. Создать оконное приложение с кнопками

## Варианты заданий

1. На форме две кнопки А, В.
   1. По щелчку по кнопке А кнопка В перемещается влево на 10 пикселей, при достижении левого края формы кнопка В перескакивает к правому краю формы.
   2. По щелчку по кнопке В кнопка А перемещается вправо на 10 пикселей, при достижении правого края формы кнопка В перескакивает к левому краю формы.
2. На форме две кнопки А, В.
   1. По щелчку по кнопке А кнопка В движется по диагонали формы (вниз на 2 пикселя и вправо на 4 пикселя), при достижении края формы кнопка В останавливается
   2. По щелчку по кнопке В кнопка А движется по диагонали формы (вверх на 2 пикселя и влево на 4 пикселя), при достижении края формы кнопка В останавливается
3. На форме три кнопки А, В, С.
   1. По щелчку по кнопке C кнопки А и В сближаются на 5 пикселей, но не перекрывают друг друга (минимальное расстояние между ними 5 пикселей)
   2. По щелчку по кнопке А кнопки получают случайный цвет
4. На форме две кнопки А, В.
   1. По щелчку по кнопке А кнопка А перемещается вверх на 10 пикселей, при достижении верхнего края текст на кнопке меняется на курсивный.
   2. По щелчку по кнопке В форма получают случайный цвет
5. На форме две кнопки А, В.
   1. По щелчку по кнопке А кнопка В перемещается вверх на 10 пикселей, но останавливается около верхнего края.
   2. По щелчку по кнопке В кнопка В перемещается вниз на 10 пикселей, но останавливается при достижении нижнего края.

# Тема 2. Основы объектно–ориентированного программирования

## 2.1 Основы ООП

Объектно–ориентированное программирование (ООП) — это парадигма программирования, которая фокусируется на представлении данных в виде объектов. Объекты обладают свойствами (данными) и методами (действиями), которые взаимодействуют между собой, чтобы моделировать реальный мир.

Основные принципы ООП:

-Абстракция: Скрытие деталей реализации от пользователя и представление только необходимой информации. Например, при работе с автомобилем не нужно знать, как устроен его двигатель, чтобы его завести.

-Инкапсуляция: Объединение данных и методов, относящихся к одному объекту, и скрытие деталей реализации от внешнего мира. Это позволяет защитить данные от некорректного доступа и упростить разработку.

Наследование: Создание новых классов (подклассов) на основе существующих (родительских). Подклассы наследуют свойства и методы родительского класса, что позволяет создавать более специализированные объекты.

-Полиморфизм: Способность объектов разных типов реагировать на одинаковый вызов метода по–разному. Это позволяет создавать более гибкий и расширяемый код.

Преимущества ООП:

* Структурированный код: ООП позволяет структурировать код в виде объектов, что делает его более понятным, читаемым и поддерживаемым.
* Повторное использование кода: Наследование и полиморфизм позволяют создавать новые объекты на основе существующих, что сокращает время разработки и повышает качество кода.
* Гибкость и расширяемость: ООП позволяет легко добавлять новые функции и объекты в программу, не затрагивая существующий код.
* Совместимость: Принципы ООП применяются во многих языках программирования, что делает код более переносимым.

## 2.2 Класс. Объекты класса

Класс в C# — это ссылочный тип. Класс имеет описание, которое должно быть в отдельном файле. Описание класса начинается со слова "class". Перед этим словом может быть указан уровень доступа класса.

Далее идет имя класса, например class A {...

Членами типов могут быть поля, предназначенные для хранения данных, и методы, события, свойства, события и вложенные типы.

Существует несколько модификаторов доступа для классов (п. 2.4), из них основные:

- public – доступ к классу возможен из любого места приложения,

- internal – доступ к классу возможен только из сборки, в которой он объявлен.

Во всех объектно-ориентированных языках четко различаются понятия «класс» и «объект». Класс – это определяемый пользователем тип данных, его можно представить себе как чертеж, по которому будут строиться сами данные, т.е. объекты (экземпляры) класса.

Поле – это переменная, объявленная внутри класса.

Метод или функция-член – это функция, объявленная внутри класса.

Для объявления объекта используется следующая конструкция:

<имя класса> имя переменной = new <имя класса>();

Именем объекта будет ссылка на этот объект.

Точка — это операция доступа к членам класса (пр.: cat.x = ...)

Ключевое слово new означает, что среде выполнения следует вызвать конструктор класса и выделить необходимое количество оперативной памяти под экземпляр класса. Поскольку класс относится к ссылочному типу, то выделении памяти производится из объемов в распоряжении среды выполнения .NET.

Команда newА(), создаст объект, т.е. среда .NET выделит под этот объект память, но такой объект не будет иметь имени и с ним невозможно будет работать.

Для того, чтобы дать объекту имя нужно создать ссылку. Команда А а; создает ссылку типа А. Ссылка похожа на указатель языка Си++, но она, в отличии от указателя, не может содержать произвольный адрес. При выполнении команды А а; создается ссылка, со значением null, т.е. она содержит некоторый стандартный адрес.

При выполнении команды

а= newА();

адрес созданного объекта класса записывается в ссылку а и а становится именем созданного объекта. Теперь мы уже можем работать с созданным объектом

## 2.3 Конструктор класса

Конструктор класса – это метод класса, который предназначен для создания объектов класса и инициализации их.

Конструктор обладает следующими свойствами:

1. Имя конструктора совпадает с именем класса.
2. Конструктор не имеет возвращаемого значения, даже типа *void*
3. Класс может содержать несколько конструкторов, отличающихся либо количеством, либо типами параметров (отличие в этом случае должно быть существенным), либо и тем и другим.
4. Конструктор без параметров называется конструктором по умолчанию.
5. Если ни один конструктор не был написан программистом, то конструктор по умолчанию генерируется компилятором.
6. Если в классе объявлен хотя бы один конструктор, то компилятор не создает своего.
7. В C# , в отличие от многих других языков, любой конструктор, если не указано иное, обнуляет значения полей.
8. Конструктор не может быть вызван явно, он вызывается автоматически при создании экземпляров класса.

Рассмотрим пример.

Определим в классе **А** два конструктора.

class A { public int x,y;

public int f() {return x+y;}

public A() { x=1; y=2; }

public A (int xx, int yy) { x=xx; y=yy;} }

При выполнении команды *A a=new A();*для создания объекта класса используется конструктор без параметров. Полям объекта *а* будут присвоены, соответственно, значения 1 и 2.

Если при создании объекта требуется использовать конструктор с параметрами, то в операторе *new*после имени класса в круглых скобках указываются значения параметров, например, *A b=new A(6,13);*

А команда *A c=new A(3);*вызовет ошибку, потому что класс *А* не содержит конструктора с одним параметром, которому можно было бы присвоить значение 3.

В языке С# существует упрощенная форма написания конструктора: при описании полей класса можно написать инициализирующие выражения.

Пример.

class В {

public int i=10;}

Все строки, содержащие инициализирующие выражения, включаются в начало всех конструкторов. В данном примере строка *i=10;* будет включена в конструктор, определяемый компилятором.

Объект исчезает, если он теряет свое имя, например, из-за того, что ссылка стала указывать на другой объект. Значит, время жизни зависит, прежде всего, от ссылки-имени. Если ссылка объявлена, как локальная переменная, и другого имени объект не приобрел, то он будет локальным.

## Пример 2.1. Пример класса, описывающего правильный треугольник

public class PrTr

{

private double side;

public double perim () {return side\*3;}

public double area () {return Math.Sqrt(3)/4\*side\*side;}

public PrTr() { }

public PrTr (double xx) { side=xx;}

}

}

## 2.4 Уровни доступа к членам класса

В С# имеется пять спецификаторов доступа: *public*, *private, protected, internal, protected internal* . Значения спецификаторов доступа следующие:

* *public* – член класса может быть использован любым методом;
* *private* – член класса может быть использован только методом того же класса,
* *protected* – член класса может быть использован только либо методом того же класса, либо методом производного класса;
* *internal* – член класса может быть использован только методом из той же сборки;
* *protected internal* – член класса может быть использован только либо методом из той же сборки или методом производного класса из любой сборки.

## 2.5 Ссылки на объекты

Ссылки на объекты в C# не содержат сами данные объекта, а указывают на место в памяти, где эти данные хранятся.

Например, можно выполнить операцию object myObject = new MyClass();

В этом случае myObject является ссылкой, хранящей адрес объекта MyClass в памяти. Создаётся новый объект MyClass и ссылка myObject указывает на него.

Когда человек копирует ссылку, копируется адрес, а не сам объект. Изменения, внесённые через одну копию ссылки, будут видны и через другие ссылки, указывающие на тот же объект.

Переменная, содержащая ссылку, может содержать значение null, если она не указывает на какой-либо объект.

C# использует ссылки, а не традиционные указатели, как в C или C++. Это делает работу с объектами более безопасной.

Когда все ссылки на объект исчезают, сборщик мусора может освободить память, занимаемую объектом.

public class MyClass

{

public int Value { get; set; }

}

public class Example

{

public static void Main(string[] args)

{

MyClass obj1 = new MyClass { Value = 10 };

MyClass obj2 = obj1; // obj2 теперь указывает на тот же объект, что и obj1

Console.WriteLine($"obj1.Value: {obj1.Value}"); // Вывод: 10

Console.WriteLine($"obj2.Value: {obj2.Value}"); // Вывод: 10

obj2.Value = 20;

Console.WriteLine($"obj1.Value: {obj1.Value}"); // Вывод: 20

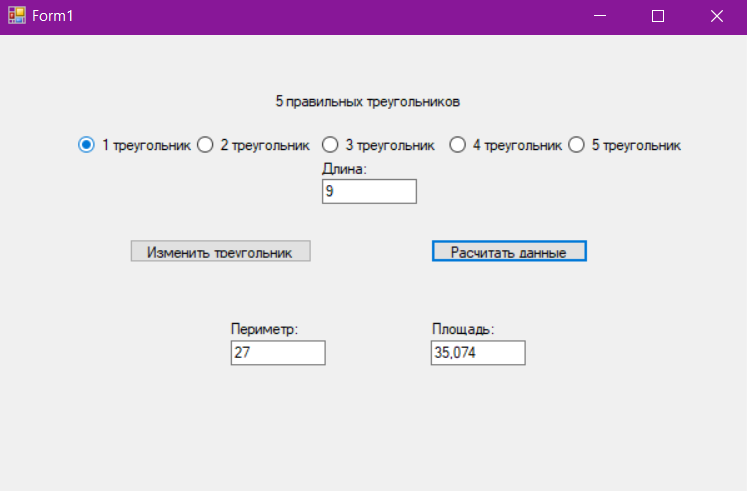
Console.WriteLine($"obj2.Value: {obj2.Value}"); // Вывод: 20

}

}

## Задания по теме

ЛР 2. Создать класс с полями и методом по индивидуальному заданию. В коде или с помощью формы создать 2-3 объекта этого класса. Создать на форме радиокнопки. Выводить на форму данные полей и метода для текущего объекта, выбираемого с помощью радиокнопок (рис. 3).



Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Содержимое, созданное искусственным интеллектом, может быть неверным.

1. Пример к ЛР 1.2

## Варианты заданий

*Общий принцип формирования. Нужен класс, 3-5 полей и 1 числовый метод.*

1. **Автомобили**. Создайте класс Автомобиль с полями марка, модель, год выпуска. Реализуйте метод расчета стоимость обслуживания, который вычисляет ежегодные расходы на обслуживание автомобиля.

2. **Электроприборы**. Создайте класс Электроприбор с полями название, мощность, цена. Реализуйте метод расчета стоимость эксплуатации, который вычисляет ежемесячные расходы на электроэнергию.

3. **Мебель**. Создайте класс Мебель с полями название, материал, цена. Реализуйте метод расчёта стоимость доставки, который вычисляет стоимость доставки мебели в зависимости от ее веса.

4. **Книги**. Создайте класс Книга с полями название, автор, цена. Реализуйте метод расчёта скидки, который вычисляет размер скидки на книгу в зависимости от ее цены.

5. **Инструменты**. Создайте класс Инструмент с полями название, вес, цена. Реализуйте метод расчёта стоимость аренды, который вычисляет стоимость аренды инструмента в зависимости от его веса.

6. **Одежда**. Создайте класс Одежда с полями название, размер, цена. Реализуйте метод расчёта скидки, который вычисляет размер скидки на одежду в зависимости от ее цены.

7. **Мобильные устройства**. Создайте класс Мобильное устройство с полями название, модель, цена. Реализуйте метод расчета стоимости аксессуаров, который вычисляет стоимость дополнительных аксессуаров для устройства в зависимости от его цены.

8. **Спортивный инвентарь.** Создайте класс Спортивный инвентарь с полями название, вес, цена. Реализуйте метод расчёта стоимости доставки, который вычисляет стоимость доставки инвентаря в зависимости от его веса.

9. **Электронные устройства**. Создайте класс Электронные устройства с полями название, модель, цена. Реализуйте метод расчёта стоимости гарантии, который вычисляет стоимость гарантийного обслуживания устройства в зависимости от его цены.

10. **Игрушки**. Создайте класс Игрушки с полями название, возрастное ограничение, цена. Реализуйте метод расчёта стоимости доставки, который вычисляет стоимость доставки игрушки в зависимости от ее веса.

11. **Канцелярские товары**. Создайте класс Канцелярские товары с полями название, тип, цена. Реализуйте метод расчёта стоимости оптовой закупки, который вычисляет скидку на товары при оптовой закупке в зависимости от их цены.

12. **Бытовая техника**. Создайте класс Бытовая техника с полями название, энергопотребление, цена. Реализуйте метод расчёта стоимости страховки, который вычисляет стоимость страхового покрытия техники в зависимости от ее цены.

13. **Музыкальные инструменты**. Создайте класс Музыкальные инструменты с полями название, материал, цена. Реализуйте метод расчёта стоимости аренды, который вычисляет стоимость аренды инструмента в зависимости от его цены.

14. **Электроника**. Создайте класс Электроника с полями: название, тип (например, телефон, ноутбук, телевизор), цена, производитель. Реализуйте метод расчёта гарантии, который определяет, сколько осталось времени гарантии в зависимости от даты покупки.

15. **Спортивная одежда**. Создайте класс Спортивная одежда с полями название, размер, цена. Реализуйте метод расчёта скидки, который вычисляет размер скидки на одежду в зависимости от ее цены.

16. **Чай**. Создайте класс Чай с полями название, сорт, цена. Реализуйте метод расчёта скидки, который вычисляет размер скидки на чай в зависимости от его цены.

17. **Товары**. Создайте класс Товар с полями название, производитель, цена, количество. Реализуйте метод расчёта стоимости, который вычисляет общую стоимость товара.

18. **Здания**. Создайте класс Здание с полями название, город, число этажей, площадь, стоимость эксплуатации здания на 1 квадратный метр. Реализуйте метод расчёта общих затрат на эксплуатацию здания.

19. **Сотрудники**. Создайте класс Сотрудник с полями Ставка за час, отработанные часы, процент налога, премия, вычеты. Реализуйте метод расчёта зарплаты к получению.

20. **Корабль**. Создайте класс Корабль с полями: Страна производитель, водоизмещение, дата ввода в эксплуатацию). Реализуйте метод износа исходя из срока службы 20 лет.

21. **Автомобили**. Создайте класс Автомобиль с полями марка, модель, расход топлива (л/100 км), пробег, объем топливного бака. Реализуйте метод расчёта расхода топлива на км.

22. **Банковские счета**. Создайте класс Банковский счет с полями баланс, процентная ставка, номер счета, имя счета, валюта. Реализуйте метод расчёта суммы начисленных процентов за год.

23. **Студенты платного отделения**. Создайте класс Студент платного отделения с полями ФИО, Номер студенческого билета, стоимость обучения за год, оплачено. Реализуйте метод, который возвращает количество денег, необходимых для завершения обучения.

24. **Поезд**. Создайте класс Поезд с полями: название, количество вагонов, производитель, дата ввода в эксплуатацию). Реализуйте метод износа исходя из срока службы 20 лет.

25. **Страны**. Создайте класс Страна с полями название, столица, население, площадь, валюта. Реализуйте метод расчёта плотности населения.

36. **Мебель**. Создайте класс Мебель с полями название, материал, цена. Реализуйте метод расчёта стоимость доставки, который вычисляет стоимость доставки мебели в зависимости от ее веса.

### Примеры расчета методов

1. Класс Автомобиль

Метод расчёта стоимости обслуживания:

если год\_выпуска < 5: return 500

elif год\_выпуска < 10: return 800

else: return 1200

2. Класс Электроприбор

Метод расчёта стоимости эксплуатации:

return мощность \* 0.5 \* 30 # 0.5 - средний тариф на электроэнергию, 30 - дни в месяце

3. Класс Мебель

Метод расчёта стоимости доставки:

return вес \* 50

5. Класс Инструмент

Метод расчёта стоимости аренды:

return вес \* 10

6. Класс Одежда

Метод расчёта скидки:

if цена <= 1000:

return 0.1

else: return 0.2

9. Класс Электронные устройства

Метод расчёта стоимости гарантии:

если цена <= 20000:

return 500

elif цена <= 50000:

return 1000

else: return 2000

15. Класс Спортивная одежда

Метод расчёта скидки:

if цена <= 1000:

return 0.1

elif цена <= 3000:

return 0.15

else:

return 0.2

# Тема 3. Статические члены классов. Свойства

## 3.1 Статические члены классов

Иногда возникает необходимость, того, что бы какое-то поле было общим для всех объектов одного типа. В этом случае используются статические поля. Статическое поле описывается с использованием слова static. Статическим может быть и метод.

В языке программирования C# статические методы представляют собой функции, которые принадлежат всему классу или структуре, а не конкретному экземпляру. Их удобно использовать в различных ситуациях, таких как вспомогательные операции или создание методов для работы с данными, которые не зависят от состояния объекта. Ниже рассмотрим, как работают статические методы в классах и структурах C#.

Статические методы объявляются с ключевым словом static. Они могут вызываться без создания экземпляра класса.

Пример:

public class MathUtils

{

// Статический метод для нахождения максимума из двух чисел

public static int Max(int a, int b)

{

return (a > b) ? a : b;

}

}

// Использование статического метода

class Program

{

static void Main()

{

int maxNumber = MathUtils.Max(10, 20);

Console.WriteLine("Max: " + maxNumber); // Вывод: Max: 20

}

}

В этом примере Max — это статический метод класса MathUtils, который можно вызывать без создания экземпляра MathUtils.

Статические методы также могут быть определены в структурах. Они работают аналогично статическим методам в классах.

Обратиться к статическому члену класса можно через имя класса. Например. В команде int i= Convert.ToInt32(textBox1.Text); используется библиотечный класс Convert, в котором собраны многие функции преобразования типов, оформленные, как статические методы. Convert.ToInt32 – это вызов статического метода, выполняющего перевод в тип int.

Со статическими членами класса могут работать нестатические методы класса, но также придуманы статические методы. Статический метод может непосредственно вызывать только статические методы и оперировать непосредственно только статическими полями. Чтобы стат.метод обратился к нестатическому полю, надо создать внутри этого метода или получать извне объект и работать с полями объекта.

Рассмотрим пример, предположим нам нужно знать сколько объектов определенного класса создано. Назовем этот класс буквой D.

class D{

private static int n;

public D(){n++;}

public static int get\_N()

{return n;}}

В классе D есть статические поле и метод. Со статическим полем может работать и нестатический метод, в данном случае с полем работает конструктор. При каждом создании объекта конструктор вызывается, и, следовательно, значение n увеличивается на единицу. Для того, чтобы узнать, сколько объектов класса D создано нужно вызвать метод get\_N(), т.е. написать команду D. get\_N();

D ob1= new D ();

D ob2=new D ();

D ob3=new D();

I=B.get\_N; //I=3

Статические члены класса являются общими для всех объектов этого класса.

