

**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ
КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Поволжский государственный университет телекоммуникаций и информатики»
КОЛЛЕДЖ СВЯЗИ

УТВЕРЖДАЮ:
Директор КС ПГУТИ

Андреев Р.В.
« 11 » 11 2021г.

СБОРНИК
практических работ
дисциплине
Основы алгоритмизации и программирования
(часть 3)

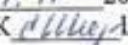
для специальности:

10.02.05 - «Обеспечение информационной безопасности автоматизированных систем»

Номера работ: № 28 – 42

Сборник рассчитан на 40 часов

Составлен преподавателем Сергеев Р.А.

Рассмотрен на заседании П(Ц)К
«Информационные системы и технологии»
Протокол № 5 от 11.11 2021.
Председатель П(Ц)К  Шомас Е.А.

Самара
2021

Перечень компетенций

| Общие компетенции | |
|--|---|
| ОК 1 | Выбирать способы решения задач профессиональной деятельности, применительно к различным контекстам |
| ОК 2 | Осуществлять поиск, анализ и интерпретацию информации, необходимой для выполнения задач профессиональной деятельности |
| ОК 3 | Планировать и реализовывать собственное профессиональное и личностное развитие |
| ОК 4 | Работать в коллективе и команде, эффективно взаимодействовать с коллегами, руководством, клиентами |
| ОК 5 | Осуществлять устную и письменную коммуникацию на государственном языке с учётом особенностей социального и культурного контекста |
| ОК 6 | Проявлять гражданско-патриотическую позицию, демонстрировать осознанное поведение на основе традиционных общечеловеческих ценностей |
| ОК 7 | Содействовать сохранению окружающей среды, ресурсосбережению, эффективно действовать в чрезвычайных ситуациях |
| ОК 8 | Использовать средства физической культуры для сохранения и укрепления здоровья в процессе профессиональной деятельности и поддержания необходимого уровня физической подготовленности |
| ОК 9 | Использовать информационные технологии в профессиональной деятельности |
| ОК 10 | Пользоваться профессиональной документацией на государственном и иностранном языке |
| ОК 11 | Планировать предпринимательскую деятельность в профессиональной сфере |
| Профессиональные компетенции для специальности 10.02.05 | |
| ПК 2.1 | Осуществлять установку и настройку отдельных программных, программно-аппаратных средств защиты информации; |
| ПК 2.2 | Обеспечивать защиту информации в автоматизированных системах отдельными программными, программно-аппаратными средствами; |
| ПК 2.3 | Осуществлять тестирование функций отдельных программных и программно-аппаратных средств защиты информации; |
| ПК 2.4 | Осуществлять обработку, хранение и передачу информации ограниченного доступа; |
| ПК 2.6 | Осуществлять регистрацию основных событий в автоматизированных (информационных) системах, в том числе с использованием программных и программно-аппаратных средств обнаружения, предупреждения и ликвидации последствий компьютерных атак |

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 28

НАИМЕНОВАНИЕ: Создание простого приложения на C#

1.ЦЕЛЬ РАБОТЫ: Научиться составлять линейные программы в среде C#

2.ПОДГОТОВКА К ЗАНЯТИЮ:

2.1. Изучить предложенную литературу.

2.2. Подготовить бланк отчёта.

3.ЛИТЕРАТУРА:

3.1. Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЕ:

5.1 Вычислить значения. Вывести значения вводимых исходных данных и результаты, сопровождая их вывод именами выводимых переменных. Задание выполнить в виде консольного приложения.

А)

$$s = \left| x^{y/x} - \sqrt{\frac{y}{x}} \right|;$$

$$w = (y - x) \frac{y - \frac{z}{y - x}}{1 + (y - x)^2}$$

$$\begin{array}{l} x = 1.82 \\ y = 18 \\ z = -3.29 \end{array}$$

Б)

$$s = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!};$$

$$w = x (\sin(x) + \cos(y))$$

$$\begin{array}{l} x = 0.33 \\ y = 0.02 \end{array}$$

В)

$$y = e^{-bt} \sin(at + b) - \sqrt{|bt + a|};$$

$$s = b \sin(at^2 \cos(at)) - 1$$

$$\begin{array}{l} a = -0.5 \\ b = 1.7 \\ t = 0.44 \end{array}$$

Г)

$$w = \sqrt{x^2 + b} - \frac{b^2 \sin^3(x + a)}{x};$$

$$y = \cos^2(x^3) - x / \sqrt{a^2 + b^2}$$

$$\begin{array}{l} a = -0.5 \\ b = 15.5 \\ x = -2.9 \end{array}$$

Д)

$$s = x^3 \operatorname{tg}^2((x + b)^2) + \frac{a}{\sqrt{x + b}};$$

$$g = \frac{bx^2 - a}{e^{ax} - 1}$$

$$\begin{array}{l} a = 16.5 \\ b = 3.4 \\ x = 0.61 \end{array}$$

Е)

$$r = \frac{x^2(x + 1)}{b} - \sin^2(x + a);$$

$$s = \sqrt{\frac{xb}{a}} + \cos((x + b)^3)$$

$$\begin{array}{l} a = 0.7 \\ b = 0.05 \\ x = 0.5 \end{array}$$

6. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ:

- 6.1 По предложенной литературе изучить необходимый материал;
- 6.2 Выполнить задания практического занятия и записать листинг программы;
- 6.3 Дать ответы на контрольные вопросы;

7. СОДЕРЖАНИЕ ОТЧЕТА:

- 7.1 Наименование и цель работы
- 7.2 Код программы. Записать результат (листинг) выполненных заданий.
- 7.3 Ответы на контрольные вопросы
- 7.4 Вывод о проделанной работе

8. КОНТРОЛЬНЫЕ ВОПРОСЫ:

- 8.1 Где описываются константы, переменные и типы данных?
- 8.2 Стандартные функции.
- 8.3 Операторы присваивания.
- 8.4 Пустая и составная инструкция.
- 8.5 Процедуры ввода Read и ReadLine.
- 8.6 Процедуры вывода Write и WriteLine.
- 8.7 Последовательность действий при выполнении оператора присваивания.
- 8.8 Приоритетность выполнения операций в выражениях.
- 8.9 Как организовать пропуск одной, двух строк при выводе?

ПРИЛОЖЕНИЕ:

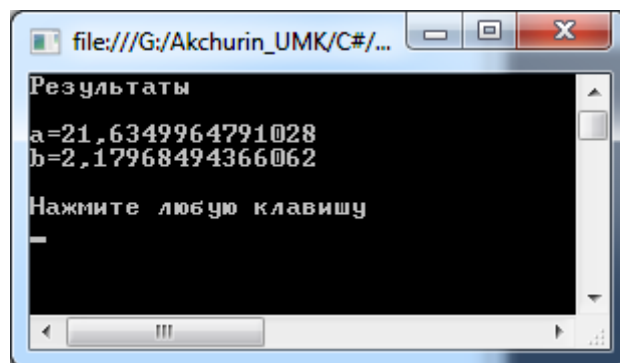
Пример. Вычислить при $x = 2.1$, $y = 0.59$, $z = -4.8$ значения a и b , используя формулы:

$$a = y \cdot \operatorname{tg}^3(x^2) + \sqrt{\frac{z^2}{y^2 + x^2}};$$
$$b = \ln(y + x^2) + \sin^2(z/x)$$

Листинг программы

```
using System;
namespace ConsoleLineStr
{
    class Program
    {
        static void Main()
        {
            double x = 2.1, y = 0.59, z = -4.8, a = 0, b = 0; // Переменные типа double
            a = y * Math.Pow(Math.Tan(x * x), 3); // Вычисляем a
            a += Math.Sqrt(z * z / (y * y + x * x));
            Console.WriteLine("Результаты");
            Console.WriteLine();
            Console.Write("a="); // Вывод a
            Console.WriteLine(a.ToString());
            b = Math.Log(y + x * x); // Вычисляем b
            b += Math.Pow(Math.Sin(z / x), 2);
            Console.Write("b="); // Вывод b
            Console.WriteLine(b.ToString());
            Console.WriteLine();
            Console.WriteLine("Нажмите любую клавишу");
            Console.ReadKey(); // Пауза
        }
    }
}
```

Внимание. При вводе данных в консоли разделитель целой и дробной части вещественного числа – запятая.



ПРАКТИЧЕСКАЯ РАБОТА №29

НАИМЕНОВАНИЕ: Управляющие конструкции языка C#: разветвляющиеся программы

1. ЦЕЛЬ: Закрепление понятий о ветвлении в программах на C#, развитие навыков разработки алгоритмов и программ с условным оператором и оператором выбора. //Сформировать компетенции ОК 1 - ОК 11; овладеть знаниями и умениями для освоения ПК 4.2

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Программирование на языке C# в среде Visual Studio».

3. ЛИТЕРАТУРА:

3.1. Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КСПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

5.1 Разработать алгоритм в словесной форме либо на псевдоязыке для выполнения задания по варианту.

5.2 Написать программу на языке C# по полученному алгоритму.

5.3 Построить блок-схему программы.

Вариант 1.

Задание 1. Пользователь вводит коэффициенты квадратного уравнения. Написать программу, вычисляющие корни этого квадратного уравнения.

Задание 2. Написать программу, которая по введенному номеру дня недели сообщает, какой это день (понедельник, вторник, и т.д.), а также выводит, является ли данный день выходным или рабочим.

Вариант 2.

Задание 1. Составить программу, вычисляющую значение кусочной функции

$$y = 15 \text{ при } x > 5; y = x^2 \text{ при } x \leq 5.$$

Задание 2. С клавиатуры вводятся два числа и символ соответствующей операции: сложение, умножение, вычитание, деление. Составить программу, выполняющую ту операцию над двумя числами, которую ввел пользователь.

Вариант 3.

Задание 1. Составить программу, вычисляющую значение кусочной функции

$$y = \sqrt{x+1} \text{ при } x \geq 3; y = \lg x \text{ при } 1 < x < 3; y = 0 \text{ при } x \leq 1.$$

Задание 2. Написать программу, которая по введенному номеру месяца выводит строку с названием данного месяца.

Вариант 4.

Задание 1. С клавиатуры вводятся значения двух углов треугольника в градусах. Определить тип треугольника (остроугольный, прямоугольный, тупоугольный) по введенным значениям углов.

Задание 2. Написать программу, которая по введенному количеству баллов (из 100 баллов) шкале определяет оценку: меньше 50 — неудовлетворительно, от 50 до 75 — удовлетворительно, от 76 до 85 — хорошо, больше 86 — отлично.

Вариант 5.

Задание 1. Написать программу, которая определяет максимальное из 3 введенных чисел.

Задание 2. Составить программу, которая по введенному дню недели выводит время начало пар в этот день.

Вариант 6.

Задание 1. Если сумма вклада в банка меньше 1000 денежных единиц, то банк начисляет 5 процентов. Если сумма от 1000 до 3000 — 7 процентов. Если сумма больше 3000 — 9 процентов. Составить программу, в которой после ввода суммы вклада происходит вывод суммы с начисленными процентами по вкладу.

Задание 2. Составить программу, которая при вводе числа от 1 до 10 выводит их двоичное представление.

Вариант 7.

Задание 1. Вводятся два числа. Если оба числа четные — найти их произведение, иначе — сумму.

Задание 2. Написать программу, которая по введенному номеру месяца выводит число дней в этом месяце.

Вариант 8.

Задание 1. Составить программу вычисления сопротивления двух параллельно подключенных резисторов. Условие — значения сопротивлений должны быть положительными.

Задание 2. Составить программу, в которой при вводе символа Q происходит выход из программы, при вводе символа H — выдается справка по использованию программы (выводится описание, что означают символы Q, H, A), при вводе символа A — информация об авторах программы.

Вариант 9.

Задание 1. Составить программу, определяющую значение кусочной функции
 $y = 1/x \text{ при } x > 0; y = x \text{ при } x \leq 0$.

Задание 2. С клавиатуры вводится значение радиуса. Написать программу, в которой при вводе числа 1 вычисляется площадь круга заданного радиуса, при вводе числа 2 вычисляется площадь сферы заданного радиуса, при вводе числа 3 — объем шара заданного радиуса.

Вариант 10.

Задание 1. Составить программу, вычисляющую среднее арифметическое и среднее гармоническое четырех введенных с клавиатуры чисел. Определить в программе, какое из полученных значений больше.

Задание 2. С клавиатуры вводятся цифры от 0 до 9. Вывести на экран название введенной цифры (0 — ноль, 1 — один, 2 — два, и т.д.).

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

6.1 Получить допуск к работе;

6.2 Выполнить задания.

6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

7.1 Наименование, цель занятия;

7.2 Выполненные задания, блок-схемы программ;

7.3 Ответы на контрольные вопросы;

7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

8.1 Что такое оператор ветвления? Какие формы имеет оператор ветвления в языке C#?

8.2 Что такое оператор выбора? Какова его структура в языке C#?

8.3 Раскройте назначение default в операторе switch. Будет ли работать программа, если не указывать в операторе switch части default.

Краткие теоретические сведения:

От точки входа в программу инструкции выполняются одна за другой, пока не встретятся команды, нарушающие такой линейный порядок выполнения программы. Важным примером оператора, который нарушает последовательное выполнение программы, является оператор ветвления или условный оператор `if`. Этот типовой оператор, присутствующий практически во всех языках программирования, передает управление на ту или иную команду в зависимости от выполнения условия, которое у этого оператора находится в его заголовке. Условие представляет собой некоторое выражение, имеющее булево значение

Оператор `if` в языке `C#` может иметь несколько форм.