Для инициализации статических полей используется статический конструктор. Он описывается с использованием слова static, имеет то же самое имя, что и класс, ничего не возвращает и не имеет параметров. Статический конструктор вызывается автоматически при первом обращении к классу. Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса. Статические конструкторы нельзя вызвать в программе вручную.

## 3.2 GET/SET

Get и Set — это свойства в C#, которые предоставляют контролируемый доступ к полям класса.

Зачем нужны Get и Set?

Есть несколько причин.

1. Инкапсуляция: они помогают спрятать внутреннюю реализацию класса, делая доступ к данным более контролируемым.

2. Проверка данных: Свойства позволяют добавить логику проверки данных перед их записью в поле.

3. Изменение логики доступа: появляется возможность изменить логику доступа к данным без изменения кода, который использует свойства.

Пример:

public class Person

{

private string \_name; // Приватное поле для имени,

// которое хранит имя человека. Оно доступно только внутри класса Person.

private int \_age; // Приватное поле для возраста

// Свойство для получения и установки имени

public string Name

{

get { return \_name; }

set

{

if (string.IsNullOrEmpty(value))

{

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

}

\_name = value;

}

}

// Свойство для получения и установки возраста

public int Age

{

get { return \_age; }

set

// Сеттер устанавливает значение поля \_name. В нем происходит проверка данных, чтобы имя не было пустым. Если проверка не проходит, генерируется исключение.

{

if (value < 0)

{

throw new ArgumentException("Возраст не может быть отрицательным.");

}

\_age = value;

}

}

}

public string Name { get; set; } это свойство с именем Name, которое предоставляет доступ к полю \_name.

Пример использования:

Использование свойств:

Person person = new Person();

// Установка имени с помощью свойства

person.Name = "Иван";

// Получение имени с помощью свойства

Console.WriteLine(person.Name); // Выведет "Иван"

// Попытка установки пустого имени

person.Name = "";

// Возникнет ошибка ArgumentException: "Имя не может быть пустым."

Внутри геттера и сеттера можно выполнять произвольные операции с полями.

## 3.3 Генерация случайной даты

Чтобы сгенерировать случайную дату в C#, вы можете использовать класс Random для получения случайного числа, а затем создать дату на основе этого числа.

Рассмотрим, как это сделать.

using System;

class Program

{

static void Main()

{

DateTime startDate = new DateTime(2000, 1, 1); // Начальная дата

DateTime endDate = DateTime.Now; // Конечная дата (текущая дата)

DateTime randomDate = GenerateRandomDate(startDate, endDate);

Console.WriteLine($"Случайная дата: {randomDate.ToShortDateString()}");

}

static DateTime GenerateRandomDate(DateTime startDate, DateTime endDate)

{

Random random = new Random();

// Генерируем случайное количество дней между двумя датами

int range = (endDate - startDate).Days; // Общее количество дней между датами

return startDate.AddDays(random.Next(range)); // Возвращаем случайную дату

}

}

# Тема 4. Массивы. Преобразование типов

## 4.1 Массивы в С#

Массив в C# — это структура данных, которая хранит упорядоченный набор элементов одного типа. Каждый элемент массива имеет свой уникальный индекс, начиная с 0.

Создание массива производится командой <тип> [] имя массива = new тип [размер]

Примеры:

// Создание массива целых чисел

int[] numbers = new int[5];

// Создание массива строк

string[] names = new string[3] { "Иван", "Мария", "Петр" };

// Создание массива с инициализацией всех элементов одним значением

bool[] flags = new bool[10] { true, true, true, true, true, true, true, true, true, true };

// Создание массива с инициализацией элементов

double[] scores = { 9.5, 8.0, 7.5, 9.0 };

Доступ к элементам массива можно производить по индексу в квадратных скобках

Примеры:

// Доступ к элементу массива по индексу

int firstNumber = numbers[0];

string secondName = names[1];

// Изменение элемента массива

numbers[2] = 10;

names[0] = "Анна";

Перебор массива:

// Цикл for

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

{

Console.WriteLine(numbers[i]);

}

// Цикл foreach

foreach (string name in names)

{

Console.WriteLine(name);

}

Основные методы массива:

* Length: возвращает количество элементов в массиве.
* IndexOf(element): возвращает индекс первого вхождения элемента в массив, или –1, если элемент не найден.
* Contains(element): возвращает true, если массив содержит элемент, или false в противном случае.
* Sort(): сортирует элементы массива по возрастанию.
* Reverse(): переворачивает порядок элементов в массиве.

C# поддерживает многомерные массивы, которые могут хранить данные в нескольких измерениях.

// Создание двумерного массива

int[,] matrix = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };

// Доступ к элементу двумерного массива

int element = matrix[0, 1]; // элемент со значением 2

## 4.2 Изменение длины массива

В C# нельзя напрямую изменять длину массива после его создания. Массив в C# — это фиксированной длины структура данных.

Если нужно динамически изменять размер массива, можно пользоваться следующими вариантами:

* List<T>: Это обобщенный класс, который предоставляет динамическую коллекцию элементов. Можно добавлять и удалять элементы из списка без необходимости изменять его размер.
* Array.Resize(): Этот метод может увеличить или уменьшить размер существующего массива, создавая новый массив и копируя в него элементы исходного. Но имейте в виду, что при уменьшении размера массива, лишние элементы будут потеряны.

Пример с List<T>:

List<int> numbers = new List<int>() { 1, 2, 3 };

// Добавление элемента

numbers.Add(4);

// Удаление элемента

numbers.RemoveAt(1);

// Получение размера списка

int count = numbers.Count;

// Перебор элементов

foreach (int number in numbers)

{

Console.WriteLine(number);

}

Пример с Array.Resize():

int[] numbers = { 1, 2, 3 };

// Увеличение размера массива

Array.Resize(ref numbers, 5); // Теперь массив вмещает 5 элементов

// Заполнение добавленных элементов

numbers[3] = 4;

numbers[4] = 5;

// Уменьшение размера массива (потеря элементов)

Array.Resize(ref numbers, 2); // Теперь массив вмещает только 2 элемента

// Вывод элементов

foreach (int number in numbers)

{

Console.WriteLine(number);

}

При использовании Array.Resize() для увеличения размера массива новые элементы будут инициализированы значением по умолчанию для соответствующего типа.

При уменьшении размера массива с помощью Array.Resize() лишние элементы будут потеряны, а размер массива будет изменен.

В C# члены массива могут быть объектами классов. Вы можете создавать массивы, элементы которых представляют собой экземпляры определенных классов.

Пример:

// Определение класса Person

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

// Создание массива объектов Person

Person [] people = new Person[3];

// Инициализация элементов массива

people[0] = new Person("Иван", 30);

people[1] = new Person("Мария", 25);

people[2] = new Person("Петр", 40);

// Вывод информации о людях

foreach (Person person in people)

{

Console.WriteLine($"Имя: {person.Name}, Возраст: {person.Age}");

}

В этом примере:

1. Создается класс Person с свойствами Name и Age.

2. Создается массив people типа Person[] с размером 3.

3. Каждый элемент массива people инициализируется новым объектом класса Person.

4. В цикле foreach перебираются элементы массива, и для каждого объекта Person выводится информация о его имени и возрасте.

В C# массив может содержать только объекты одного типа, поэтому все элементы массива people должны быть объектами класса Person.

Вы можете использовать методы и свойства объектов класса через доступ к элементам массива. Например, people[0].Name возвращает имя первого человека в массиве.

## 4.3 Обработка исключений

Обработка исключений в C# — это механизм, который позволяет программе продолжить работу, даже если во время выполнения произошла ошибка. Исключения — это объекты, которые генерируются, когда возникает ошибка, например деление на ноль, неверный формат данных или попытка доступа к несуществующему файлу.

Ключевые слова:

* try: Блок кода, в котором может возникнуть исключение.
* catch: Блок кода, который выполняется, если в блоке try возникает исключение.
* finally: Блок кода, который выполняется всегда, независимо от того, возникло исключение или нет.
* throw: используется для генерации исключения.

Пример:

using System;

public class Example

{

public static void Main(string [] args)

{

try

{

// Блок кода, в котором может возникнуть исключение

int result = Divide(10, 0);

Console.WriteLine("Результат: " + result);

}

catch (DivideByZeroException ex)

{

// Обработка исключения деления на ноль

Console.WriteLine("Ошибка: " + ex.Message);

}

finally

{

// Код, который выполняется всегда

Console.WriteLine("Блок finally выполнен.");

}

}

static int Divide(int a, int b)

{

// Выбрасываем исключение, если деление на ноль

if (b == 0)

{

throw new DivideByZeroException();

}

return a / b;

}

}

Объяснение:

В методе Main() в блоке try вызывается функция Divide(), которая может вызвать исключение DivideByZeroException.

Если происходит деление на ноль, функция Divide() генерирует исключение с помощью throw.

Блок catch перехватывает это исключение и выводит сообщение об ошибке. Блок finally выполняется независимо от того, возникло исключение или нет.

Есть возможность создавать свои собственные исключения, наследуя от Exception или одного из его подклассов.

## 4.4 Преобразование числовых типов

В C#, при работе с данными часто возникает необходимость переводить значения между различными числовыми типами (например, из int в string или из string в double). Этот раздел посвящен методам преобразования числовых типов, особенно в строковый тип и обратно. Понимание правильного преобразования и обработки потенциальных ошибок крайне важно для разработки надежных приложений.

### 4.4.1 Преобразование числовых типов в строковый тип

Для преобразования числовых типов в строковый тип в C# используется метод ToString(). Каждый числовой тип имеет свой метод ToString(), который позволяет преобразовать значение в строку.

int целоеЧисло = 123;

double вещественноеЧисло = 3.14159;

decimal десятичноеЧисло = 12345.6789m;

string строкаЦелого = целоеЧисло.ToString();

string строкаВещественного = вещественноеЧисло.ToString();

string строкаДесятичного = десятичноеЧисло.ToString();

В этом примере ToString() используется для преобразования int, double и decimal в строки. Результат будет храниться в переменных строкаЦелого, строкаВещественного и строкаДесятичного соответственно.

### 4.4.2 Преобразование строкового типа в числовой тип

Для преобразования строк, представляющих числа, в числовые типы, рекомендуется использовать метод TryParse. Этот метод позволяет проверить, может ли строка быть преобразована в числовой тип, и при успешном преобразовании сохраняет результат в переменной.

string строкаЦелого = "456";

string строкаВещественного = "7.89";

int целоеЧисло;

double вещественноеЧисло;

if (int.TryParse(строкаЦелого, out целоеЧисло))

{

Console.WriteLine($"Строка в целое число: {целоеЧисло}");

}

else

{

Console.WriteLine($"Ошибка при преобразовании строки \"{строкаЦелого}\" в целое число.");

}

if (double.TryParse(строкаВещественного, out вещественноеЧисло))

{

Console.WriteLine($"Строка в вещественное число: {вещественноеЧисло}");

}

else

{

Console.WriteLine($"Ошибка при преобразовании строки \"{строкаВещественного}\" в вещественное число.");

}

### 4.4.3 Метод Convert

В некоторых случаях можно использовать статический класс Convert для преобразования типов. Однако, он может выбрасывать исключение, если преобразование невозможно.

try

{

int числоИзСтроки = Convert.ToInt32("999");

Console.WriteLine($"Преобразование с Convert.ToInt32: {числоИзСтроки}");

}

catch (FormatException ex)

{

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

}

## Задание по теме

ЛР 3.1 Создать массив из 7-15 объектов класса из ЛР 2.

# Тема 5. Основы визуализации. Компоненты вывода данных в C#

## 5.1 WPF и WinForms

WPF (Windows Presentation Foundation) и WinForms — это два основных фреймворка для разработки графических интерфейсов пользователя (GUI) в среде .NET. Несмотря на то, что оба позволяют создавать приложения с интуитивно понятными визуальными элементами, их архитектура и функциональность существенно различаются. Эти различия влияют на сложность разработки, производительность, масштабируемость и внешний вид создаваемых приложений.

WPF основан на декларативном подходе к проектированию интерфейса. Это означает, что пользователь описывает, как должен выглядеть интерфейс, а не как его необходимо реализовать. Описание UI происходит в языке разметки XAML (Extensible Application Markup Language), который имеет XML-структуру. WPF обрабатывает рендеринг и обновление элементов интерфейса. Этот декларативный подход делает проектирование и поддержку интерфейса более управляемым и позволяет создавать визуально привлекательные и гибкие приложения.

Преимущества WPF:

* Визуальное проектирование: XAML позволяет создавать UI посредством визуального редактора, что существенно упрощает разработку.
* Встроенные механизмы макета:WPF предлагает мощные и гибкие инструменты для макетирования, такие как `Grid`, `Canvas`, `StackPanel`, позволяющие создавать сложные и адаптивные интерфейсы.
* Высокая производительность: WPF использует DirectX для рендеринга, что позволяет создавать приложения со сложной анимацией, эффектами и векторной графикой, не жертвуя производительностью.
* Отличная поддержка данных:WPF предоставляет мощные средства для связывания данных (data binding), что упрощает работу с данными и их отображение в интерфейсе.

Недостатки WPF:

* Усложнённый синтаксис:XAML может быть немного сложным для начинающих разработчиков.
* Свойственные XAML-приложениям проблемы:Ошибки в XAML могут быть сложными для локализации.
* Более высокая кривая обучения:WPF требует более глубокого понимания принципов декларативного программирования.
* WinForms основан на процедурномподходе к проектированию интерфейса. Разработчик подробно описывает поведение и визуальное состояние UI-элементов в коде. Этот подход более привычен тем, кто приходит из традиционных моделей программирования.

Преимущества WinForms:

* Простая кривая обучения:Процедурный подход WinForms проще для понимания, чем декларативный подход WPF.
* Простое макетирование (для простых приложений): для простых приложений WinForms предоставляет достаточные инструменты для создания интерфейса.
* Высокая производительность (для простых приложений):WinForms может быть более эффективным для создания приложений с простой логикой и макетом.

Недостатки WinForms:

* Усложнение с масштабируемостью:Разработка сложных или адаптивных интерфейсов может стать сложной задачей.
* Сложность макетирования: при сложном макете код становится громоздким и трудно поддерживаемым.
* Ограниченная возможность визуального проектирования: сделать сложный UI с помощью WinForms в визуальном редакторе сложно.
* Ограниченный набор визуальных элементов:WinForms не содержит таких сложных элементов, как, например, в WPF.

Ключевые различия в таблицах:

|  |  |  |
| --- | --- | --- |
| Характеристика | WPF | WinForms |
| Подход | Декларативный | Процедурный |
| Масштабируемость | Высокая | Ограниченная |
| Производительность | Высокая, особенно со сложными элементами | Низкая, при больших сложных UI |
| Дизайн | Визуальный редактор, XAML | Визуальный редактор, код |
| Адаптивность | Высокая, хорошо справляется с разрешениями | Ограниченная |

WPF – это современный подход для создания визуально богатых и сложных приложений. WinForms – это классический подход, хорошо подходящий для простых задач, но со значительными ограничениями при создании масштабируемых приложений. Выбор между WPF и WinForms зависит от специфических потребностей проекта и опыта команды разработчиков.

## 5.2 ListBox

ListBox — это компонент в C#, который представляет собой элемент управления, позволяющий отображать список элементов. Он позволяет пользователю выбирать один или несколько элементов из списка.

ListBox может быть использован для отображения списка элементов, таких как текстовые строки, изображения или пользовательские объекты. Он также поддерживает возможность множественного выбора элементов и предоставляет события для обработки изменений выбора.

ListBox имеет различные свойства, которые позволяют настраивать его внешний вид и поведение, такие как количество отображаемых элементов, режим выбора (одиночный или множественный), цвет фона, шрифт и т. д. Он также имеет методы для добавления, удаления и обновления элементов списка.

ListBox может быть использован в различных типах приложений, таких как десктопные приложения, веб–приложения и мобильные приложения, для отображения и выбора данных пользователем. Он является одним из основных элементов управления в пользовательских интерфейсах приложений, предоставляя удобный способ взаимодействия с данными.

## 5.3 Datagridview

DataGridView — это компонент в Windows Forms, который позволяет отображать и редактировать данные в виде таблицы. Он предоставляет удобный способ отображения данных из источника данных, таких как база данных, массивы или коллекции.

DataGridView имеет множество функций, включая сортировку, фильтрацию, редактирование, удаление и добавление строк. Также можно настроить внешний вид DataGridView, используя различные стили и настройки.

В C# DataGridView можно использовать для создания пользовательского интерфейса для работы с данными, таким образом улучшая пользовательский опыт и обеспечивая удобный способ просмотра и редактирования данных.

Кроме того, DataGridView поддерживает события, которые позволяют реагировать на изменения данных и пользовательские действия. Это делает его мощным инструментом для создания интерактивных пользовательских интерфейсов (рис. 4-5).

Изображение выглядит как снимок экрана, текст, диаграмма, число

Автоматически созданное описание Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

1. Иллюстрация компонентов Datagridview

Изображение выглядит как стол

Автоматически созданное описание

1. Использование таблицы для вывода адресов

## 5.4 Сравнение ListBox и Datagridview

Сравнение элементов ListBox и Datagridview демонстрируется на рис. 6–7.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

1. Демонстрация работы Datagridview

Изображение выглядит как текст, электроника, снимок экрана, программное обеспечение

Автоматически созданное описание

1. Демонстрация работы ListBox

### Пример реализации вывода данных для ListBox

//Создание класса Правильный Треугольник

public class PrTr {

private double side;

public double get\_X(){return side;}

public void set\_X(double xx){

if (xx>8 & xx<20) side=xx;

else {MessageBox.Show ("Неправильный диапазон данных");}

public double perim () {return side\*3;}

public double area () {return Math.Sqrt(3)/4\*side\*side;}

public PrTr() { }

public PrTr (double xx) { side=xx;}

} }

//Вывод данных в компонент

void output ()

{

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

{

string s = "сторона: "+mass[i].get\_X().ToString()

+", периметр:"+(int)mass[i].perim()+", площадь:"+mass[i].area();

listBox1.Items.Add (s);

}

}

//Кнопка очистки

void ButtonClearClick(object sender, EventArgs e)

{

listBox1.Items.Clear();

textBox1.Enabled=false; textBox3.Enabled=false; textBox4.Enabled=false;

}

//Заполнение массива в цикле

void MainFormLoad(object sender, EventArgs e)

{

for (int i=0; i<21;i++)

mass[i]=new PrTr (8+i/2.0);

}

//Работа с выбранным элементом

void ListBox1SelectedIndexChanged(object sender, EventArgs e)

{

textBox1.Enabled=true;

textBox3.Enabled=true;

textBox4.Enabled=true;

n=listBox1.SelectedIndex;

textBox1.Text=Convert.ToString(mass[n].get\_X());

textBox3.Text=Convert.ToString(mass[n].perim());

textBox4.Text=Convert.ToString(mass[n].area());

}

void ButtonIzmClick(object sender, EventArgs e)

{

try {

mass[n].set\_X(Convert.ToDouble(textBox1.Text));

textBox1.Text=Convert.ToString(mass[n].get\_X());

textBox3.Text=Convert.ToString(mass[n].perim());

textBox4.Text=Convert.ToString(mass[n].area());

listBox1.Items.Clear(); output();

}

catch (System.IndexOutOfRangeException)

{

MessageBox.Show ("Неверный индекс значения");

}

catch ( System.FormatException)

{

MessageBox.Show ("Неверный ввод");

} }

### Пример реализации вывода данных для Datagridview

//Кнопка очистки

void ButtonClearClick(object sender, EventArgs e)

{

dataGridView1.Rows.Clear();

textBox1.Enabled=false;

textBox3.Enabled=false;

textBox4.Enabled=false;

textBox1.Text = "";

textBox4.Text = "";

textBox3.Text = "";

}

//Работа с выбранной строкой

private void dataGridView1\_SelectionChanged(object sender, EventArgs e)

. . .

n = dataGridView1.CurrentCell.RowIndex;

. . .