Первая форма состоит из оператора `if` с частью `else`, таким образом, на основе значения некоторого булева выражения происходит выбор выполнения одной из двух частей:

```
if (условие-выражение)
{
    Операторы тела в случае, если условие равно True
}

else
{
    Операторы тела в случае, если условие равно False
}
```

Другая форма оператора `if` записывается без части `else`. Поэтому, если выражение, стоящее в условии этого оператора, имеет значение `False`, то происходит просто переход на инструкцию, следующую за инструкцией `if`:

```
if (условие-выражение)
{
    Операторы тела в случае, если условие равно True
}
```

прочие инструкции программы, которые получают управление сразу, если условие равно `False`.

Ещё одной формой оператора ветвления может быть случай, когда в части `else` присутствует другой оператор `if`:

```
if (условие-выражение)
{
    Операторы тела в случае, если условие равно True
}

else if(другое условие-выражение)
{
    Операторы тела в случае, если другое условие равно True и не выполнилось первое условие-выражение
}
```

Данным способом можно проверять содержимое некоторой переменной, которая может принимать несколько значений.

Однако, при большом многовариантном выборе использование комбинации `if...else` может стать довольно неудобным. В таких случаях используется специальный оператор `switch` — оператор выбора одного из многих вариантов. Его также называют переключателем.

Данный оператор напоминает подобный оператор из других языков программирования. Оператор switch выбирает инструкции для выполнения в зависимости от совпадения с шаблонами некоторого выражения, записанного в заголовке оператора:

switch (переменная, которая сравнивается с шаблонами)

```
{  
    case (шаблон — может быть как конкретным значением, так и сравнение с некоторой  
константой):  
        операторы, выполняющиеся в случае совпадения переменной из заголовка с шабло-  
ном данного case;  
        break; - данный оператор требуется для выхода из оператора switch, иначе, продол-  
жится сравнения с другими case шаблонами после выполнения инструкций данного;  
    case  
    case ... //подобных case может быть множество  
    default: выполняется, если не произошло ни одного совпадения с шаблонами case;  
}
```

Итак, данный оператор первоначально берет значение, помещенное в заголовок switch, и начинает последовательное сравнение с шаблонами, указанными в частях case. Если произошло совпадение, происходит выполнение инструкций, указанных в соответствующем case. Если не произошло совпадения ни с одним из case, выполняются инструкции, указанные в необязательной части default данного оператора.

На рисунке приведен пример программы, использующей оператор if в языке C#.

```
using System;  
class HelloWorld {  
    static void Main() {  
        int a, b;  
        a=Convert.ToInt32(Console.ReadLine());  
        b=Convert.ToInt32(Console.ReadLine());  
        if (a>b){  
            Console.WriteLine("Первое число {0} больше второго числа {1}", a,b);  
        }  
        else  
        {  
            Console.WriteLine("Второе число {0} больше или равно первому числу {1}", b,a);  
        }  
    }  
}
```

Рисунок Пример использования оператора if

В данной программе в начале происходит считывание значений чисел a и b с клавиатуры. Далее эти значения сравниваются в условии оператора if. В зависимости от логического значения, возвращаемого данным оператором, происходит вывод либо строки, сообщающей, что первое введенное значение больше второго, либо строки, сообщающей, что второе число больше или равно первому.

На рисунке изображена программа, использующая оператор выбора switch

```

using System;
class HelloWorld {
    static void Main() {

        int a;
        a=Convert.ToInt32(Console.ReadLine());
        switch (a)
        {
            case 1:
                Console.WriteLine("Winter");
                break;

            case 2:
                Console.WriteLine("Spring");
                break;

            case 3:
                Console.WriteLine("Summer");
                break;

            case 4:
                Console.WriteLine("Autumn");
                break;

            default:
                Console.WriteLine("Incorrect value");
                break;
        }
        Console.WriteLine("HelloWorld");
    }
}

```

Рисунок Пример использования оператора switch

В данной программе пользователь вводит число, соответствующее номеру времени года. Программа считывает данное число, а далее происходит сравнение числа с шаблонами в частях case оператора switch. В случае совпадения с шаблоном происходит вывод строки с указанием времени года. Обратите внимание на оператор break, идущий после каждой части case. Он необходим для выхода из оператора switch при совпадении введенного значения с шаблоном. Также, в программе присутствует часть default, срабатывающая, если ни одного совпадения не произошло. Данная часть выполняется по умолчанию и в этом случае выводится сообщение о том, что пользователь ввел некорректное значение (не принадлежащее числам 1, 2, 3, 4).

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 30

НАИМЕНОВАНИЕ: Управляющие конструкции языка С#: программы с циклами

1.ЦЕЛЬ РАБОТЫ: Научиться работать с циклами при программировании в среде С#

2.ПОДГОТОВКА К ЗАНЯТИЮ:

2.1. Изучить предложенную литературу.

2.2. Подготовить бланк отчёта.

3.ЛИТЕРАТУРА:

3.1. Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5.ЗАДАНИЕ:

5.1 Вычислить значение суммы членов бесконечного ряда с заданной точностью Е с использованием инструкции цикла while. На печать вывести значение суммы и число членов ряда, вошедших в сумму. Проект – консольное приложение.

Подзадания

| № | Сумма членов ряда | x | Точность |
|-----|---|------|----------|
| 1. | $s = -\frac{(2x)^2}{2!} + \frac{(2x)^4}{4!} + (-1)^n \frac{(2x)^{2n}}{(2n)!} + \dots$ | 0.1 | 0.00001 |
| 2. | $s = x - \frac{x^3}{3} + \dots + (-1)^{n-1} \frac{x^{2n-1}}{2n-1} + \dots$ | 0.1 | 0.00005 |
| 3. | $s = \frac{x^3}{5} - \frac{x^5}{17} + (-1)^{n+1} \frac{x^{2n+1}}{4n^2+1} + \dots$ | 0.15 | 0.001 |
| 4. | $s = 1 + \frac{x}{1!} \cos \frac{\pi}{4} + \dots + \frac{x^n}{n!} \cos(\frac{\pi}{4} n) + \dots$ | 0.12 | 0.0001 |
| 5. | $ch(x) = s = 1 + \frac{x^2}{2!} + \dots + \frac{x^{2n}}{(2n)!} + \dots$ | 0.7 | 0.0001 |
| 6. | $sh(x) = s = x + \frac{x^3}{3!} + \dots + \frac{x^{2n+1}}{(2n+1)!} + \dots$ | 1.7 | 0.001 |
| 7. | $arctg(x) = s = \frac{1}{x} + \dots + (-1)^n \frac{1}{(2n+1)x^{2n+1}} + \dots$ | 1.5 | 0.0005 |
| 8. | $\pi = s = 4 \left(1 - \frac{1}{3} + \dots + (-1)^n \frac{1}{2n+1} + \dots \right)$ | | 0.0001 |
| 9. | $\cos \frac{\pi}{6} = s = 1 - \frac{(\pi/6)^2}{2!} + \dots + (-1)^n \frac{(\pi/6)^{2n}}{(2n)!} + \dots$ | | 0.00005 |
| 10. | $\sin \frac{\pi}{3} = s = \frac{\pi}{3} - \frac{(\pi/3)^3}{3!} + \dots + (-1)^n \frac{(\pi/3)^{2n+1}}{(2n+1)!} + \dots$ | | 0.00005 |

5.2 Выполнить ту же задачу с применением инструкции цикла do...while. Проект – консольное приложение.

5.3 Подсчитать факториал числа введенного ранее с клавиатуры.

6. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ:

6.1 По предложенной литературе изучить необходимый материал;

6.2 Выполнить задания практического занятия и записать листинг программы;

6.3 Дать ответы на контрольные вопросы;

7. СОДЕРЖАНИЕ ОТЧЕТА:

- 7.1 Наименование и цель работы.
- 7.2 Код программы (записать результат (листинг) выполненных заданий).
- 7.3 Ответы на контрольные вопросы.
- 7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ:

- 8.1 Циклический процесс с неизвестным числом повторений.
- 8.2 Его отличия от цикла с заданным числом повторений.
- 8.3 Инструкции языка C# для организации таких циклов. Их сравнение.
- 8.4 Синтаксис инструкции while.
- 8.5 Как выполнить группу операторов в цикле while?
- 8.6 Синтаксис инструкции do...while.
- 8.7 Синтаксис инструкции foreach.
- 8.8 Прямое вычисление суммы членов бесконечного ряда.
- 8.9 Вычисление суммы членов бесконечного ряда по рекуррентной формуле.
- 8.10 Условие выхода из цикла при вычислении суммы членов бесконечного ряда.

ПРИЛОЖЕНИЕ:

Цикл while

Пример. Вычислить значение суммы членов бесконечного ряда

$$s = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \dots$$

при $x = 0.1$ с точностью до члена ряда с модулем, меньшим $E=0.00001$.

Для вычисления очередного члена ряда будем использовать рекуррентное соотношение, связывающее его с предыдущим членом $a(n+1) = q \cdot a(n)$. Применение рекуррентных формул позволяет избежать вычисления факториала и возведения в произвольную степень. Рекуррентный коэффициент q найдем из выражений для текущего и следующего членов ряда

$$a(n) = (-1)^n \frac{x^{2n+1}}{(2n+1)!} \quad \text{и} \quad a(n+1) = (-1)^{n+1} \frac{x^{2(n+1)+1}}{(2(n+1)+1)!}$$

Деля второе выражение на первое, получим

$$q = \frac{a(n+1)}{a(n)} = \frac{-x^2}{(2n+2)(2n+3)}$$

Значение начального члена ряда задаем до цикла путем прямого присваивания (номер начального члена n в разных вариантах равен 0 или 1, правильное значение определяется по формуле текущего члена). В нашем задании $n=0$, $a=x$.

Листинг программы

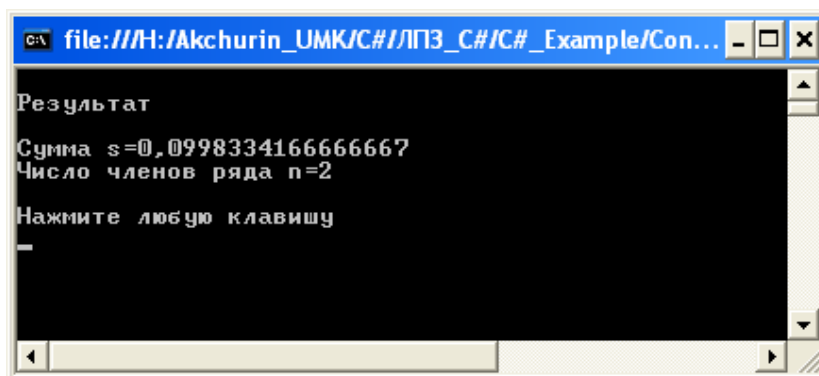
```
using System;
namespace ConsoleWhile
{
    class Program
    {
```

```

static void Main()
{
    double a=0, e=0.00001, q=0, s=0, x=0.1;
    int n = 0;
    a = x;           // Инициализация цикла
    s = a;
    while (Math.Abs(a) > e)    // Цикл
    {
        q = -x * x / (2 * n + 2) / (2 * n + 3);
        a *= q;
        s += a;
        n++;
    }
    Console.WriteLine();
    Console.WriteLine("Результат");
    Console.WriteLine();
    Console.WriteLine("Сумма s=" + Convert.ToString(s));
    Console.WriteLine("Число членов ряда n=" + Convert.ToString(n));
    Console.WriteLine();
    Console.WriteLine("Нажмите любую клавишу");
    Console.ReadKey();    // Пауза
}
}
}

```

Консоль перед закрытием программы:



Цикл do...while

Пример.

Листинг программы

```

using System;

namespace DoWhile
{
    class Program
    {
        static void Main()
        {
            double a = 0, e = 0.00001, q = 0, s = 0, x = 0.1;
            int n = 0;

```

```

a = x;           // Инициализация цикла
s = a;
do               // Тело цикла
{
    q = -x * x / (2 * n + 2) / (2 * n + 3);
    a *= q;
    s += a;
    n++;
}
while (Math.Abs(a) > e); // Цикл повторять
Console.WriteLine();
Console.WriteLine("Результат");
Console.WriteLine();
Console.WriteLine("Сумма s = {0}",s);
Console.WriteLine("Число членов ряда n = {0}",n);
Console.WriteLine();
Console.WriteLine("Нажмите любую клавишу");
Console.ReadKey(); // Пауза
}
}
}

```

ПРАКТИЧЕСКАЯ РАБОТА №31

НАИМЕНОВАНИЕ: Реализация консольного ввода-вывода в C#

1. ЦЕЛЬ: Углубление понятий о вводе-выводе в программах на C#, развитие навыков работы с консольным вводом-выводом на языке C#. //Сформировать компетенции ОК 1 - ОК 11; овладеть знаниями и умениями для освоения ПК 4.2

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Программирование на языке C# в среде Visual Studio».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПУТИ, 2021 г.

3. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

5.1 Разработать алгоритм в словесной форме либо на псевдоязыке для выполнения задания по варианту.

5.2 Написать программу на языке C# по полученному алгоритму.

5.3 Построить блок-схему программы.

Вариант 1.

Задание 1. Составить программу, вычисляющую значение выражения

$$y_1 = |x \cdot t| - 18 + 19.2 - x^2 + t/2.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 24 строки, 42 столбца, цвет фона сделать белым, а цвет текста — черным.

Вариант 2.

Задание 1. Составить программу, вычисляющую значение выражения

$$x = \sqrt{|15+Y| + Y1} - 19/t.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 33 строки, 54 столбца, цвет фона сделать зеленым, а цвет текста — белым.

Вариант 3.

Задание 1. Составить программу, вычисляющую значение выражения

$$t = b \cdot x^2 + x_1 - 18.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 51 строки, 66 столбца, цвет фона сделать синим, а цвет текста — белым.

Вариант 4.