//Работа кнопки изменения

void ButtonIzmClick(object sender, EventArgs e)

{

try {

mass[n].set\_X(Convert.ToDouble(textBox1.Text));

textBox1.Text=Convert.ToString(mass[n].get\_X());

textBox3.Text=Convert.ToString(mass[n].perim());

textBox4.Text=Convert.ToString(mass[n].area());

dataGridView1.RowCount = 0; output();

}

catch (System.IndexOutOfRangeException)

{ . . . }

catch ( System.FormatException)

{ . . .}

}

## 5.5 Нежелательная реализация

Не всегда пользователи грамотно используют компоненты для вывода информации (рис. 8-9).

Изображение выглядит как текст, Шрифт, число, диаграмма

Автоматически созданное описание

1. Нежелательная реализация вывода данных об объектах

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

1. Нежелательная реализация вывода данных об объектах

## 5.6 Datagridview и БД

В C# компонент Datagridview легко связать с БД (рис. 10–16).

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

1. Задачи компонента Datagridview

Изображение выглядит как текст

Автоматически созданное описание

1. Мастер настройки источника данных

Изображение выглядит как текст

Автоматически созданное описание

1. Выбор подключения для соединения

Изображение выглядит как текст

Автоматически созданное описание

1. Добавление подключения

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

1. Ввод авторизационных данных

Изображение выглядит как текст, снимок экрана, внутренний

Автоматически созданное описание

1. Проверка соединения

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

1. Выбор объектов базы данных

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

1. Таблица «Книга учёта»

## Задание по теме

ЛР 3.2

Вывести элементы из ЛР 3.1

на форму в виде:

а) нечетные варианты: Listbox

б) чётные варианты: таблицы

# Тема 6. Структурные и ссылочные типы. Абстрактные класс и интерфейсы

В C# все типы данных делятся на структурные и ссылочные. Это разделение влияет на то, как данные хранятся в памяти и как с ними взаимодействуют переменные.

## 6.1 Структурные типы

Структурные типы (например, int, double, DateTime, пользовательские структуры) хранят данные непосредственно в памяти переменной. При копировании переменной структурного типа создается копия данных.

При передаче структурного типа в функцию или метод создается копия данных, а не ссылка на исходные данные. Изменения, внесенные в переменную внутри функции, не влияют на исходную переменную.

Пример:

struct Point

{

public int X;

public int Y;

}

public class Example

{

public static void Main(string[] args)

{

Point p1 = new Point { X = 10, Y = 20 };

Point p2 = p1; // Копирование данных

p2.X = 30; // Изменение p2 не влияет на p1

Console.WriteLine($"p1.X = {p1.X}, p2.X = {p2.X}"); // Выведет: p1.X = 10, p2.X = 30

}

}

## 6.2 Ссылочные типы

Ссылочные типы (например, string, классы, массивы) хранят в переменной ссылку на данные в куче. При копировании переменной ссылочного типа создается копия ссылки, а не копирование самих данных.

При передаче ссылочного типа в функцию или метод передается ссылка на данные в куче, а не сама копия данных. Это означает, что изменения, внесенные в переменную внутри функции, могут влиять на исходную переменную.

Пример:

public class Point

{

public int X;

public int Y;

}

public class Example

{

public static void Main(string[] args)

{

Point p1 = new Point { X = 10, Y = 20 };

Point p2 = p1; // Копирование ссылки

p2.X = 30; // Изменение p2 влияет на p1

Console.WriteLine($"p1.X = {p1.X}, p2.X = {p2.X}"); // Выведет: p1.X = 30, p2.X = 30

}

}

Классы – это ссылочные типы. в C# определяются еще и структурные (иногда их называют размерными или значимыми) типы. К структурным типам относятся встроенные числовые типы, структуры и перечисления. Структуры похожи на классы, они тоже имеют описание (начинающееся со слова struct) и по описаниям создаются объекты, но они проще. Структура не может объявлять используемый по умолчанию конструктор (конструктор без параметров).

Структуры могут быть инициализированы прямо в момент их объявления. Это очень удобно для простых типов данных, которые должны иметь начальное значение.

Создавать объекты структур можно, так же как объекты классов с оператором new: H h=new H(); а можно и так H h1; h1.x=6; h1.y=9;

struct Point

{

public int X;

public int Y;

}

public class Example

{

public static void Main(string[] args)

{

Point p = new Point { X = 10, Y = 20 }; // Инициализация при объявлении

Console.WriteLine($"X: {p.X}, Y: {p.Y}");

}

}

Ссылочные типы (классы) не могут быть инициализированы при описании типа напрямую, потому что при инициализации ссылочного типа в момент объявления, на самом деле создается лишь ссылка, а не сам объект. Объект класса необходимо создать отдельно, используя оператор new.

public class Point

{

public int X;

public int Y;

}

public class Example

{

public static void Main(string[] args)

{

Point p = new Point { X = 10, Y = 20 }; //Правильная инициализация

Console.WriteLine($"X: {p.X}, Y: {p.Y}");

}

}

В обоих примерах выполняется инициализация полей типа Point. Но в первом случае (структура) инициализация происходит внутри объявления, а во втором (класс) - после использования new.

Ключевое различие структурных и ссылочных типов в хранении данных, подробнее — в таблице.

Таблица 1 — Сравнение структурного и ссылочного типов

|  |  |  |
| --- | --- | --- |
| Вопрос | Структурный | Ссылочный |
| Где располагаются? | В стеке | В управляемой куче. |
| а=в | Инф. из «в» копируется в «а», в и а-разные объекты | Информации из «в» копирована в «а» (в «в» записана ссылка (адрес на объект), этот объект переписывает в «а» и после этого а становится именем объекта В |
| Может ли использоваться в качестве базового при наследовании | Нет | Да, если нет запрета на наследование |
| Может ли тип быть производным от некоторого другого типа | Все структ.типы являются производными от библиотечного типа System.ValueType, который наследуется от System.Object. | Может |
| Может ли программист описать свой конструктор | Может, но только с параметрами | Да |
| Как создается объект | Можно создавать без new, задавая значения всех полей | С помощью new |
| Как передаются параметры методу | По значению | По ссылке |

## 6.3 Сравнение работы оператора присваивания со структурными и со ссылочными типами

Работа оператора присваивания на числовых переменных происходит, так же, как и в других языках программирования (int-структурный тип)

Пример: int x,y;// описываем переменные х и у.

y=7; // переменной у присваивается значение 7.

x=y; // значение переменной у копируется в х, при этом х и у остаются разными, независимыми друг от друга переменными.

Рассмотрим, как будет работать оператор присваивания со ссылочными типами.

Пусть *а1, а* объекты класса *А.*

*а=а1;*

а и *а1* *–* ссылки, т.е. переменные содержащие адреса, поэтому адрес из ссылки *а1* будет переписан в ссылку *а*. Если до этого, ссылки *а* и *а1* (Рис 2.а), содержали разные адреса, то после выполнения оператора присваивания, они будут содержать один и тот же адрес (Рис. 18, в). Объект *а* получит второе имя, *а* объект *а1*, свое имя потеряет, т.е. превратится в «мусор».

Изображение выглядит как круг, зарисовка, черно-белый

Автоматически созданное описание

1. Сравнение объектов

Пример расчёта

Определены типы

class AA { public int x;

public int f( int n){return x + n;}}

struct SS { public int x;

public int f( int n){return x + n;}}

Фрагмент программы

AA a1=new AA( ); AA a2=new AA( );

SS s1=new SS( ); SS s2=new SS( );

// a1 и а2 классы, s1 и s2 структуры

a1.x = 3; a2.x = 1;

s1.x = 3; s2.x = 1;

a2 = a1; // a2.x=a1.x=3;

s2 = s1; // s1.x=3;

a2.x = 2;

// Классы после приравнивания существуют как один объект, поэтому //a1.x=а2.х=2

int rex1 = a1.f(1 );

// Метод f возвращает x+1; значит для a1 результат будет 2+1=3;

s2.x = 2;

// s2 и s1 разные объекты

int rex2 = s1.f(1 );

// Метод f возвращает x+1; значит для s1 результат будет 3+1=4;

## 6.4 Абстрактные классы

Абстрактный класс это класс, который содержит хотя бы один абстрактный метод. Абстрактный метод — это метод, который не имеет реализации в классе, но должен быть переопределен в производном классе.

Абстрактные классы используются для создания общего интерфейса для производных классов и определения общей функциональности. Они не могут быть инстанцированы напрямую, но могут быть использованы в качестве базового класса для других классов.

Для создания абстрактного класса в C# используется ключевое слово abstract перед определением класса.

Пример:

**abstract class Shape**

**{**

**public abstract double Area();**

**}**

В данном примере класс Shape является абстрактным, так как содержит абстрактный метод Area(). Этот метод должен быть реализован в производных классах, которые наследуются от класса Shape.

Абстрактные классы позволяют создавать абстрактные типы данных и обеспечивают более гибкую структуру программы, позволяя определить общую функциональность для группы классов.

## 6.5 Интерфейсы

В C# интерфейс – это абстрактный тип, который определяет набор методов, свойств и событий, но не предоставляет их реализацию. Интерфейсы используются для описания контракта, который должен быть реализован классом или структурой.

Фраза "Класс А поддерживает интерфейс I" означает, что Класс А содержит все методы, определенные в интерфейсе I.

Если интерфейс I объявляет метод Метод1(), то класс А должен содержать метод Метод1() с такой же сигнатурой (название, параметры и тип возвращаемого значения).

Методы класса А должны соответствовать сигнатурам интерфейса I: Это означает, что они должны иметь одинаковое имя, количество и типы параметров, и возвращать значение того же типа, что и в интерфейсе.

Класс А не обязан содержать дополнительные методы, которые не объявлены в I: Класс может содержать свои собственные методы, помимо тех, что описаны в интерфейсе.

Пример интерфейса в C#:

**public interface IMyInterface**

**{**

**void Method1();**

**string Property1 { get; set; }**

**event EventHandler MyEvent;**

**}**

Класс или структура, которая реализует интерфейс, должна реализовать все методы, свойства и события, объявленные в интерфейсе. Например:

**public class MyClass : IMyInterface**

**{**

**public void Method1()**

**{**

**// реализация метода**

**}**

**public string Property1**

**{**

**get { return "value"; }**

**set { /\* реализация сеттера \*/ }**

**}**

**public event EventHandler MyEvent;**

**}**

Интерфейсы в C# позволяют реализовать множественное наследование, что позволяет классам имплементировать несколько интерфейсов одновременно. Использование интерфейсов также способствует созданию более гибкой и расширяемой архитектуры приложения.