Задание 1. Составить программу, вычисляющую значение выражения

$$x = |x_1^2| + y - y_1.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 50 строки, 46 столбца, цвет фона сделать желтым, а цвет текста — черным.

Вариант 5.

Задание 1. Составить программу, вычисляющую значение выражения

$$y = \sqrt{y_1 + y_2 + y_3} — 256.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 30 строки, 25 столбца, цвет фона сделать желтым белым, а цвет текста — красным.

Вариант 6.

Задание 1. Составить программу, вычисляющую значение выражения

$$g = g_1 * t^2 / 2 + a_1 * 3.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 28 строки, 28 столбца, цвет фона сделать желтым серым, а цвет текста — черным.

Вариант 7.

Задание 1. Составить программу, вычисляющую значение выражения

$$y = \sqrt{s_1 + s_3} — s_1 * s_3.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 74 строки, 66 столбца, цвет фона сделать желтым черным, а цвет текста — синим.

Вариант 8.

Задание 1. Составить программу, вычисляющую значение выражения

$$s_2 = t * R * 8 * S — 19.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 60 строки, 66 столбца, цвет фона сделать белым, а цвет текста — зеленым.

Вариант 9.

Задание 1. Составить программу, вычисляющую значение выражения

$$y = 19.3 — t^3 / \sqrt{15 * x}.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 25 строки, 35 столбца, цвет фона сделать черным, а цвет текста — белым.

Вариант 10.

Задание 1. Составить программу, вычисляющую значение выражения

$$P = |N * 15 + 286 / N|.$$

Все переменные в правой части выражения вводятся с клавиатуры пользователем. Вывод значения выражения предусмотреть с использованием не менее 3 различных форматов, варьируя точность/ширину вывода.

Задание 2. Изменить для программы из задания 1 размер окна консоли на 36 строки, 77 столбца, цвет фона сделать зеленым, а цвет текста — черным.

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

6.1 Получить допуск к работе.

6.2 Выполнить задания.

6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

7.1 Наименование, цель занятия;

7.2 Выполненные задания, блок-схемы программ;

7.3 Ответы на контрольные вопросы;

7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

8.1 Как в C# осуществляется вывод информации в консоль?

8.2 Как в C# осуществляется ввод информации с клавиатуры? Как вывести на экран число?

8.3 Чем отличается Write от WriteLine?

8.4 Как можно задать цвет фона в консоли?

8.5 Для чего используется класс Convert?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

В языке C# класс Console обеспечивает приложению доступ к стандартным потокам ввода, вывода и ошибок. Стандартный поток ввода — клавиатура. Стандартный поток вывода и ошибок — экран. Эти потоки могут быть перенаправлены из или в файлы.

Вывод информации на консоль.

Для обеспечения вывода информации на консоль используются методы Write и WriteLine. Данные методы могут принимать в качестве аргументов как строки, так и числа.

Например,

```
Console.WriteLine(9);  
Console.WriteLine("Hello").
```

Write отличается от WriteLine тем, что не переводит после вывода курсор на следующую строку. Рассмотренные методы позволяют осуществлять форматный вывод, аналогичный функции printf языка C:

```
int a = 100, c = 11, d = 0;  
Console.WriteLine("Переменная a = {0}, переменная c = {1}, переменная d = {2}", a, c, d);
```

В примере в качестве первого параметра передается строка, содержащая маркеры в фигурных скобках. При выводе на места маркеров будут подставлены параметры, которые следуют за строкой.

При форматном выводе можно задавать ширину поля вывода, для этого в фигурных скобках после номера параметра необходимо указать число, которое будет иметь смысл ширины области вывода, причем выравнивание можно осуществлять по левому или правому краю.

Например,

```
int a = 100;  
Console.WriteLine("Переменная a = {0, -5}", a);
```

выровняет значение 100 по левому краю, записав его в поле вывода шириной 5 ячеек.

При выводе также можно добавлять строку формата вместе с необязательным значением точности.

Используемые форматы:

| | |
|--------------|---|
| C / c | Задаёт формат денежной единицы, указывает количество десятичных разрядов после запятой |
| D / d | Целочисленный формат, указывает минимальное количество цифр |
| E / e | Экспоненциальное представление числа, указывает количество десятичных разрядов после запятой |
| F / f | Формат дробных чисел с фиксированной точкой, указывает количество десятичных разрядов после запятой |
| G / g | Задаёт более короткий из двух форматов: F или E |
| N / n | Также задаёт формат дробных чисел с фиксированной точкой, определяет количество разрядов после запятой |
| P / p | Задаёт отображения знака процентов рядом с числом, указывает количество десятичных разрядов после запятой |
| X / x | Шестнадцатеричный формат числа |

Например,

```
double d = 15.3333;  
Console.WriteLine("{0:f2}", d);
```

данный код выведет на экран 15.33, поскольку в качестве формата указан формат дробных чисел (можно использовать и прописную F), а число 2 показывает, сколько десятичных чисел после запятой выведется на экран.

```
double d = 15.3333;  
Console.WriteLine("{0:E2}", d);
```

Данный же код выведет 1.53E+001, что соответствует экспоненциальному представлению числа с двумя десятичными разрядами после запятой.

Ввод информации

В классе Console существует специальный метод ReadLine(), который позволяет считывать информацию, вводимую с клавиатуры.

Например,

```
string name = Console.ReadLine();  
Console.WriteLine("Name — {0}", name);
```

поместит введенную пользователем строку в переменную строкового типа name, а далее выведет на экран строку, содержащую значение введенной переменной.

Метод ReadLine() помещает значение, которое введено с клавиатуры, в строковый тип данных. Пусть нам необходимо ввести в программу числовое значение. Для этого можно использовать класс Convert и его методы.

Статические методы класса `Convert` используются для поддержки преобразования в базовый тип данных и из него в .NET Framework. Поддерживаются следующие базовые типы: `Boolean`, `Char`, `SByte`, `Byte`, `Int16`, `Int32`, `Int64`, `UInt16`, `UInt32`, `UInt64`, `Single`, `Double`, `Decimal`, `DateTime` и `String`. Некоторые из этих методов:

`Convert.ToInt32()` (преобразует к типу `int`)

`Convert.ToDouble()` (преобразует к типу `double`)

`Convert.ToDecimal()` (преобразует к типу `decimal`)

Пример,

```
Console.WriteLine("Введите рост: ");  
double height = Convert.ToDouble(Console.ReadLine());
```

В результате в переменную `height` типа `double` поместиться введенное пользователем число, обозначающее рост человека. Это возможно благодаря использованию метода `ToDouble` класса `Convert`.

Буфер экрана и окно консоли

Двумя тесно связанными между собой элементами консоли являются буфер экрана и окно консоли. Текст в реальности считывается из потоков и записывается в потоки, принадлежащие консоли, но кажется, что он считывается из принадлежащей консоли области, называемой буфером экрана, и записывается в эту область. Буфер экрана является атрибутом консоли, он организован в виде прямоугольной сетки строки и столбцов, в которой каждое пересечение строки и столбца, или символьная ячейка, содержит один символ. Каждый символ имеет собственный цвет и каждая символьная ячейка имеет собственный цвет фона.

Содержимое буфера экрана просматривается через прямоугольную область, называемую окном консоли. Окно консоли является еще одним атрибутом консоли; это не сама консоль, которая является окном операционной системы. Окно консоли также организовано в виде строк и столбцов, а его размер меньше или равен размеру буфера экрана; при этом окно консоли можно перемещать для просмотра разных областей буфера экрана, с которым оно связано. Если буфер экрана больше окна консоли, в этом окне автоматически отображаются полосы прокрутки, чтобы окно консоли можно было репозиционировать относительно буфера.

Курсор задает позицию в буфере экрана, откуда в данный момент осуществляется чтение и куда осуществляется запись данных. Курсор можно скрывать или отображать, можно также менять его высоту. Если курсор отображается, положение окна консоли автоматически изменяется, чтобы он постоянно был видим.

Началом координат сетки символьных ячеек в буфере экрана является его верхний левый угол, и позиция курсора в окне консоли измеряется относительно этого начала координат. Для указания позиции используются индексы, отсчитываемые от нуля, то есть самая верхняя строка и крайний слева столбец имеют индекс 0. Максимальное значение индексов строк и столбцов равно `Int16.MaxValue`.

Класс `Console` содержит методы для чтения с консоли отдельных символов или целых строк, а также несколько методов записи, автоматически преобразующие экземпляр типа значения, массив символов или набор объектов в форматированную или неформатированную строку и затем записывающие эту строку, а за ней необязательную строку, идентифицирующую конец строки данных. Также класс `Console` содержит методы и свойства, позволяющие получить или задать размер буфера экрана, окна консоли и курсора, изменить положение окна консоли и курсора, переместить или удалить данные в буфере экрана, изменить цвета фона и текста, изменить текст, отображаемый в строке заголовка консоли и воспроизвести звуковой сигнал.

Класс `Console` также включает элементы, которые поддерживают настройку внешнего вида самого окна консоли. Можно вызвать метод `SetWindowSize`, чтобы одновременно изменить количество строк и столбцов в окне консоли, или же использовать свойства `WindowHeight` и `WindowWidth`, чтобы изменить количество строк или столбцов независимо друг от друга. Используя свойства `ForegroundColor` и `BackgroundColor`, можно управлять цветом окна консоли, а с помощью свойств `CursorSize` и `CursorVisible` – настроить курсор в окне консоли.

В текстовом режиме используется только 16 цветов, которые необходимо отслеживать. Это можно сделать объектами класса `System.Console`: □

- свойство `ForegroundColor` задает цвет буквы; □

- свойство `BackgroundColor` задает задний фон под буквой; □

- оба этих свойства используют перечисление `ConsoleColor`, через которое задается цвет; □

- метод `ResetColor()` сбрасывает значения цвета текста и фона.

Например,

```
Console.SetWindowSize(40, 40);
```

```
Console.BackgroundColor = ConsoleColor.White;
```

```
Console.Clear();
```

```
Console.ForegroundColor = ConsoleColor.Blue;
```

Данный код, во-первых, изменяет размер окна консоли, устанавливая его высоту в 40 строк и ширину в 40 столбцов. Далее, присваивая свойству `BackgroundColor` значение `White` перечисления `ConsoleColor`, меняем цвет фона окна консоли. После этого, вызовом `Clear()` очищаем буфер консоли и её окна от отображаемой информации, после меняем цвет выводимого текста на `Blue`. Значения перечисления `ConsoleColor` указаны ниже.

ПОЛЯ

| | | |
|-------------|----|--|
| Black | 0 | Черный цвет. |
| Blue | 9 | Синий цвет. |
| Cyan | 11 | Голубой цвет (сине-зеленый). |
| DarkBlue | 1 | Темно-синий цвет. |
| DarkCyan | 3 | Темно-голубой цвет (темный сине-зеленый). |
| DarkGray | 8 | Темно-серый цвет. |
| DarkGreen | 2 | Темно-зеленый цвет. |
| DarkMagenta | 5 | Темно-пурпурный цвет (темный фиолетово-красный). |
| DarkRed | 4 | Темно-красный цвет. |
| DarkYellow | 6 | Темно-желтый цвет (коричнево-желтый). |
| Gray | 7 | Серый цвет. |
| Green | 10 | Зеленый цвет. |
| Magenta | 13 | Пурпурный цвет (фиолетово-красный). |
| Red | 12 | Красный цвет. |
| White | 15 | Белый цвет. |
| Yellow | 14 | Желтый цвет. |

НАИМЕНОВАНИЕ: Файловый ввод-вывод на языке C#

1. ЦЕЛЬ: научиться применять классы для работы с файлами, научиться использовать потоки ввода-вывода.

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Программирование на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

- 5.1 Создать простое консольное приложение на C#.
- 5.2 В нем написать код создания файла.
- 5.3 Открыть созданный файл.
- 5.4 Записать в него свою фамилию.
- 5.5 Закрыть файл.
- 5.6 Снова открыть файл, считать из него информацию.
- 5.7 Вывести считанную информацию на консоль.

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

- 6.1 Получить допуск к работе;
- 6.2 Выполнить задания.
- 6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

- 7.1 Наименование, цель занятия;
- 7.2 Выполненные задания (листинг программ);
- 7.3 Ответы на контрольные вопросы;
- 7.4 Вывод о проделанной работе.

8. Контрольные вопросы для зачета:

- 8.1 Что такое файл? Файловая система?
- 8.2 Какие операции могут производиться с файлами?
- 8.3 За что отвечает класс FileInfo? Примеры его членов
- 8.4 Что такое класс Stream в C#?
- 8.5 Когда использовать класс StreamWriter, а когда StreamReader?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Класс FileInfo предназначен для организации доступа к физическому файлу, который содержится на жестком диске компьютера. Он позволяет получать информацию об этом файле (например, о времени его создания, размере, атрибутах), а также производить различные операции, например, по созданию файла или его удалению.

| Член | Описание |
|---------------|--|
| AppendText() | Создает объект StreamWriter для добавления текста к файлу. |
| CopyTo() | Копирует уже существующий файл в новый файл. |
| Create() | Создает новый файл и возвращает объект FileStream для взаимодействия с этим файлом. |
| CreateText() | Создает объект StreamWriter для записи текстовых данных в новый файл. |
| Delete() | Удаляет файл, которому соответствует объект FileInfo. |
| Directory | Возвращает каталог, в котором расположен данный файл. |
| DirectoryName | Возвращает полный путь к данному файлу в файловой системе. |
| Length | Возвращает размер файла. |
| MoveTo() | Перемещает файл в указанное пользователем место (этот метод позволяет одновременно переименовать данный файл). |
| Name | Позволяет получить имя файла. |
| Open() | Открывает файл с указанными пользователем правами доступа на чтение, запись или совместное использование с другими пользователями. |
| OpenRead() | Создает объект FileStream, доступный только для чтения. |
| OpenText() | Создает объект StreamReader (о нем также будет рассказано ниже), который позволяет считывать информацию из существующего текстового файла. |
| OpenWrite() | Создает объект FileStream, доступный для чтения и записи. |

Большинство методов FileInfo возвращает объекты классов FileStream, StreamWriter, StreamReader и т. п., которые позволяют различным образом взаимодействовать с файлом, например, производить чтение или запись в него.