## Пример 7.1. Интерфейс для вывода данных в консоль

Генерируются случайные числа и выводятся в консоль

public interface IDataProvider

{

List<double> GetData();

}

public interface IChartRenderer

{

void RenderChart(List<double> data);

}

public class RandomDataProvider : IDataProvider

{

public List<double> GetData()

{

// Логика для генерации случайных данных

var random = new Random();

var data = new List<double>();

for (int i = 0; i < 10; i++)

{

data.Add(random.NextDouble() \* 100);

}

return data;

}

}

public class ConsoleChartRenderer : IChartRenderer

{

public void RenderChart(List<double> data)

{

// Логика для вывода графика в консоль

foreach (var value in data)

{

var barLength = (int)Math.Round(value);

Console.WriteLine(new string('#', barLength));

}

}

}

public class ChartApplication

{

private readonly IDataProvider \_dataProvider;

private readonly IChartRenderer \_chartRenderer;

public ChartApplication(IDataProvider dataProvider, IChartRenderer chartRenderer)

{

\_dataProvider = dataProvider;

\_chartRenderer = chartRenderer;

}

public void Run()

{

var data = \_dataProvider.GetData();

\_chartRenderer.RenderChart(data);

}

static void Main(string[] args)

{

var provider = new RandomDataProvider();

var renderer = new ConsoleChartRenderer();

var app = new ChartApplication(provider, renderer);

app.Run();

}

}

В этом примере IDataProvider определяет метод GetData(), который генерирует случайные данные для графика. IChartRenderer определяет метод RenderChart(), который отображает график на основе полученных данных. RandomDataProvider реализует IDataProvider и генерирует случайные данные. ConsoleChartRenderer реализует IChartRenderer и выводит график в консоль. ChartApplication принимает экземпляры IDataProvider и IChartRenderer через конструктор и использует их для получения данных и вывода графика.

Можно использовать эту структуру, чтобы создать конкретные реализации IDataProvider и IChartRenderer, например, для получения данных из базы данных или для вывода графика на графическую поверхность вместо консоли.

Изображение выглядит как электроника, текст, снимок экрана, компьютер

Автоматически созданное описание

1. Результат работы программы

## Задание по теме

Определите, чему будут равны переменные х и у в ходе выполнения программы вручную и программно.

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Содержимое, созданное искусственным интеллектом, может быть неверным.

1. Практическое задание

# Тема 7. Наследование классов. Принципы ООП

В основе ООП лежат три основных понятия: инкапсуляция, наследование и полиморфизм. ООП стоит на следующей логической ступень после модульного программирования, оно углубляет понятие модульности и добавляет наследование и полиморфизм.

## 7.1 Наследование классов

Наследование является одним из фундаментальных атрибутов объектно-ориентированного программирования. Оно позволяет определить дочерний класс, который использует (наследует), расширяет или изменяет возможности родительского класса. Класс, члены которого наследуются, называется базовым классом. Класс, который наследует члены базового класса, называется производным классом.

C# и .NET поддерживают только одиночное наследование. Это означает, что каждый класс может наследовать члены только одного класса. Но зато поддерживается транзитивное наследование, которое позволяет определить иерархию наследования для набора типов. Другими словами, тип D может наследовать возможности типа C, который в свою очередь наследует от типа B, который наследует от базового класса A. Благодаря транзитивности наследования члены типа A будут доступны для типа D.

Не все члены базового класса наследуются производными классами. Следующие члены не наследуются:

* Статические конструкторы, которые инициализируют статические данные класса
* Конструкторы экземпляров, которые вызываются для создания нового экземпляра класса. Каждый класс должен определять собственные конструкторы.
* Методы завершения, которые вызываются сборщиком мусора среды выполнения для уничтожения экземпляров класса.

В C# для того, чтобы класс унаследовал другой класс, необходимо указать этот класс после двоеточия после имени наследующего класса.

Пример классов животное и собака:

**class Animal**

**{**

**public void Eat()**

**{**

**Console.WriteLine("Animal is eating");**

**}**

**}**

**class Dog : Animal**

**{**

**public void Bark()**

**{**

**Console.WriteLine("Dog is barking");**

**}**

**}**

В приведенном примере класс Dog наследует класс Animal, что означает, что класс Dog получает все поля и методы класса Animal. Теперь у объектов класса Dog будут доступны как метод Eat() из класса Animal, так и метод Bark() из самого класса Dog.

## 7.2 Виртуальные и невиртуальные методы

Виртуальные методы — это методы, которые объявлены с использованием ключевого слова virtual в базовом классе. Они могут быть переопределены в производных классах с использованием ключевого слова override. Это полезно, когда вы хотите предоставить специфическую реализацию метода в дочерних классах.

Невиртуальные методы — это обычные методы, которые не могут быть переопределены в производных классах. По умолчанию все методы являются невиртуальными, если они не объявлены как virtual, abstract или override.

Рассмотрим иерархию классов (рис. 21)

1. Пример иерархии классов.

Ссылки базового класса могут указывать на объекты производных классов.

Примеры.

B b=new B();

B d=new E();

При работе с объектами производных классов, через ссылки базовых классов, доступны, за одним исключением, только члены, определенные в базовом классе (классе ссылки). Исключение – это виртуальные методы.

Если нужно перейти от ссылки базового класса к ссылке производного класса, то надо выполнить преобразование типов.

class B

{

public void MethodB()

{

Console.WriteLine("Method of B");

}

}

class D : B

{

public void MethodD()

{

Console.WriteLine("Method of D");

}

}

class Program

{

static void Main(string[] args)

{

// Создаем объект класса D и присваиваем его ссылке базового класса B

B b = new D();

// Используем методы базового класса

b.MethodB(); // Работает

// Явное преобразование ссылки базового класса к ссылке производного класса

D d = (D)b; // Приведение типа

d.MethodD(); // Теперь можно использовать метод производного класса

}

}

## 7.3 Полиморфизм

Полиморфизм — один из основных принципов объектно-ориентированного программирования (ООП), который позволяет объектам разных классов обрабатывать данные одинаковым образом, использующим один и тот же интерфейс. Проще говоря, это способность объектов различных типов реагировать на одно и то же сообщение (метод) по-разному.

Полиморфизм позволяет писать более общий и гибкий код, уменьшая количество дублирования и улучшая читаемость. Это делает программы более понятными, мобильными и легче поддерживаемыми

Основные аспекты полиморфизма:

1. Метод скрытия: Одна и та же операция может иметь разные реализации в зависимости от типа объекта, к которому применяется. Это позволяет более гибко и удобно работать с объектами разных классов.

2. Интерфейсы: Полиморфизм часто возникает при использовании интерфейсов. Классы могут реализовывать общий интерфейс и предоставлять свои специфические реализации методов.

Примеры полиморфизма:

1. Полиморфизм через переопределение методов (override):

Когда класс наследует другой класс, он может переопределить методы базового класса, предоставляя специфичное поведение.

## Пример 7.1. Пример классов

Иерархия из нескольких классов

// Определение класса Animal (родительский класс)

public class Animal

{

// Свойства

public string Name { get; set; }

public int Age { get; set; }

// Конструктор

public Animal(string name, int age)

{

Name = name;

Age = age;

}

// Метод для вывода звука

public virtual void Speak()

{

Console.WriteLine("Звук животного");

}

}

// Определение класса Dog (подкласс Animal)

public class Dog : Animal

{

// Конструктор

public Dog(string name, int age) : base(name, age) { }

// Переопределенный метод Speak() для вывода звука собаки

public override void Speak()

{

Console.WriteLine("Гав–гав!");

}

}

// Определение класса Cat (подкласс Animal)

public class Cat : Animal

{

// Конструктор

public Cat(string name, int age) : base(name, age) { }

// Переопределенный метод Speak() для вывода звука кошки

public override void Speak()

{

Console.WriteLine("Мяу–мяу!");

}

}

// Главный класс

public class Program

{

public static void Main(string[] args)

{

// Создание объекта Dog

Dog dog = new Dog("Рекс", 3);

// Создание объекта Cat

Cat cat = new Cat("Мурка", 2);

// Вызов метода Speak() для объекта Dog

dog.Speak();

// Вызов метода Speak() для объекта Cat

cat.Speak();

Console.ReadKey(); // Пауза для просмотра результата

}

}

Объяснение кода:

Animal: Родительский класс, который определяет общие характеристики для всех животных (имя и возраст).

Dog и Cat: Подклассы, которые наследуют от Animal. Они переопределяют метод Speak() для вывода соответствующих звуков.

Speak(): Метод, который в Animal выводит "Звук животного". В подклассах он переопределен, чтобы выводить звук, соответствующий животному.

virtual: Ключевое слово, которое делает метод доступным для переопределения в подклассах.

override: Ключевое слово, которое используется для переопределения виртуальных методов в подклассах.

Результат выполнения программы:

Гав–гав!

Мяу–мяу!

Как видно, метод Speak() для объектов dog и cat выводит разные звуки, что демонстрирует полиморфизм в ООП.

## 7.4 Инкапсуляция

Это объектно-ориентированная характеристика модульности. С помощью инкапсуляции программа разделяется на объекты, выполняющие отдельные функции, детали которых скрыты от внешнего мира.

Инкапсуляция позволяет создавать сокрытые объекты (рисунок 22).

Изображение выглядит как диаграмма, текст, Технический чертеж, План

Автоматически созданное описание

1. Интерфейс

Все знают, что делает программа лишь по тому, что представлено в его интерфейсе. Интерфейс, по сути, реализует контакт с внешним миром, в нем определено, какие запросы внешние объекты могут посылать данному объекту, но не говориться о том, как объект будет выполнять эти запросы. Интерфейс – это пульт управления объектом.

При хорошей инкапсуляции реализация объекта полностью скрыта от внешнего мира, и ее можно в любой момент изменить, внешние объекты этого не почувствуют. Например, для вывода на экран рисунка объекта сначала может быть использована декартова система координат, а затем по какой-то причине было решено перейти к полярной системе координат, если при этом не изменить интерфейса, то внешний мир об этом даже не догадается. Изменение интерфейса приведет к необходимости изменений во всех фрагментах программы, которые этот объект используют.

Какие программные средства позволяют реализовать инкапсуляцию? Уровни доступа. Интерфейс образуют общедоступные члены.

Поскольку данные хранятся в полях, то их рекомендуют делать закрытыми, тогда объекты смогут полностью контролировать доступ к ним и смогут защититься от непредвиденного и деструктивного использования.

Инкапсуляция делает программу более гибкой, так как соседние объекты не обязаны учитывать реализацию друг друга. Хорошо инкапсулированный объект легко использовать в других программах, т.е. повышается повторное использование кода.

Сокрытие реализации объекта способствует написанию слабо связанного программного кода. Слабосвязанный программный код – это код, в котором объекты довольно независимы друг от друга.

Полностью зависимость между объектами ликвидировать нельзя, поскольку ОО программа – это совокупность взаимодействующих объектов, но связанность не должна приводить к сращиванию объектов.

Если объект сильно связан с другими, то во первых, его приходится часто модифицировать, поскольку малейшее изменение окружения требует этого, во-вторых, затрудняется его повторное использование, так как при повторном использовании надо восстановить его связи.

В сильно связанном программном коде затруднено рассмотрение каждого объекта в отдельности.

Для создания слабо связанного программного кода нужно должным образом распределить ответственность, т.е. выяснить, что должен делать каждый из объектов. Каждый объект должен выполнять определенную функцию, все члены этого объекта должны на нее быть сориентированы, между ними должна быть логическая связь, как некоторые авторы называют высокое зацепление. Объект должен быть некоторым целым. Нет смысла в инкапсуляции случайного набора методов и данных.

Итак, для правильной инкапсуляции необходимо хорошо продумать структуру объекта, добиться того, чтобы все элементы объекта были логически связаны и нацелены на решение основной задачи.

Достоинства инкапсуляции

* Повторное использование кода.
* Возможность изменения реализации объекта. Изменения локализуются в объекте, а не превращаются в лавинный поток.
* Защита от деструктивного использования.

Недостаток инкапсуляции: Может быть скрыта важная для окружающего мира информация, например, при проведении приближенных расчетов важна точность вычисления и если пользователь, не только не в состоянии ее изменить, но даже узнать о ней, то он может отказаться от использования этого объекта. В этом случае, предлагают все особенности объекта описывать в сопровождающих текстах. Если изменение точности не предусмотрено, то надо хотя бы о ней сообщить. Возможно, ее значение окажется подходящим.

## 7.5 Агрегация

Наследование позволяет создавать новый класс на основе существующего. При наследовании в производный класс добавляются новые поля и новые методы, некоторые методы класса родителя изменяются. При этом не затрагивается код класса родителя, т.е. повторно используется старый код.

Агрегация:

public abstract class Engine

{ }

public class Car

{

Engine engine;

public Car(Engine eng)

{

engine = eng;

}

}

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

Старый код используется и при агрегации, включении в качестве полей объектов других классов.

От множественного наследования, как оказалось, можно отказаться без потери функциональности именно благодаря сочетанию наследования с агрегацией.

Встает вопрос, в каких случаях надо использовать наследование, а в каких случаях агрегацию?

Ответ может быть получен только с опорой на здравый смысл и опыт и частично на концептуальную модель.

Есть общее правило: при наследовании происходит расширение возможностей, конкретизируется поведение, но не изменяется типизация объекта, когда можно сказать, что об объекте дочернего класса, что он есть объект родительского класса. Например, у нас уже есть класс «Псовые». Нам надо создать класс «Волк». Ясно, что волк есть псовое. Следовательно, класс «Волк» должен быть наследником класса «Псовые».

Теперь пусть у нас есть класс «Хвост» и надо создать класс «Собака», в этом случае невозможно сказать собака есть хвост, но можно сказать, что собака имеет хвост, что хвост часть собаки, поэтому нужно использовать агрегацию.

Итак, глагол-связка «есть» определяет наследование. А глаголы: «имеет», «содержит», «включает» определяют агрегацию.

Замечание. Классы наследники часто называются подклассами, а родительские классы надклассами. Термин подклассы, кажется не естественным, если не знать ничего о моделировании, поскольку дочерний класс содержит все члены родительского, да еще некоторые свои члены. Но все встает на свои места, если понять, что дочерний класс конкретизирует родительский, что объекты дочернего класса можно рассматривать как объекты родительского класса.

Множество волков является подмножеством псовых.

## Задание по теме

*Общие принципы. Нужен класс, несколько полей, какой-то числовой метод. Чтобы от этого класса наследовалось еще два класса. Например, Фрукты - яблоки и груши. Метод можно переопределить. В классах-наследниках должны определяться дополнительные поля.*

*Например: есть класс треугольник. Нужно несколько полей, какой-то числовой метод. Чтобы от этого класса наследовалось еще два класса. Метод можно переопределить. В классах-наследниках должны определяться дополнительные поля.*

*Например: Геометрические фигуры. Создайте класс Круг с полем радиус. Реализуйте метод площадь и периметр (длина окружности). Затем создайте два класса, наследуемых от класса Круг: Конус с дополнительным методом объем и Цилиндр также с дополнительным методом объем. Переопределите метод площади (полной площади поверхности) в каждом из этих классов.*

*Например: Геометрические фигуры. Создайте класс Параллелограмм с полями сторона1, сторона 2, угол. Реализуйте методы площадь и периметр.*

*Затем создайте два класса, наследуемых от класса Параллелограмм: Квадрат с дополнительным полем диагональ и Ромб также дополнительным методом высота. Переопределите метод площади в каждом из этих классов.*

ЛР 4. Создать классы в соответствии с вариантами задания. Создать по 3 объекта классов-наследников. Вывести на форму поля и результаты вычисления метода для всех объектов классов в виде таблицы или ListBox.

### Варианты заданий

1. **Автомобили**. Создайте класс Автомобиль с полями марка, модель, год выпуска. Реализуйте метод расчета стоимости обслуживания, который вычисляет ежегодные расходы на обслуживание автомобиля. Затем создайте два класса, наследуемых от класса Автомобиль: Легковой автомобиль с дополнительным полем тип кузова и Грузовик с дополнительным полем грузоподъемность. Переопределите метод рассчитать стоимость обслуживания хотя бы в одном из этих классов.

2. **Электроприборы**. Создайте класс Электроприбор с полями название, мощность, цена. Реализуйте метод рассчитать стоимость эксплуатации, который вычисляет ежемесячные расходы на электроэнергию. Затем создайте два класса, наследуемых от Электроприбор: Холодильник с дополнительным полем объем и Телевизор с дополнительным полем диагональ экрана. Переопределите метод рассчитать стоимость эксплуатации хотя бы в одном из этих классов.

3. **Мебель**. Создайте класс Мебель с полями название, материал, цена. Реализуйте метод расчета стоимость доставки, который вычисляет стоимость доставки мебели в зависимости от ее веса. Затем создайте два класса, наследуемых от класса Мебель: Кресло с дополнительным полем высота и Стол с дополнительным полем размер. Переопределите метод рассчитать стоимость доставки хотя бы в одном из этих классов.

4. **Книги**. Создайте класс Книга с полями название, автор, цена. Реализуйте метод рассчитать скидку, который вычисляет размер скидки на книгу в зависимости от ее цены. Затем создайте два класса, наследуемых от класса Книга: Художественная литература с дополнительным полем жанр и Учебная литература с дополнительным полем уровень сложности. Переопределите метод рассчитать скидку хотя бы в одном из этих классов.

5. **Инструменты**. Создайте класс Инструмент с полями название, вес, цена. Реализуйте метод рассчитать стоимость аренды, который вычисляет стоимость аренды инструмента в зависимости от его веса. Затем создайте два класса, наследуемых от класса Инструмент: Ручной инструмент с дополнительным полем материал ручки и Электрический инструмент с дополнительным полем мощность. Переопределите метод рассчитать стоимость аренды хотя бы в одном из этих классов.

6. **Одежда**. Создайте класс Одежда с полями название, размер, цена. Реализуйте метод рассчитать скидку, который вычисляет размер скидки на одежду в зависимости от ее цены. Затем создайте два класса, наследуемых от Одежда: Верхняя одежда с дополнительным полем тип ткани и Нижнее белье с дополнительным полем материал. Переопределите метод расчёта скидки хотя бы в одном из этих классов.

7. **Мобильные устройства**. Создайте класс Мобильное устройство с полями название, модель, цена. Реализуйте метод расчета стоимости аксессуаров, который вычисляет стоимость дополнительных аксессуаров для устройства в зависимости от его цены. Затем создайте два класса, наследуемых от Мобильное устройство: Смартфон с дополнительным полем объем памяти и Планшет с дополнительным полем диагональ экрана. Переопределите метод рассчитать стоимость аксессуаров хотя бы в одном из этих классов.

8. **Спортивный инвентарь.** Создайте класс Спортивный инвентарь с полями название, вес, цена. Реализуйте метод расчёта стоимости доставки, который вычисляет стоимость доставки инвентаря в зависимости от его веса. Затем создайте два класса, наследуемых от Спортивный инвентарь: Мячи с дополнительным полем тип покрытия и Ракетки с дополнительным полем материал рукояти. Переопределите метод рассчитать стоимость доставки хотя бы в одном из этих классов.

9. **Электронные устройства**. Создайте класс Электронные устройства с полями название, модель, цена. Реализуйте метод расчёта стоимости гарантии, который вычисляет стоимость гарантийного обслуживания устройства в зависимости от его цены. Затем создайте два класса, наследуемых от Электронные устройства: Компьютер с дополнительным полем тип процессора и Телевизор с дополнительным полем диагональ экрана. Переопределите метод рассчитать стоимость гарантии хотя бы в одном из этих классов.