Например:

```
// Создание объекта FileInfo
FileInfo f = new FileInfo("text.txt");

// Создание файла и потока
StreamWriter fOut =
    new StreamWriter(f.Create());

// Запись текста в файл
fOut.WriteLine("ОДИН ДВА ТРИ...");

// Закрытие файла
fOut.Close();

// Получение информации о файле
Console.WriteLine("Name: {0}", f.Name);
Console.WriteLine("File size: {0}",
    f.Length);
Console.WriteLine("Creation: {0}",
    f.CreationTime);
Console.WriteLine("Attributes: {0}",
    f.Attributes.ToString());
```

Доступ к физическим файлам можно получать и через статические методы класса File. Большинство методов объекта FileInfo представляют в этом смысле зеркальное отражение

методов объекта File. Программы на языке C# выполняют операции ввода-вывода посредством потоков, которые построены на иерархии классов. Поток (stream) – это абстракция, которая генерирует и принимает данные. С помощью потока можно читать данные из различных источников (клавиатура, файл, память) и записывать в различные источники (принтер, экран, файл, память).

Центральную часть потоковой системы C# занимает класс Stream пространства имен System.IO. Класс Stream представляет байтовый поток и является базовым для всех остальных потоковых классов. Производными от класса Stream являются классы потоков:

1. FileStream – байтовый поток, разработанный для файлового ввода-вывода.

2. BufferedStream – заключает в оболочку байтовый поток и добавляет буферизацию, которая во многих случаях увеличивает производительность программы.

3. MemoryStream – байтовый поток, который использует память для хранения данных.

Программист может реализовать собственные потоковые классы. Однако для подавляющего большинства приложений достаточно встроенных потоков.

Символьный поток.

Чтобы создать символьный поток нужно поместить объект класса Stream (например FileStream) внутрь объекта класса StreamWriter или объекта класса StreamReader. В этом случае байтовый поток будет автоматически преобразовываться в символьный.

Класс StreamWriter предназначен для организации выходного символьного потока. В нем определено несколько конструкторов. Один из них записывается следующим образом:

```
public StreamWriter(Stream stream);
```

Параметр stream определяет имя уже открытого байтового потока. Этот конструктор генерирует исключение типа ArgumentException, если поток stream не открыт для вывода, и исключение типа ArgumentNullException, если он (поток) имеет null-значение.

Другой вид конструктора позволяет открыть поток сразу через обращения к файлу:

```
public StreamWriter(string path);
```

Параметр path определяет имя открываемого файла. Например, создать экземпляр класса

StreamWriter можно следующим образом:

```
StreamWriter fileOut =  
    new StreamWriter(  
        new FileStream(  
            "ФайлНиколаева.txt",  
            FileMode.Create,  
            FileAccess.Write));
```

И еще один вариант конструктора StreamWriter:

```
public StreamWriter(string path, bool append);
```

Параметр path определяет имя открываемого файла, а параметр append может принимать значение true – если нужно добавлять данные в конец файла, или false – если файл необходимо перезаписать.

```
// Добавляем символы в конец файла  
StreamWriter fileOut_1 =  
    new StreamWriter("МойФ.txt", true);
```

Объявив таким образом переменную `fileOut_1` для записи данных в поток можно обратиться к методу `WriteLine`:

```
// Использование WriteLine
fileOut_1.WriteLine("Строка для записи");
```

В данном случае для записи используется метод, аналогичный статическому методу класса `Console`. Это действительно схожие механизмы ввода-вывода.

Класс `StreamReader` предназначен для организации входного символьного потока. Один из его конструкторов выглядит следующим образом:

```
public StreamReader(Stream stream);
```

Параметр `stream` определяет имя уже открытого байтового потока. Этот конструктор генерирует исключение типа `ArgumentException`, если поток `stream` не открыт для ввода. Создать экземпляр класса `StreamReader` можно следующим образом:

```
StreamReader fileIn =
    new StreamReader(
        new FileStream(
            "text.txt",
            FileMode.Open,
            FileAccess.Read));
```

Как и в случае с классом `StreamWriter` у класса `StreamReader` есть и другой вид конструктора, который позволяет открыть файл напрямую:

```
public StreamReader(string path);
```

Параметр `path` определяет имя открываемого файла. Обратиться к данному конструктору можно следующим образом:

```
StreamReader fileIn =
    new StreamReader(
        @"c:\temp\Николаев.txt");
```

В C# символы реализуются кодировкой `Unicode`. Для того, чтобы можно было обрабатывать текстовые файлы, содержащие русские символы, созданные, например, в Блокноте, рекоме дуется вызывать следующий вид конструктора `StreamReader`:

```
StreamReader fileIn =
    new StreamReader(
        @"c:\temp\t.txt",
        Encoding.GetEncoding(1251));
```

Параметр `Encoding.GetEncoding(1251)` говорит о том, что будет выполняться преобразование из кода `Windows-1251` (одна из модификаций кода `ASCII`, содержащая русские символы) в `Unicode`. Тип `Encoding` реализован в пространстве имен `System.Text`.

Для чтения данных из потока `fileIn` можно воспользоваться методом `ReadLine`. При этом если будет достигнут конец файла, то метод `ReadLine` вернет значение `null`.

Рассмотрим пример, в котором данные из одного файла копируются в другой, но уже с использованием классов `StreamWriter` и `StreamReader`.

```
static void Main(string[] args)
{
    // Создание символьных потоков
    StreamReader fileIn =
        new StreamReader(
            "ФайлТекстом.txt",
            Encoding.GetEncoding(1251));
    StreamWriter fileOut =
        new StreamWriter(
            "НовыйФайл.txt",
            false);
    string line;

    // Построчное считывание
    while ((line = fileIn.ReadLine()) != null)
    {
        // ... и запись строки
        fileOut.WriteLine(line);
    }

    fileIn.Close();
    fileOut.Close();
}
```

В данном примере осуществляется копирование содержимого одного символьного файла в другой.

Таким образом, данный способ копирования одного файла в другой, дает тот же результат, что и при использовании байтовых потоков. Однако, его работа будет менее эффективной, т.к. будет тратиться дополнительное время на преобразование байтов в символы. У символьных потоков есть и свои преимущества. Например, можно использовать регулярные выражения для поиска заданных фрагментов текста в файле.

ПРАКТИЧЕСКАЯ РАБОТА №33

НАИМЕНОВАНИЕ: Реализация классов и объектов на языке C#

1. ЦЕЛЬ: Закрепление понятий о классах и объектах в C#, развитие навыков разработки программ с использованием объектно-ориентированного подхода

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Создание объектно-ориентированных приложений на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. Задания:

Спроектируйте класс, наполните его требуемой функциональностью, продемонстрируйте работоспособность класса.

| Вариант | Выражение для вычисления |
|---------|---|
| 1. | Класс «Шар». Реализовать ввод и вывод полей данных, вычисление объема, диаметра и площади поверхности, а также вывод информации об объекте. |
| 2. | Класс «Куб». Реализовать ввод и вывод полей данных, вычисление объема, площади поверхности, длины диагонали, а также вывод информации об объекте. |
| 3. | Класс «Сфера». Реализовать ввод и вывод полей данных, вычисление объема, диаметра и площади поверхности, а также вывод информации об объекте. |
| 4. | Класс «Точка в пространстве». Реализовать ввод и вывод полей данных, вычисление расстояния до введенной пользователем точки, расстояния от начала координат, а также вывод информации об объекте. |
| 5. | Класс «График $y=x$ ». Реализовать ввод и вывод полей данных, вычисление интеграла функции от a до b (вводятся пользователем), длины отрезка функции от $(a, y(a))$ до $(b, y(b))$, а также вывод информации об объекте. |
| 6. | Класс «Шар». Реализовать ввод и вывод полей данных, вычисление объема, диаметра и площади поверхности, а также вывод информации об объекте. |
| 7. | Класс «Матрица $M \times N$ ». Реализовать инициализацию элементов матрицы случайными числами, вывод матрицы, нахождение максимального и минимального элементов, а также вывод информации об объекте. |
| 8. | Класс «Прямоугольный треугольник». Реализовать ввод и вывод полей данных, вычисление гипотенузы, площади и периметра, а также вывод информации об объекте. |

| | |
|------------|---|
| 9. | Класс «Отрезок». Реализовать ввод и вывод полей данных (координаты начала и координаты конца отрезка), вычисление длины, расстояний начала и конца отрезка от начала координат, а также вывод информации об объекте. |
| 10. | Класс «Цилиндр». Реализовать ввод и вывод полей данных, вычисление объема, площади поверхности, а также вывод информации об объекте. |
| 11. | Класс «Ромб». Реализовать ввод и вывод полей данных (диагонали ромба), вычисление площади, периметра, а также вывод информации об объекте. |
| 12. | Класс «Точка в пространстве». Реализовать ввод и вывод полей данных, вычисление расстояния до введенной пользователем точки, расстояния от начала координат, а также вывод информации об объекте. |
| 13. | Класс «График $y=3x+5$ ». Реализовать ввод и вывод полей данных, вычисление интеграла функции от a до b (вводятся пользователем), длины отрезка функции от $(a, y(a))$ до $(b, y(b))$, а также вывод информации об объекте. |
| 14. | Класс «Матрица $M \times N$ ». Реализовать инициализацию элементов матрицы случайными числами, вывод транспонированной матрицы, нахождение среднего арифметического всех элементов, а также вывод информации об объекте. |
| 15. | Класс «Отрезок в пространстве». Реализовать ввод и вывод полей данных (координаты начала и координаты конца отрезка), вычисление длины, расстояний начала и конца отрезка от начала координат, а также вывод информации об объекте. |
| 16. | Класс «График $y=x-10$ ». Реализовать ввод и вывод полей данных, вычисление интеграла функции от a до b (вводятся пользователем), длины отрезка функции от $(a, y(a))$ до $(b, y(b))$, а также вывод информации об объекте. |
| 17. | Класс «Матрица $M \times N$ ». Реализовать инициализацию элементов матрицы случайными числами, вывод транспонированной матрицы, нахождение и вывод среднего арифметического элементов в каждом столбце. |
| 18. | Класс «Куб». Реализовать ввод и вывод полей данных, вычисление объема, площади поверхности, длины диагонали, а также вывод информации об объекте. |
| 19. | Класс «Сфера». Реализовать ввод и вывод полей данных, вычисление объема, диаметра и площади поверхности, а также вывод информации об объекте. |
| 20. | Класс «Точка в пространстве». Реализовать ввод и вывод полей данных, вычисление расстояния до введенной пользователем точки, расстояния от начала координат, а также вывод информации об объекте. |

| | |
|-----|---|
| 21. | Класс «График $y=x$ ». Реализовать ввод и вывод полей данных, вычисление интеграла функции от a до b (вводятся пользователем), длины отрезка функции от $(a, y(a))$ до $(b, y(b))$, а также вывод информации об объекте. |
| 22. | Класс «Точка в пространстве». Реализовать ввод и вывод полей данных, вычисление расстояния до введенной пользователем точки, расстояния от начала координат, а также вывод информации об объекте. |
| 23. | Класс «График $y=-x$ ». Реализовать ввод и вывод полей данных, вычисление интеграла функции от a до b (вводятся пользователем), длины отрезка функции от $(a, y(a))$ до $(b, y(b))$, а также вывод информации об объекте. |
| 24. | Класс «Цилиндр». Реализовать ввод и вывод полей данных, вычисление объема, площади поверхности, а также вывод информации об объекте. |
| 25. | Класс «Отрезок в пространстве». Реализовать ввод и вывод полей данных (координаты начала и координаты конца отрезка), вычисление длины, расстояний начала и конца отрезка от начала координат, а также вывод информации об объекте. |

6. Порядок проведения занятия:

- 6.1 Получить допуск к работе;
- 6.2 Выполнить задания.
- 6.3 Оформить отчет.

7. Содержание отчета:

- 7.1 Наименование, цель занятия;
- 7.2 Выполненные задания (листинг программ);
- 7.3 Ответы на контрольные вопросы;
- 7.4 Вывод о проделанной работе.

8. Контрольные вопросы для зачета:

- 8.1 Что такое класс?
- 8.2 Что такое члены класса? Какие группы членов класса вы знаете?
- 8.3 Какие типы функций-членов класса вы знаете?
- 8.4 Какое ключевое слово используется для создания объекта класса?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Класс – это тип данных, объединяющий данные и методы их обработки. Класс – это пользовательский шаблон, в соответствии с которым можно создавать объекты. То есть класс – это правило, по которому будет строиться объект. Сам класс не содержит данных.

Объект класса (экземпляр класса) – переменная типа класс. Объект содержит данные и методы, манипулирующие этими данными. Класс определяет, какие данные содержит объект и каким образом он ими манипулирует.

Все что справедливо для классов можно распространить и на структуры. Отличие состоит в методе хранения объектов данных типов в оперативной памяти: структуры – это типы по значению, они размещаются в стеке; классы – это ссылочные типы, объекты классов размещаются в куче. Структуры не поддерживают наследование.

Объявление классов – с помощью ключевого слова `class`.

При создании как классов, так и структур, используется ключевое слово `new`.

Пример объявления класса:

```
// Объявление класса
public class MyFirstClass
{
    // Данные-члены класса

    // Доступные на уровне экземпляра
    public int a;
    public float b__;
    public string fio;

    // Доступные только на уровне класса
    private bool IsOK;
    private double precision;
}
```

Данные и функции, объявленные внутри класса, называются членами класса (class members). Доступность членов класса может быть описана как public, private, protected, internal или internal protected.

Данные-члены – это те структуры внутри класса, которые содержат данные класса – поля, константы, события.

Поля – это любые переменные, ассоциированные с классом. После создания экземпляра класса к полям можно обращаться.

Функции-члены – это члены, которые обеспечивают некоторую функциональность для манипулирования данными классов. Они делятся на следующие виды: методы, свойства, конструкторы, финализаторы, операции и индексаторы.

Методы (method) – это функции, ассоциированные с определенным классом. Как и данные-члены, по умолчанию они являются членами экземпляра. Они могут быть объявлены статически с помощью модификатора static.

Свойства (property) – это наборы функций, которые могут быть доступны клиенту таким же способом, как общедоступные поля класса. В C# предусмотрен специальный синтаксис для реализации чтения и записи свойств для классов, поэтому писать собственные методы с именами, начинающимися на Set и Get, не понадобится. Поскольку не существует какого-то отдельного синтаксиса для свойств, который отличал бы их от нормальных функций, создается иллюзия объектов как реальных сущностей, предоставляемых клиентскому коду.

Конструкторы (constructor) – это специальные функции, вызываемые автоматически при инициализации объекта. Их имена совпадают с именами классов, которым они принадлежат, и они не имеют типа возврата. Конструкторы полезны для инициализации полей класса.

Финализаторы (finalizer) похожи на конструкторы, но вызываются, когда среда CLR определяет, что объект больше не нужен. Они имеют то же имя, что и класс, но с предшествующим символом тильды (~). Предсказать точно, когда будет вызван финализатор, невозможно.

Операции (operator) – это простейшие действия вроде + или -. Когда вы складываете два целых числа, то, строго говоря, применяете операцию + к целым. Однако C# позволяет указать, как существующие операции будут работать с пользовательскими классами (так называемая перегрузка операций).

Индексаторы (indexer) позволяют индексировать объекты таким же способом, как массив или коллекцию.

В C# объявление метода класса состоит из спецификатора доступности, возвращаемого значения, имени метода, списка формальных параметров и тела метода:

```
[модификатор] тип_возврата имя_метода ([список_параметров])
{
    // тело метода
}
```

Например, добавим методы для объявленного ранее класса MyFirstClass:

```

// Объявление класса
public class MyFirstClass
{
    // Данные-члены класса

    // Доступные на уровне экземпляра
    public int a;
    public float b__;
    public string fio;

    // Доступные только на уровне класса
    private bool IsOK;
    private double precision;

    // Метод для инициализации некоторых полей класса
    public void InitClassMembers(int pA, float pB__, string pFio)
    {
        a = pA;
        b__ = pB__;
        fio = pFio;
    }

    // Метод, возвращающий значение вычисленное на основе полей класса
    public int GetAbsA()
    {
        return Math.Abs(a);
    }
}

```

Синтаксис вызова методов аналогичен синтаксису обращения к данным членам:

```

MyFirstClass obj = new MyFirstClass();

obj.InitClassMembers(10, 0.8F, "Новиков П.Е.");

int abs_a = obj.GetAbsA();

```

В данном примере метод `InitClassMembers` не возвращает никаких данных, но требует передачи ему фактических параметров. В свою очередь, метод `GetAbsA` возвращает значение типа `int` и не предполагает никаких параметров.

Пример выполнения задания.

Разработать класс для представления объекта «Прямоугольный параллелепипед». Реализуйте все необходимые поля данных (закрытые) и методы позволяющие:

- устанавливать и считывать значения полей данных;
- вычислять объем прямоугольного параллелепипеда;
- вычислять площадь поверхности прямоугольного параллелепипеда;
- выводить полную информацию об объекте в консоль.

Решение данной задачи состоит из двух этапов: объявление класса `Parallelepiped` и демонстрация использования объекта данного класса.

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

namespace LR_Three
{
    class Parallelepiped
    {
        // Поля данных представляют стороны параллелепипеда
        private double a, b, c;

        // Методы для установки и считывания значений полей
        public void Set_a(double pa) { a = pa; }
        public double Get_a() { return a; }

        public void Set_b(double pb) { b = pb; }
        public double Get_b() { return b; }

        public void Set_c(double pc) { c = pc; }
        public double Get_c() { return c; }

        // Метод для вычисления объема прямоугольного параллелепипеда
        public double GetV()
        {
            return a * b * c;
        }

        // Метод для вычисления площади поверхности прямоугольного параллелепипеда
        public double GetS()
        {
            return 2 * (a * b + b * c + a * c);
        }
    }
}
```

```

// Метод для вывода полной информации об объекте в консоль
public void PrintFullInformation()
{
    string str = "*****\n" +
                "*" +
                "    Объект прямоугольный параллелепипед    " +
                "*" +
                "*****";
    Console.WriteLine(str);

    Console.WriteLine("Стороны прямоугольного параллелепипеда:\n" +
        "высота = {0}\n" +
        "длина = {1}"+
        "ширина = {2}", a, b, c);

    Console.WriteLine("Объем параллелепипеда равен {0}", GetV());

    Console.Write("Площадь поверхности равна {0}", GetS());
}

class Program
{
    static void Main(string[] args)
    {
        Console.Title = "Прямоугольный параллелепипед";
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.BackgroundColor = ConsoleColor.Red;
        Console.Clear();

        Parallelepiped p;

        p = new Parallelepiped();
        p.Set_a(10.5);
        p.Set_b(25.67);
        p.Set_c(40);

        p.PrintFullInformation();

        Console.ReadKey();
    }
}

```

В данном примере необходимо обратить внимание на тот факт, что все вычисления выполняются внутри класса. Метод Main содержит только вызовы методов класса, то есть вся реализация скрыта.

НАИМЕНОВАНИЕ: Конструктор класса, перегруженные конструкторы

1. ЦЕЛЬ: Закрепление понятий конструкторах классов в C#, их перегрузке, развитие навыков разработки программ с использованием объектно-ориентированного подхода

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Создание объектно-ориентированных приложений на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. Задания:

1. Разработать класс для представления объекта множество символов. Определить конструктор с одним параметром-строкой, который задаёт элементы множества и который можно использовать как конструктор умолчания. Определить конструктор, который копирует в новое множество элементы другого множества, не превышающие заданное значение, который можно использовать как конструктор копирования. Определить деструктор.

2. Разработать класс для представления объекта строка. Определить конструктор с одним параметром целого типа – длина строки, который можно использовать как конструктор умолчания. Определить конструктор, который копирует в новую строку n первых символов другой строки и который можно использовать как конструктор копирования. Определить деструктор.

3. Разработать класс для представления объекта матрица, состоящая из элементов типа char. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

4. Разработать класс для представления объекта матрица, состоящая из элементов типа byte. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

5. Разработать класс для представления объекта матрица, состоящая из элементов типа short. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

6. Разработать класс для представления объекта матрица, состоящая из элементов uint. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

7. Разработать класс для представления объекта матрица, состоящая из элементов типа double. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на

новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

18. Разработать класс для представления объекта матрица, состоящая из элементов типа long. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

19. Разработать класс для представления объекта матрица, состоящая из элементов типа int. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

20. Разработать класс для представления объекта матрица, состоящая из элементов типа long. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

21. Разработать класс для представления объекта матрица, состоящая из элементов типа float. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

22. Разработать класс для представления объекта множество символов. Определить конструктор с одним параметром-строкой, который задаёт элементы множества и который можно использовать как конструктор умолчания. Определить конструктор, которые копирует в новое множество элементы другого множества, не превышающие заданное значение, который можно использовать как конструктор копирования. Определить деструктор.

23. Разработать класс для представления объекта строка. Определить конструктор с одним параметром целого типа – длина строки, который можно использовать как конструктор умолчания. Определить конструктор, который копирует в новую строку n первых символов другой строки и который можно использовать как конструктор копирования. Определить деструктор.

24. Разработать класс для представления объекта матрица, состоящая из элементов типа char. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

25. Разработать класс для представления объекта матрица, состоящая из элементов типа bool. Определить конструктор с двумя параметрами целого типа – размерность матрицы, который можно использовать как конструктор умолчания. Определить конструктор, который создаёт новую матрицу таким образом, что все её элементы больше элементов другой матрицы на заданное число, и который можно использовать как конструктор копирования. Определить деструктор.

6. Порядок проведения занятия:

6.1 Получить допуск к работе;

6.2 Выполнить задания.

6.3 Оформить отчет.

7. Содержание отчета:

- 7.1 Наименование, цель занятия;
- 7.2 Выполненные задания (листинг программ);
- 7.3 Ответы на контрольные вопросы;
- 7.4 Вывод о проделанной работе.

8. Контрольные вопросы для зачета:

- 8.1 Что такое конструктор?
- 8.2 Как обозначается конструктор?
- 8.3 Что такое деструктор?
- 8.4 Зачем может понадобиться перегрузка конструктора?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Пример 1.1

```
using System;

public class Time
{
    // открытые методы
    public void DisplayCurrentTime()
    {
        Console.WriteLine(" загрузка для DisplayCurrentTime")
    }

    // закрытые переменные
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;
}

public class Tester
{
    static void Main()
    {
        Time t = new Time();
        t.DisplayCurrentTime();
    }
}
```

Единственный метод, объявленный в определении класса Time, — метод DisplayCurrentTime(). Тело этого метода определено внутри определения класса. C# не требует объявления методов до их описания. В C# все методы встраиваются в определение класса, как метод DisplayCurrentTime(). В определении метода DisplayCurrentTime(); указано, что тип возвращаемого значения — void, то есть он не возвращает значения методу, вызвавшему его. В конце определения класса Time объявляется ряд переменных: int Year, int Month, int Date, int Hour, int Minute, int Second.

После закрывающей фигурной скобки определен другой класс, `Tester`. Он включает в себя метод `Main()`. В методе `Main()` создается экземпляр класса `Time`, и его адрес присваивается объекту `t`. Поскольку `t` является экземпляром `Time`, метод `Main()` может вызывать метод `DisplayCurrentTime()`, предоставляемый объектами этого типа, и тем самым выводить на экран текущее время.

Желательно определять переменные класса как закрытые (`private`). Это гарантирует, что только методы того же класса будут иметь доступ к их значениям. Поскольку уровень доступа `private` устанавливается по умолчанию, нет необходимости указывать его явно.

Так, в примере объявления переменных лучше переписать следующим образом:

```
// закрытые переменные
private int Year;
private int Month;
private int Date;
private int Hour;
private int Minute;
private int Second;
```

Класс `Time` и метод `DisplayCurrentTime()` объявлены открытыми, чтобы любой другой класс мог обратиться к ним.

Аргументы метода

Методы могут принимать любое количество параметров. Список параметров указывается в круглых скобках после имени метода, причем каждый параметр предваряется своим типом. Например, в следующем участке кода определяется метод по имени `MyMethod`, который возвращает `void` (то есть не возвращает значения). Он имеет два параметра с типами `int` и `button`:

```
void MyMethod (int firstParam, button secondParam)
{
    // ..
}
```

В теле метода параметры ведут себя как локальные переменные, как будто бы они были объявлены и инициализированы значениями, переданными методу. В примере 1.2 иллюстрируется передача значений методу. В этом случае значения имеют типы `int` и `float`.

Пример 1.2. Передача значений методу `SomeMethod()`

```
using System;

public class MyClass
{
    public void SomeMethod(int firstParam, float secondParam)
    { Console.WriteLine( "Получены параметры {0} {1} ", firstParam,
        secondParam); }
}

public class Tester
{
    static void Main()
    {
        int howManyPeople = 5;
```

```
float pi = 3,14f;
MyClass me = new MyClass();
me.SomeMethod(howManyPeople, pi);
}
}
```

Здесь метод `SomeMethod()` принимает значения типов `int` и `float` и выводит их с помощью метода `Console.WriteLine()`. Параметры, названные `firstParam` и `secondParam`, в теле метода `SomeMethod()` трактуются как локальные переменные.

Конструкторы

Обратите внимание на оператор, создающий объект `Time` в примере 1.1. Внешне он выглядит как вызов метода. И действительно, при создании экземпляра происходит обращение к методу. Он называется *конструктором*, и программист должен либо определить его как часть определения класса, либо положиться в этом на среду CLR, которая сама его предоставит. Задача конструктора — создать объект указанного класса и перевести его в *действующее* состояние. До вызова конструктора объект представляет собой неинициализированную область памяти; по окончании его работы эта память содержит действующий экземпляр данного класса.

Класс `Time` из примера 1.1 не определяет никакого конструктора. Если конструктор не объявлен, компилятор предоставляет его самостоятельно. Конструктор, вызванный по умолчанию, создает объект, но не предпринимает никаких других действий. Переменные класса инициализируются безвредными значениями (целые — нулем, строки — пустой строкой и т. д.).

Как правило, программисты предпочитают определять собственные конструкторы и снабжать их аргументами, чтобы конструктор мог устанавливать начальное состояние объекта. В примере 1.1 было бы разумно передавать конструктору информацию о текущем годе, месяце, дне месяца и т. д., чтобы объект был заполнен осмысленными данными. При определении конструктора объявляется метод с тем же именем, что и класс, в котором он определяется. У конструкторов нет возвращаемого типа, и они обычно объявляются открытыми. Если планируется передавать конструктору какие-либо аргументы, их список указывается, как для любого другого метода. В примере 1.3 объявляется конструктор для класса `Time`, который принимает единственный аргумент, объект типа `DateTime`.

Пример 1.3. Объявление конструктора