10. **Игрушки**. Создайте класс Игрушки с полями название, возрастное ограничение, цена. Реализуйте метод рассчитать стоимость доставки, который вычисляет стоимость доставки игрушки в зависимости от ее веса. Затем создайте два класса, наследуемых от Игрушки: Мягкие игрушки с дополнительным полем материал и Конструкторы с дополнительным полем количество деталей. Переопределите метод расчета стоимости доставки хотя бы в одном из этих классов.

11. **Канцелярские товары**. Создайте класс Канцелярские товары с полями название, тип, цена. Реализуйте метод рассчитать стоимость оптовой закупки, который вычисляет скидку на товары при оптовой закупке в зависимости от их цены. Затем создайте два класса, наследуемых от Канцелярские товары: Письменные принадлежности с дополнительным полем цвет и Бумага с дополнительным полем плотность. Переопределите метод расчета стоимости оптовой закупки хотя бы в одном из этих классов.

12. **Бытовая техника**. Создайте класс Бытовая техника с полями название, энергопотребление, цена. Реализуйте метод рассчитать стоимость страховки, который вычисляет стоимость страхового покрытия техники в зависимости от ее цены. Затем создайте два класса, наследуемых от Бытовая техника: Стиральные машины с дополнительным полем загрузка и Холодильники с дополнительным полем объем. Переопределите метод рассчитать стоимость страховки хотя бы в одном из этих классов.

13. **Музыкальные инструменты**. Создайте класс Музыкальные инструменты с полями название, материал, цена. Реализуйте метод рассчитать стоимость аренды, который вычисляет стоимость аренды инструмента в зависимости от его цены. Затем создайте два класса, наследуемых от Музыкальные инструменты: Струнные инструменты с дополнительным полем количество струн и Клавишные инструменты с дополнительным полем количество октав. Переопределите метод рассчитать стоимость аренды хотя бы в одном из этих классов.

14. **Электроника**. Создайте класс Электроника с полями: название, тип (например, телефон, ноутбук, телевизор), цена, производитель. Реализуйте метод расчёта гарантии, который определяет, сколько осталось времени гарантии в зависимости от даты покупки.

Затем создайте два класса, наследуемых от класса Электроника: Ноутбук с дополнительным полем объем оперативной памяти и Смартфон с дополнительным полем параметры камеры.

Переопределите метод расчёта гарантии хотя бы в одном из этих классов.

15. **Спортивная одежда**. Создайте класс Спортивная одежда с полями название, размер, цена. Реализуйте метод рассчитать скидку, который вычисляет размер скидки на одежду в зависимости от ее цены. Затем создайте два класса, наследуемых от Спортивная одежда: Футболки с дополнительным полем состав ткани и Шорты с дополнительным полем тип пояса. Переопределите метод расчёта скидку хотя бы в одном из этих классов.

16. **Чай**. Создайте класс Чай с полями название, сорт, цена. Реализуйте метод расчета скидки, который вычисляет размер скидки на чай в зависимости от его цены. Затем создайте два класса, наследуемых от класса Чай: Чай в пакетиках с дополнительным полем количество пакетиков и Чай в банке с дополнительным полем вес. Переопределите метод рассчитать скидку хотя бы в одном из этих классов.

17. **Товары**. Создайте класс Товар с полями название, производитель, цена, количество. Реализуйте метод стоимость, который вычисляет общую стоимость товара. Затем создайте два класса, наследуемых от класса Товар: Электронный товар с дополнительным полем Тип файла и Физический товар с дополнительным полем вес. Переопределите метод рассчитать стоимость хотя бы в одном из этих классов.

18. **Здания**. Создайте класс Здание с полями название, город, число этажей, площадь, стоимость эксплуатации здания на 1 квадратный метр. Реализуйте метод расчёта общих затрат на эксплуатацию здания.

Затем создайте два класса, наследуемых от класса Здание: Театр с дополнительным полем Художественный руководитель и Музей с дополнительным полем Стоимость экскурсии. Переопределите расчёта общих затрат на эксплуатацию здания хотя бы в одном из этих классов.

19. **Сотрудники**. Создайте класс Сотрудник с полями Ставка за час, отработанные часы, процент налога, премия, вычеты. Реализуйте метод расчёта зарплаты к получению.

Затем создайте два класса, наследуемых от класса Сотрудник: Менеджер с дополнительным полем Бонус за руководство и Стажер с дополнительным полем количество часов обучения, которое стажер проходит в неделю или месяц. Переопределите метод расчёта зарплаты хотя бы в одном из этих классов.

20. **Корабль**. Создайте класс Корабль с полями: Страна производитель, водоизмещение, дата ввода в эксплуатацию). Реализуйте метод износа исходя из срока службы 20 лет.

Затем создайте два класса, наследуемых от класса Грузовое судно: Ноутбук с дополнительным полем объем оперативной памяти и Пассажирское судно с дополнительным полем пассажировместимость. Переопределите метод расчёта износа хотя бы в одном из этих классов.

21. **Автомобили**. Создайте класс Автомобиль с полями марка, модель, год выпуска, пробег, цена. Реализуйте метод расхода топлива на км.

Затем создайте два класса, наследуемых от класса Автомобиль: Электроавтомобиль с дополнительным полем Емкость батареи и Спорткар с дополнительным полем Тип двигателя. Переопределите метод расхода топлива хотя бы в одном из этих классов.

22. **Банковские счета**. Создайте класс Банковский счет с полями баланс, процентная ставка, номер счета, имя счета, валюта. Реализуйте метод расчёта суммы начисленных процентов за год.

Затем создайте два класса, наследуемых от класса Банковский счет: Текущий счёт с дополнительным полем Лимит овердрафта и Сберегательный счёт с дополнительным полем Минимальный баланс для сбережений. Переопределите метод расчёта процентов хотя бы в одном из этих классов.

23. **Студенты платного отделения**. Создайте класс Студент платного отделения с полями ФИО, Номер студенческого билета, стоимость обучения за год, оплачено. Реализуйте метод, который возвращает количество денег, необходимых для завершения обучения.

Затем создайте два класса, наследуемых от класса Студент платного отделения: Студент очной формы с дополнительным полем количество практик и Студент заочной формы с дополнительным полем Продолжительность установочной недели.

24. **Поезд**. Создайте класс Поезд с полями: название, количество вагонов, производитель, дата ввода в эксплуатацию). Реализуйте метод износа исходя из срока службы 20 лет.

Затем создайте два класса, наследуемых от класса Поезд: Электропоезд с дополнительным полем Напряжение в сети и Дизельпоезд с дополнительным полем мощность двигателя.

Переопределите метод расчёта износа хотя бы в одном из этих классов.

25. **Страны**. Создайте класс Страна с полями название, столица, население, площадь, валюта. Реализуйте метод расчёта плотности населения.

Затем создайте два класса, наследуемых от класса Страна: Республика с дополнительным полем Число ветвей власти и Монархия с дополнительным полем Королевский титул.

36. **Мебель**. Создайте класс Мебельные изделия с полями название, материал, цена. Реализуйте метод рассчитать стоимость доставки, который вычисляет стоимость доставки изделия в зависимости от его веса. Затем создайте два класса, наследуемых от Мебельные изделия: Шкафы с дополнительным полем количество секций и Столы с дополнительным полем размер. Переопределите метод расчета стоимости доставки хотя бы в одном из этих классов.

37. **Сотрудники**. Создайте класс Сотрудник с полями Ставка за час, отработанные часы, процент налога, премия, вычеты. Реализуйте метод расчёта зарплаты к получению.

Затем создайте два класса, наследуемых от класса Сотрудник: Фрилансер с дополнительным полем Число завершенных проектов и Удаленный работник с дополнительным полем Доплата за удалённую работу. Переопределите метод расчёта зарплаты хотя бы в одном из этих классов.

38. **Автомобили**. Создайте класс Автомобиль с полями марка, модель, расход топлива (л/100 км), пробег, объем топливного бака. Реализуйте метод расчёта, который возвращает запаса хода автомобиля на полном баке.

Затем создайте два класса, наследуемых от класса Автомобиль: Электромобиль с дополнительным полем Потребление энергии (в кВтч/100 км) и Гибридный автомобиль с дополнительным полем Запас хода на электротяге

### Примеры расчета методов

1. Класс Автомобиль

Метод рассчитать стоимость обслуживания:

если год\_выпуска < 5: return 500

elif год\_выпуска < 10: return 800

else: return 1200

Класс Легковой автомобиль

Метод рассчитать стоимость обслуживания:

if тип кузова == "седан":

return super().рассчитать стоимость обслуживания \* 0.9

elif тип кузова == "хэтчбек":

return super().рассчитать стоимость обслуживания \* 0.8

else: return super().рассчитать стоимость обслуживания \* 1.1

Класс Грузовик

Метод рассчитать стоимость обслуживания:

if грузоподъемность <= 5000:

return super().рассчитать стоимость обслуживания \* 1.2

elif грузоподъемность <= 10000:

return super().рассчитать стоимость обслуживания \* 1.5

else: return super().рассчитать стоимость обслуживания \* 2

2. Класс Электроприбор

Метод рассчитать стоимость эксплуатации:

return мощность \* 0.5 \* 30 # 0.5 - средний тариф на электроэнергию, 30 - дни в месяце

Класс Телевизор

Метод рассчитать стоимость эксплуатации:

if диагональ\_экрана <= 32:

return super().рассчитать стоимость эксплуатации \* 0.6

elif диагональ\_экрана <= 50:

return super().рассчитать стоимость эксплуатации \* 0.8

else: return super().рассчитать стоимость эксплуатации \* 1

3. Класс Мебель

Метод рассчитать стоимость доставки:

return вес \* 50

5. Класс Инструмент

Метод рассчитать стоимость аренды:

return вес \* 10

6. Класс Одежда

Метод рассчитать скидку():

if цена <= 1000:

return 0.1

else: return 0.2

9. Класс Электронные устройства

Метод рассчитать стоимость гарантии:

if self.цена <= 20000:

return 500

elif self.цена <= 50000:

return 1000

else: return 2000

15. Класс Спортивная одежда

Метод рассчитать скидку()

if цена <= 1000:

return 0.1

elif sцена <= 3000:

return 0.15

else:

return 0.2