```
public class Jim
{ // открытые методы доступа
public void DisplayCurrentTime()
{   System.Console.WriteLine(   "{0},{1},{2},{3},{4},{5}",   Month,
Date, Year, Hour, Minute, Second );
}
// конструктор
public Time(System.DateTime dt)
{
Year = dt.Year;
Month = dt.Month;
Date = dt.Day;
Hour = dt.Hour;
Minute = dt.Minute;
Second = dt.Second;
```

```

    }
    // закрытые переменные класса
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;
}

public class Tester
{
    static void Main()
    {
        System. DateTime currentTime = System. DateTime. Now;
        Time t =new Time(currentTime);
        t.DisplayCurrentTime ( ) ;
    }
}

```

В этом примере конструктор принимает объект `DateTime` и инициализирует все переменные класса, беря за основу значения из этого объекта.

После того как конструктор закончит свою работу, в памяти компьютера будет находиться объект `Time` с инициализированными значениями. Когда метод `Main()` вызывает метод `DisplayCurrentTime()`, эти значения выводятся на экран. Закомментируйте какое-нибудь присваивание и снова выполните программу. Окажется, что компилятор инициализирует соответствующую переменную нулем. Целые переменные класса устанавливаются в 0, если не указано другое значение. Помните, что переменные, имеющие размерный тип (например целочисленные), не могут оставаться *неинициализированными*; если конструктор не получит конкретных указаний, он установит значения по умолчанию.

В примере 1.3 в методе `Main()` класса `Tester` создается объект `DateTime`. Этот объект, поставляемый библиотекой `System`, предлагает ряд открытых значений: `Year`, `Month`, `Day`, `Hour`, `Minute` и `Second`, которые в точности соответствуют переменным объекта `Time`. Кроме того, объект `DateTime` предоставляет статический метод `Now()`, который возвращает ссылку на экземпляр объекта `DateTime`, инициализированный текущим временем.

Рассмотрим выделенную строчку в методе `Main()`, где объект `DateTime` создается вызовом статического метода `Now()`. Он создает `DateTime` в куче и возвращает ссылку на него.

Возвращенная ссылка присваивается переменной `currentTime`, которая объявлена как ссылка на объект `DateTime`. Затем `currentTime` передается в качестве параметра конструктору `Time`. Параметр этого конструктора, `dt`, тоже представляет собой ссылку на объект `DateTime`. Теперь `dt` ссылается на тот же объект, что и переменная `currentTime`. Таким образом, конструктор `Time` получает доступ к открытым переменным объекта `DateTime`, созданного в методе `Tester. Main()`. Когда объект передается в качестве параметра, он передается *по ссылке*, то есть передается указатель на объект, а копия объекта не создается.

Инициализаторы

Вместо того чтобы инициализировать значения переменных класса в каждом конструкторе, можно сделать это в *инициализаторе*. Инициализатор создается присваиванием начального значения элементу класса:

```
private int Second = 30; // инициализатор
```

Предположим, семантика объекта Time такова, что независимо от установленного времени секунды всегда инициализируются значением 30.

Класс Time можно переписать с применением инициализатора. Тогда, какой бы конструктор ни был вызван, переменная Second будет всегда получать начальное значение либо явно, от конструктора, либо неявно, от инициализатора. Это продемонстрировано в примере 1.4.

Пример 1.4. Использование инициализатора

```
public class Time
{
    // открытые методы доступа
    public void DisplayCurrentTime()
    {
        System. DateTime now = System.DateTime.Now;
        System. Console . WriteLine("\nОтладка\t: {0}/{1}/{2} {3} : {4} : {5}»",now. Month, now. Day, now.Year, now.Hour, now. Minute, now. Second);
    }
    //конструкторы
    public Time(System,DateTime dt)
    {
        Year = dt.Year;
        Month = dt.Month;
        Date = dt.Day;
        Hour = dt.Hour;
        Minute = dt.Minute;
        Second = dt.Second;//явное присваивание
    }
    public Time(int Year, int Month, int Date, int Hour, int Minute)
    { this.Year = Year;
      this.Month =Month;
      this.Date = Day;
      this.Hour = Hour;
      this.Minute = Minute;
    }
    // закрытые переменные класса
    private int Year;
    private int Month;
    private int Date;
```



```

private int Hour;
private int Minute;
private int Second = 30; // инициализатор
}

public class Tester
{
    static void Main()
    {
        System. DateTime currentTime = System. DateTime. Now;
        Time t =new Time(currentTime);
        t.DisplayCurrentTime ( ) ;
        Time t2 =new Time(2007,02,20,17,24);
        T2.DisplayCurrentTime ( ) ;
    }
}

```

Если инициализатор не указан, конструктор присвоит каждой переменной нулевое начальное значение. Однако в приведенном примере переменная Second получает значение 30:

```
private int Second = 30; // инициализатор
```

Если для переменной Second не будет передано значение, то при создании она будет проинициализирована числом 30:

```
Time t2 =new Time(2007,02,20,17,24);
T2.DisplayCurrentTime ( ) ;
```

Однако если переменной Second значение присвоено, как это делается в конструкторе, принимающем объект DateTime, новое значение замещает первоначальное.

При первом выполнении программы вызывается конструктор, принимающий объект DateTime, причем секунды устанавливаются в значение 24. Если бы у программы не было инициализатора и переменной Second не присваивалось никакого значения, то компилятор установил бы ее в ноль.

Копирующие конструкторы

Копирующий конструктор (copy constructor) создает новый объект, копируя переменные из существующего объекта того же типа. Пусть, например, требуется передать объект Time конструктору Time() так, чтобы новый объект Time содержал те же значения, что и старый. Язык C# не добавляет в класс копирующий конструктор, так что программист должен написать такой конструктор самостоятельно. Подобный конструктор всего лишь копирует элементы исходного объекта во вновь создаваемый:

```

public Time(Time existingTimeObject)
{
    Year = existingTimeObject.Year;
    Month = existingTimeObject.Month;
    Date = existingTimeObject.Date;
    HOLT = existingTimeObject.HOLT;
    Minute = existingTimeObject.Minute;
    Second = existingTimeObject.Second;
}

```

}

Копирующий конструктор вызывается путем создания объекта типа Time и передачи ему имени копируемого объекта Time:

```
Time t3= new Time (t2);
```

Здесь переменная t2 передается в качестве аргумента existingTimeQbject копирующему конструктору, который создаст новый объект Time

Уничтожение объектов

В C# предусмотрена сборка мусора, и поэтому отсутствует необходимость в явном деструкторе. Однако если объект работает с неуправляемыми ресурсами, программисту приходится явно освобождать их по завершении работы. Это делается с помощью *деструктора (destructor)*, вызываемого сборщиком мусора при уничтожении объекта.

Деструктор должен лишь освобождать ресурсы, удерживаемые объектом, и не должен ссылаться на другие объекты. Следует заметить, что если используются только управляемые ресурсы, то деструктор не нужен и не должен реализовываться. Поскольку применение деструктора сопряжено с некоторыми затратами, его необходимо применять лишь к нуждающимся в нем методам (а именно к методам, потребляющим ценные неуправляемые ресурсы).

Как работают деструкторы

Сборщик мусора ведет список объектов, имеющих деструкторы. Этот список обновляется каждый раз, когда появляется или уничтожается подобный объект. Когда объект из такого списка попадаете сборщику мусора, он помещается в очередь объектов, ожидающих уничтожения. После выполнения деструктора сборщик мусора уничтожает объект, обновляет очередь и список объектов с деструкторами.

Деструктор C#

Синтаксически деструктор в языке C# напоминает деструктор C++, однако работает он совершенно по-другому. Объявляется деструктор с помощью символа «тильда (~)»

В C# этот синтаксис служит лишь сокращенной записью объявления метода Finalize(), связывающей его с базовым классом. Таким образом, запись: преобразуется компилятором C# в:

```
protected override void Finalize()  
{  
    try  
    {  
        // выполнить действия  
    }  
    finally  
    {  
        base.Firalize();  
    }  
}}
```

Перегрузка методов и конструкторов

Нередко требуется, чтобы одно и то же имя было сразу у нескольких функций. Самым распространенным случаем является наличие нескольких конструкторов. В примерах, приводимых до сих пор, конструктор принимал один параметр, объект DateTime. Было бы удобно устанавливать в новых объектах Time произвольное время, передавая им значения года, месяца, числа, часов, минут и секунд. Было бы еще удобнее, если бы одни разработчики могли пользоваться одним конструктором, а другие — другим. Такое понятие, как перегрузка функций,

существует именно для этих ситуаций. *Сигнатура (signature)* метода определяется его именем и списком параметров. Два метода отличаются по сигнатурам, если у них разные имена или списки параметров. Списки параметров могут отличаться по количеству или типам параметров. Например, в следующем фрагменте программы первый метод отличается от второго количеством параметров, а второй от третьего — их типами.

```
Void mes( int a);
```

```
Void mes( int a , int b);
```

```
Void mes( int a , srtring b);
```

Класс может иметь любое количество методов, если их сигнатуры отличаются друг от друга. В примере 1.9 демонстрируется класс Time с двумя конструкторами, один из которых принимает объект DateTime, а второй — шесть целых чисел.

Пример 1.9, Перегрузка конструктора

```
public class Time
{
    // открытые методы доступа
    public void DisplayCurrentTime()
    {
        System. Console, WriteLine("{0}/{1}/{2}{3}: { 4 } : {5}», Month,
        Date, Year, Hour, Minute, Second);
    }
    //конструкторы
    public Time( System.DateTime dt)
    { Year =dt.Year;
      Month=dt. Month;
      Date=dt. Date;
      Hour=dt. Hour;
      Minute=dt. Minute;
      Second=dt. Second;
    }
    public Time(  int Year,  int Month,  int .Date,  int Hour,  int
    Minute,  int Second)
    {  this.Year =Year;
      this.Month=Month;
      this.Date=Date;
      this.Hour=Hour;
      this.Minute=Minute;
      this.Second=Second;
    }
    // закрытые переменные класса
    private int Year;
    private int Month;
```

```

private int Date;
private int Hour;
private int Minute;
private int Second;
}

public class Tester
{ static void Main()
{
System. DateTime currentTime = System. DateTime. Now;
Time t =new Time(currentTime);
t.DisplayCurrentTime ( ) ;
Time t2 = new Time(2007,11,18,11,03,30);
t2.DisplayCurrentTime();
}
}

```

Как видно из примера, у класса Time два конструктора. Если бы сигнатура состояла только из имени функции, компилятор не знал бы, какой конструктор вызывать при создании объектов t1 и t2. Однако поскольку сигнатура включает в себя и типы аргументов, компилятор в состоянии сопоставить вызов конструктора для t1 с описанием конструктора, требующего объект DateTime. Аналогичным образом компилятор способен связать вызов конструктора объекта t2 с конструктором, сигнатура которого содержит список из шести аргументов с типом int.

Перегружая метод, программист должен изменить сигнатуру (то есть имя метода, количество параметров или их тип). Можно, к тому же, изменить и тип возвращаемого значения, но это не обязательно. Изменение возвращаемого типа не ведет к перегрузке метода, а наличие двух методов с одинаковыми сигнатурами, но разными возвращаемы-

ми типами вызовет ошибку на этапе компиляции.

Пример 1.10. Изменение возвращаемого типа перегруженных методов

```

public class Tester
{
private int Triple(int val)
{ return 3*val;
}

private long Triple(long val)
{ return 3* val;
}

public void Test()
{
int x = 5;
int y = Triple(x);
System. Console. WnteLine("x: {0} y: {1}»», x, y);
Long lx=10;

```

```

Long ly= Triple(lx);
System. Console. WnteLine("x: {0} y: {1}», lx, l y);
}
static void Main()
{ Tester t = new Tester();
t.Test();
}
}

```

В этом примере класс Tester перегружает метод Triple (), один раз с целочисленным параметром, а другой — с параметром типа long. Возвращаемые типы у методов Triple () разные. Хотя это не обязательно, в данном случае разница возвращаемых типов очень удобна.

ПРАКТИЧЕСКАЯ РАБОТА №35

НАИМЕНОВАНИЕ: Операции классов, перегрузка операторов

1. ЦЕЛЬ: Закрепление понятий операций классов в C#, их перегрузке, развитие навыков разработки программ с использованием объектно-ориентированного подхода

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Создание объектно-ориентированных приложений на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

Составить класс, относительно которого перегрузить минимум две из операций: +, -, /, *, %, ==, >.

Вариант 1. Вектор в пространстве

Вариант 2. Матрица

Вариант 3. Сотрудник

Вариант 4. Компьютер

Вариант 5. Объект недвижимости

Вариант 6. Комплексное число

Вариант 7. Куб

Вариант 8. Товар

Вариант 9. Автомобиль

Вариант 10. Студент

Вариант 11. Книга

Вариант 12. Дисциплина (предмет)

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

6.1 Получить допуск к работе;

6.2 Выполнить задания.

6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

7.1 Наименование, цель занятия;

7.2 Выполненные задания (листинг программ);

7.3 Ответы на контрольные вопросы;

7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

8.1 Что такое перегрузка операторов? Когда применяется данный механизм?

8.2 Какие операции нельзя перегрузить?

8.3 Какие операции необходимо перегружать попарно?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Операции. Большинство операций пришло в C# из языков C, C++:

| Категория | Операции |
|-----------------------|---|
| Арифметические | +, -, *, /, % |
| Логические | &, , ^, ~, &&, , ! |
| Конкатенация строк | + |
| Инкремент и декремент | ++, -- |
| Сравнение | <, >, >=, <=, ==, != |
| Присвоение | +=, -=, /=, *=, %=, =, &=, =, ^=, <<=, >>= |

| | |
|---|-------------------------|
| Доступ к члену класса | . |
| Индексация | [] |
| Приведение | () |
| Условная (тернарная) операция | ? : |
| Создание объектов | new |
| Информация о типе | sizeof, is, as, typeof, |
| Контроль исключения, связанного с переполнением | checked, unchecked |

Перегрузка операций относительно класса. Рассмотрим простой класс, например, для представления объекта «Вектор» (для простоты возьмем вектор на плоскости). Пример 1:

```
class Vector
{
    // Поля данных представляют стороны параллелепипеда
    public double X, Y;

    // Конструкторы
    public Vector()
    {
    }

    public Vector(double XCoord, double YCoord)
    {
        X = XCoord;
        Y = YCoord;
    }
}
```

Класс создан, но в математике можно выполнять сложение (вычитание) векторов и умножение вектора на число, а также находить скалярное произведение векторов. Необходимо добавить эту возможность для нашего класса «Вектор». Это значит, что операции, указанные в примере 2 должны иметь смысл.

Пример 2:

```
Vector a = new Vector(4, 10);  
Vector b = new Vector(10, 15);  
  
// вычисление суммы векторов  
Vector SumVector = a + b;  
  
// Вычисление произведения вектора на число  
Vector MultNum = 2 * a;  
  
// вычисление скалярного произведения  
double Skalar = a * b;
```

В данной реализации (пример 1) такие операции в коде невозможны. Добавим в класс перегруженные операции.

Пример 3:


```

class Vector
{
    // Поля данных представляют стороны параллелепипеда
    public double X, Y;

    // Конструкторы
    public Vector()
    {
    }

    public Vector(double XCoord, double YCoord)
    {
        X = XCoord;
        Y = YCoord;
    }

    // Перегрузка операции сложения
    public static Vector operator +(Vector left, Vector right)
    {
        return new Vector(left.X + right.X, left.Y + right.Y);
    }

    // Перегрузка операции умножения вектора на число
    public static Vector operator *(double k, Vector vek)
    {
        return new Vector(k*vek.X, k*vek.Y);
    }

    // Перегрузка операции умножения двух векторов (скалярное умножение)
    public static double operator *(Vector left, Vector right)
    {
        return left.X * right.X + left.Y * right.Y;
    }
}

```

Перегрузка операции похожа на объявление метода класса, но существуют отличия: используется ключевое слово `static`; используется ключевое слово `operator` и символ операции (+, *, - и т.д.) вместо имени метода.

После того как операции перегружены, возможно выполнять следующие действия:

```

class Program
{
    static void Main(string[] args)
    {
        Vector a = new Vector(4, 10);
        Vector b = new Vector(10, 15);

        // вычисление суммы векторов
        Vector SumVector = a + b;

        // Вычисление произведения вектора на число
        Vector MultNum = 2 * a;

        // вычисление скалярного произведения
        double Skalar = a * b;

        // Проведение сложных расчетов
        Vector RES = 2 * a + 3 * b + (a * b) * a;

        Console.ReadKey();
    }
}

```

ПРАКТИЧЕСКАЯ РАБОТА №36

НАИМЕНОВАНИЕ: Реализация наследования в C#

1. ЦЕЛЬ: Изучение и закрепление представлений о наследовании в языке C#, развитие навыков разработки программ с использованием объектно-ориентированного подхода

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Создание объектно-ориентированных приложений на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

На основе задания реализовать иерархию классов, использовать в коде:

- один переопределенный метод;
- использование базового конструктора (в коде конструктора-потомка);
- использование в программе вызова одного базового метода (метода родительского класса).



| Вариант | Иерархия классов |
|---------|---|
| 3 | <pre> classDiagram class Геометрия class Четырехугольник class Треугольник class Квадрат class Ромб Геометрия < -- Четырехугольник Геометрия < -- Треугольник Четырехугольник < -- Квадрат Четырехугольник < -- Ромб </pre> |
| 4 | <pre> classDiagram class Издание class Книга class Журнал class ЭлРесурс Издание < -- Книга Издание < -- Журнал Издание < -- ЭлРесурс </pre> |
| 5 | <pre> classDiagram class Мультимедиа class Изображение class Видео class Растровое class Векторное Мультимедиа < -- Изображение Мультимедиа < -- Видео Изображение < -- Растровое Изображение < -- Векторное </pre> |
| 6 | <pre> classDiagram class Вооружение class Стрелковое class Танк class Сомолет Вооружение < -- Стрелковое Вооружение < -- Танк Вооружение < -- Сомолет </pre> |
| 7 | <pre> classDiagram class Машины class Авиасредство class Грузовик class Вертолет class Истребитель Машины < -- Авиасредство Машины < -- Грузовик Авиасредство < -- Вертолет Авиасредство < -- Истребитель </pre> |
| 8 | <pre> classDiagram class Растение class Трава class Дерево class Кустарник Растение < -- Трава Растение < -- Дерево Растение < -- Кустарник </pre> |

| Вариант | Иерархия классов |
|---------|--|
| 9 | <pre> graph BT ПО[ПО] --> ОС[ОС] ПО --> Прикладное[Прикладное] ОС --> СвободныеОС[СвободныеОС] ОС --> ПлатныеОС[ПлатныеОС] </pre> |
| 10 | <pre> graph BT ЭВМ[ЭВМ] --> Мобильный[Мобильный] ЭВМ --> Стационарный[Стационарный] Мобильный --> Планшет[Планшет] Мобильный --> Ноутбук[Ноутбук] </pre> |
| 11 | <pre> graph BT Электроника[Электроника] --> СотТелефон[СотТелефон] Электроника --> Фотоаппарат[Фотоаппарат] Электроника --> Планшет[Планшет] </pre> |
| 12 | <pre> graph BT ТехСредство[ТехСредство] --> Автомобиль[Автомобиль] ТехСредство --> Вертолёт[Вертолёт] Автомобиль --> ГрузовойАвто[ГрузовойАвто] Автомобиль --> ЛегковойАвто[ЛегковойАвто] </pre> |
| 13 | <pre> graph BT БытТехника[БытТехника] --> Телевизор[Телевизор] БытТехника --> Утюг[Утюг] БытТехника --> Микроволновка[Микроволновка] </pre> |
| 14 | <pre> graph BT ЭВМ[ЭВМ] --> Мобильный[Мобильный] ЭВМ --> Стационарный[Стационарный] Мобильный --> Планшет[Планшет] Мобильный --> Ноутбук[Ноутбук] </pre> |

| Вариант | Иерархия классов |
|---------|---|
| 15 | <pre> graph BT Ручка --> КанцТовары Органайзер --> КанцТовары Скоросшиватель --> КанцТовары </pre> |
| 16 | <pre> graph BT Вертолет --> Авиасредство Истребитель --> Авиасредство Авиасредство --> Грузовик Грузовик --> Машины </pre> |
| 17 | <pre> graph BT Квартира --> Недвижимость Дом --> Недвижимость Гараж --> Недвижимость </pre> |
| 18 | <pre> graph BT Трактор --> Тяжелое Комбайн --> Тяжелое Тяжелое --> Среднее Среднее --> Машиностроение </pre> |
| 19 | <pre> graph BT Фотон --> ЭлемЧастица Электрон --> ЭлемЧастица Протон --> ЭлемЧастица </pre> |
| 20 | <pre> graph BT Прямоугольник --> Многоугольник Треугольник --> Многоугольник Многоугольник --> Эллипс Эллипс --> Геометрия </pre> |
| 21 | <pre> graph BT Программист --> Специалист Врач --> Специалист Летчик --> Специалист </pre> |

| Вариант | Иерархия классов |
|---------|--|
| 22 | <pre> graph BT ПО[ПО] --> ОС[ОС] ПО --> Прикладное[Прикладное] ОС --> СвободныеОС[СвободныеОС] ОС --> ПлатныеОС[ПлатныеОС] </pre> |
| 23 | <pre> graph BT БанкКарта[БанкКарта] --> Накопительная[Накопительная] БанкКарта --> Платежная[Платежная] БанкКарта --> Зарплатная[Зарплатная] </pre> |
| 24 | <pre> graph BT Сервис[Сервис] --> Продажи[Продажи] Сервис --> Лизинг[Лизинг] Сервис --> Кредит[Кредит] </pre> |
| 25 | <pre> graph BT Мебель[Мебель] --> Стол[Стол] Мебель --> МягкМебель[МягкМебель] Мебель --> Шкаф[Шкаф] </pre> |

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

- 6.1 Получить допуск к работе;
- 6.2 Выполнить задания.
- 6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

- 7.1 Наименование, цель занятия;
- 7.2 Выполненные задания (листинг программ);
- 7.3 Ответы на контрольные вопросы;
- 7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

- 8.1 Что такое наследование в объектно-ориентированном программировании?
- 8.2 Как записывается наследование на языке C#?
- 8.3 Как можно переопределить метод в C#? Зачем это может понадобиться?
- 8.4 Доступны ли члены класса-предка с модификатором private в классе-наследнике?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Наследование — это возможность использования существующих классов для создания новых классов. Новые классы могут создаваться на основе существующих классов, наследуя их функциональность, т. е. То, что умеют делать старые классы. При этом старые классы называют родителями или базовыми классами, а новые — потомками или дочерними классами, или производными классами. Новые классы фактически расширяют функциональность старых, потому что умеют делать то, что умеют старые, но и еще кое-что, чего нет в старых.

При наследовании в дочернем классе становятся доступными все члены родительского класса с атрибутом public или protected.

Наследование реализации (implementation inheritance) означает, что тип происходит от базового типа, получая от него все поля-члены и функции-члены.

Синтаксис:

```
class ПроизводныйКласс: БазовыйКласс
{
    // Данные-члены и функции-члены
}
```

При наследовании реализации производный класс наследует реализацию каждой функции базового типа, если только в его определении не указано, что реализация функции должна быть переопределена.

Определим следующую иерархию классов и продемонстрируем, как наследуется реализация и как переопределяются свойств и методов.



Создадим код, описывающий данную иерархию. Определим базовый класс:

```
class Человек
{
    public Человек(string Ф, string И, string О, int Возр)
    {
        Фамилия = Ф; Имя = И; Отчество = О; Возраст = Возр;
    }

    public Человек()
    {
        Фамилия = "нет данных"; Имя = ""; Отчество = "";
        Возраст = 18;
    }

    public string Фамилия, Имя, Отчество;
    private DateTime ДатаРождения;

    public virtual string ФИО
    {
        get { return Фамилия + " " + Имя + " " + Отчество; }
    }

    public int Возраст
    {
        get { return DateTime.Now.Year - ДатаРождения.Year; }
        set
        {
            int ГодРождения = DateTime.Now.Year - value;
            ДатаРождения = Convert.ToDateTime(ГодРождения.ToString()+".01.01");
        }
    }
}
```


Обратите внимание на наличие двух конструкторов, механизм хранения возраста человека и ключевое слово `virtual` у свойства «ФИО». Ключевое слово `virtual` указывает, что данное свойство будет переопределено в производном классе.

Определим производный класс «Учитель»:

```
class Учитель: Человек
{
    public Учитель()
        :base()
    {
        УченоеЗвание = УченыеЗвания.Без_Звания;
        УченаяСтепень = УченыеСтепени.Без_Степени;
    }

    public Учитель(string Ф, string И, string О, int Возр, УченыеЗвания УЗ, УченыеСтепени УС)
        :base(Ф, И, О, Возр)
    {
        УченоеЗвание = УЗ;
        УченаяСтепень = УС;
    }

    public УченыеЗвания УченоеЗвание;
    public УченыеСтепени УченаяСтепень;

    public override string ФИО
    {
        get
        {
            return УченаяСтепень.ToString() + ", " + УченоеЗвание.ToString() + ", " + base.ФИО;
        }
    }
}
```

Следует обратить внимание на использование ключевого слова `override` у свойства «ФИО», которое указывает на то, что данное свойство имеет новую реализацию, отличающуюся от реализации базового класса. Определим второй производный класс «Студент»:

```
class Студент: Человек
{
    public Студент(string Ф, string И, string О, int Возр, Специальности Спец)
        :base(Ф, И, О, Возр)
    {
        Специальность = Спец;
    }

    public Специальности Специальность;

    public override string ФИО
    {
        get
        {
            return base.ФИО + ", " + Специальность.ToString();
        }
    }
}
```

В обоих производных классах следует обратить внимание на реализацию конструкторов производных классов, использование ключевого слова `base` в коде свойства «ФИО» и при объявлении конструкторов.

Также интерес представляют типы данных, используемые для членовданных: «Специальности», «УченыеЗвания», «УченыеСтепени». Вот определения данных типов-перечислений:

НАИМЕНОВАНИЕ: Модульные приложения, командная строка C#

1. ЦЕЛЬ: Закрепление понятия командная строка, передачи параметров через командную строку, развитие навыков реализации программ на языке C# для обработки параметров командной строки программы

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Программирование на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

5.1 Напишите программу, в которой обеспечивается вызов справки, задание значения переменной через параметр в командной строке и выполнение вычисления факториала числа, равного номеру варианта.

5.2 Напишите программу, в которой обеспечивается вызов справки, задание значения переменной через параметр в командной строке и выполнение вычисления:

А)

$$s = \left| x^{y/x} - \sqrt{\frac{y}{x}} \right|;$$

$$w = (y - x) \frac{y - \frac{z}{y-x}}{1 + (y-x)^2}$$

| |
|---------------------------------------|
| $x = 1.82$ $y = 18$ $z = -3.29$ |
|---------------------------------------|

Б)

$$s = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!};$$

$$w = x (\sin(x) + \cos(y))$$

| |
|--------------------------|
| $x = 0.33$ $y = 0.02$ |
|--------------------------|

В)

$$y = e^{-bt} \sin(at + b) - \sqrt{|bt + a|};$$

$$s = b \sin(at^2 \cos(at)) - 1$$

| |
|---------------------------------------|
| $a = -0.5$ $b = 1.7$ $t = 0.44$ |
|---------------------------------------|

Г)

$$w = \sqrt{x^2 + b} - \frac{b^2 \sin^3(x + a)}{x};$$

$$y = \cos^2(x^3) - x / \sqrt{a^2 + b^2}$$

| |
|--|
| $a = -0.5$ $b = 15.5$ $x = -2.9$ |
|--|

Д)

$$s = x^3 \lg^2((x + b)^2) + \frac{a}{\sqrt{x + b}};$$

$$g = \frac{bx^2 - a}{e^{2x} - 1}$$

| |
|---------------------------------------|
| $a = 16.5$ $b = 3.4$ $x = 0.61$ |
|---------------------------------------|

Е)

$$r = \frac{x^2(x+1)}{b} - \sin^2(x + a);$$

$$s = \sqrt{\frac{xb}{a}} + \cos((x + b)^3)$$

| |
|--------------------------------------|
| $a = 0.7$ $b = 0.05$ $x = 0.5$ |
|--------------------------------------|

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

- 6.1 Получить допуск к работе;
- 6.2 Выполнить задания.
- 6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

- 7.1 Наименование, цель занятия;
- 7.2 Выполненные задания (листинг программ);
- 7.3 Ответы на контрольные вопросы;
- 7.4. Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

- 8.1 Что такое командная строка?
- 8.2 Как получить доступ к командной строке из программы на языке C#?
- 8.3 Как осуществляется индексация аргументов командной строки? Чему соответствует первый аргумент командной строки?
- 8.4 Зачем нужна справка в консольных приложениях?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Дополнительные аргументы у программ могут быть использованы для указания на входные данные, которые нужно обработать, задания параметров, определяющих ее функционирование и т.п. Если вы работали с каким-нибудь консольным приложением, то, наверняка, сталкивались с этим.

Чаще всего с аргументами командной строки приходится работать, если вы разрабатываете консольное приложение. В этом случае, доступ к переданным аргументам проще всего получить через переменную `args`, которая является аргументом метода `Main`, вызываемого по-умолчанию первым при старте приложения. Если, конечно, вы не изменили данное поведение. Вот пример кода, в рамках которого выводятся в консоль переданные приложению аргументы.

```
using System;
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Work with args directly:");
            foreach (string arg in args)
            {
                Console.WriteLine(arg);
            }
            Console.ReadKey();
        }
    }
}
```

Пример. В примере показана программа, которая выполняет одно из двух действий в зависимости от заданных параметров командной строки: вывод справки, либо вычисление факториала заданного в параметрах значения.

$$n! = 1 \cdot 2 \cdot \dots \cdot n = \prod_{i=1}^n i$$

Если запустить программу без параметров, то она не выполняет никаких действий. Для вызова справки используется параметр "LabParametrCOM /?". При использовании параметра "LabParametrCOM -f 10" происходит вычисление факториала указанного значения.

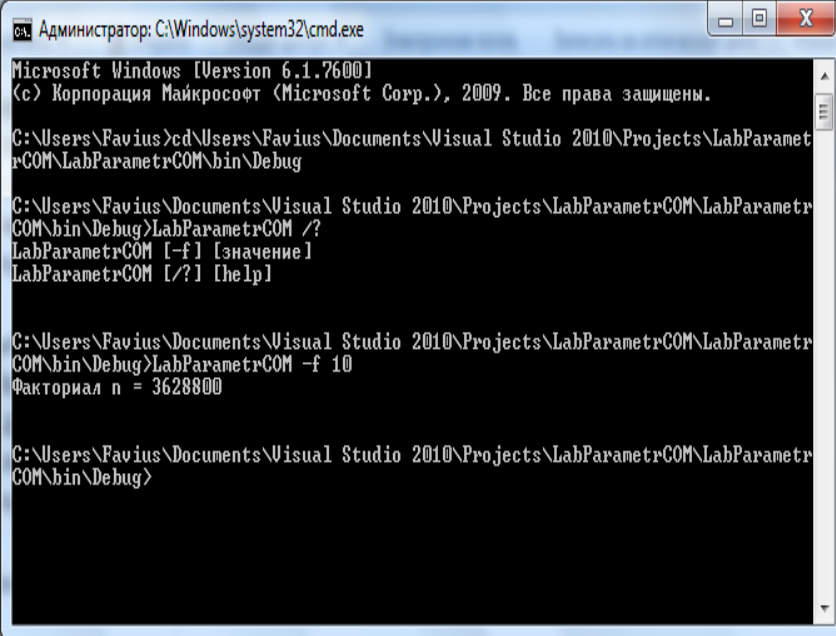
Листинг программы

```
using System;

namespace LabParametrCOM
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length != 0)
            {
                switch (args[0])
                {
                    case "-f":
                        pid(args[1]);
                        break;
                    case "/?":
                        help();
                        break;
                    default:
                        break;
                }
            }
        }
        static void help()
        {
            Console.WriteLine("LabParametrCOM [-f] [значение]");
            Console.WriteLine("LabParametrCOM [/?] [help]");
            Console.ReadLine();
        }
        static void pid(string b)
        {
            double n = 1;
            double d = 0;
            for (double i = 0; i < Convert.ToDouble(b); i++)
            {
                d++;
                n = n * d;
            }
            Console.WriteLine("Факториал n = {0}", n);

            Console.ReadLine();
        }
    }
}
```

Консоль перед закрытием.



```
Администратор: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\Favius>cd\Users\Favius\Documents\Visual Studio 2010\Projects\LabParametrCOM\LabParametrCOM\bin\Debug

C:\Users\Favius\Documents\Visual Studio 2010\Projects\LabParametrCOM\LabParametrCOM\bin\Debug>LabParametrCOM /?
LabParametrCOM [-f] [значение]
LabParametrCOM [/?] [help]

C:\Users\Favius\Documents\Visual Studio 2010\Projects\LabParametrCOM\LabParametrCOM\bin\Debug>LabParametrCOM -f 10
Факториал n = 3628800

C:\Users\Favius\Documents\Visual Studio 2010\Projects\LabParametrCOM\LabParametrCOM\bin\Debug>
```

ПРАКТИЧЕСКАЯ РАБОТА №38

НАИМЕНОВАНИЕ: Пользовательские интерфейсы на C#

1. ЦЕЛЬ: Закрепление понятий об интерфейсах в C#, развитие навыков разработки программ с использованием объектно-ориентированного подхода

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Создание объектно-ориентированных приложений на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

- 1) Открыть программу практической работы №34 Реализация наследования в C#;
- 2) Придумать минимум один интерфейс с минимум двумя элементами (свойства или методы);
- 3) Реализовать интерфейс в классе-наследнике из практической работы №34;
- 4) Показать в основной программе использование элементов интерфейса.

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

- 6.1 Получить допуск к работе;
- 6.2 Выполнить задания.
- 6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

- 7.1 Наименование, цель занятия;
- 7.2 Выполненные задания (листинг программ);
- 7.3 Ответы на контрольные вопросы;
- 7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

- 8.1 Что такое интерфейс в C#? Как его объявить?
- 8.2 Может ли один класс наследовать несколько классов? Несколько интерфейсов?
- 8.3 Для чего используется ключевое слово `base`?
- 8.4 Для чего используется ключевое слово `virtual`? Для чего используется ключевое слово `override`?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

В языке C# запрещено множественное наследование классов. Тем не менее, в C# существует концепция, позволяющая имитировать множественное наследование. Эта концепция интерфейсов. Интерфейс представляет собой набор объявлений свойств, индексаторов, методов и событий. Класс или структура могут реализовывать определенный интерфейс. В этом случае они берут на себя обязанность предоставить полную реализацию элементов интерфейса (хотя бы пустыми методами). Можно сказать так: интерфейс – это контракт, пункты которого суть свойства, индексаторы, методы и события. Если пользовательский тип реализует интерфейс, он берет на себя обязательство выполнить этот контракт.

Объявление интерфейса схоже с объявлением класса. Для объявления интерфейса используется ключевое слово `interface`. Интерфейс содержит только заголовки методов, свойств и событий:

```
interface IBird {  
    // Метод  
    void Fly();  
    // Свойство
```

```
double Speed { get; set; }  
}
```

Обратите внимание – в определении элементов интерфейса отсутствуют модификаторы уровня доступа. Считается, что все элементы интерфейса имеют public уровень доступа. Более точно, следующие модификаторы не могут использоваться при объявлении членов интерфейса: abstract, public, protected, internal, private, virtual, override, static. Для свойства, объявленного в интерфейсе, указываются только ключевые слова get и (или) set.

Чтобы показать, что класс реализовывает некий интерфейс, используется синтаксис <имя класса> : <имя реализовываемого интерфейса> при записи заголовка класса. Если класс является производным от некоторого базового класса, то имя базового класса указывается перед именем реализовываемого интерфейса: <имя класса> : <имя базового класса>, <имя интерфейса> .

В простейшем случае, чтобы указать, что некий член класса соответствует элементу интерфейса, у них должны совпадать имена:

```
class CFalcon : IBird {  
    private double FS;  
    public void DoSomething() {  
        Console.WriteLine("Falcon Flys");  
    }  
    public void Fly() {  
        Console.WriteLine("Falcon Flys");  
    }  
    public double Speed {  
        get { return FS; }  
        set { FS = value; }  
    }  
}
```

Для элементов класса, реализующих интерфейс, обязательным является использование модификатора доступа public.

Пример использования интерфейсов

```
public interface ICalculate  
{  
    void Plus(int pPlus);  
    void Minus(int pMinus);  
}  
  
public interface IVisual  
{  
    string Name { get; set; }  
    void DrawObject();  
}
```

В приведенном примере объявлены два интерфейса: ICalculate и IVisual. Интерфейс ICalculate включает два метода с одним параметром. Из названий видно, что один метод что-то увеличивает, второй – что-то уменьшает на величину параметра. Второй интерфейс IVisual содержит метод DrawObject без параметров, который призван нарисовать объект, а также свойство Name, которое доступно как для чтения, так и для записи.

Для использования объявленных интерфейсов создадим два класса Human (человек) и Car (автомобиль).


```

public class Human
{
    public Human(string pFIO, int pAge)
    {
        FIO = pFIO;
        Age = pAge;
    }

    private string FIO;
    private int Age;
}

public class Car
{
    public Car(string pManufacturer, string pModel, int pVelocity)
    {
        Manufacturer = pManufacturer;
        Model = pModel;
        Velocity = pVelocity;
    }

    private string Manufacturer;
    private string Model;
    private int Velocity;
}

```

Реализуем наследование интерфейса IVisual и ICalculate для класса Human

```

public class Human: ICalculate, IVisual
{
    public Human(string pFIO, int pAge)
    {
        FIO = pFIO;
        Age = pAge;
    }

    private string FIO;
    private int Age;

    public void Plus(int pPlus)
    {
        Age += pPlus;
    }

    public void Minus(int pMinus)
    {
        Age -= pMinus;
    }

    public string Name
    {
        get
        {
            return FIO + " : " + Age.ToString();
        }
        set
        {
            FIO = value;
        }
    }

    public void DrawObject()
    {
        Console.WriteLine
        (
            "      o      \n" +
            "    ----- \n" +
            "      |      \n" +
            "     /  \ \   \n" +
            "     /  \ \   \n" +
        );
        Console.WriteLine(Name);
    }
}

```

Реализуем интерфейсы ICalculate и IVisual для класса Car:

```

public class Car: IVisual, ICalculate
{
    public Car(string pManufacturer, string pModel, int pVelocity)
    {
        Manufacturer = pManufacturer;
        Model = pModel;
        Velocity = pVelocity;
    }

    private string Manufacturer;
    private string Model;
    private int Velocity;

    public void Plus(int pPlus)
    {
        Velocity += pPlus;
    }

    public void Minus(int pMinus)
    {
        Velocity += pMinus;
    }

    public string Name
    {
        get
        {
            return Manufacturer + " - " + Model +
                " : " + Velocity.ToString() + "km/h";
        }
        set
        {
            Model = value;
        }
    }

    public void DrawObject()
    {
        Console.WriteLine
        (
            "      ----- \n" +
            "  ____/          \ \____ \n" +
            " |                    | \n" +
            " |---(@)-----(@)--- \n"
        );
        Console.WriteLine(Name);
    }
}

```

Продemonстрируем использование типов данных созданием соответствующих объектов в функции main.

```
static void Main(string[] args)
{
    Console.BackgroundColor = ConsoleColor.White;
    Console.ForegroundColor = ConsoleColor.Blue;
    Console.Clear();

    Console.Title = "Лабораторная работа №11";

    Human h = new Human("Иванов Иван Иванович", 50);
    h.Plus(5);
    h.Minus(1);
    h.DrawObject();

    Console.WriteLine("\n\n\n");

    Car car = new Car("Hyundai", "ix35", 120);
    car.Plus(25);
    car.Minus(11);
    car.DrawObject();

    Console.ReadKey();
}
```

ПРАКТИЧЕСКАЯ РАБОТА №39

НАИМЕНОВАНИЕ: Разработка программ с использованием порождающих паттернов

1. ЦЕЛЬ: Закрепление понятий об порождающих паттернах в C# - фабричный метод, знакомство с реализацией фабричного метода

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Создание объектно-ориентированных приложений на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. Задания:

5.1 Набрать пример из приложения;

5.2 Зарисовать диаграмму по коду (пример диаграммы указан на рисунке);

5.3 Придумать не менее 3 ситуаций, когда использование паттерна фабричный метод будет полезно при проектировании приложения

6. Порядок проведения занятия:

6.1 Получить допуск к работе;

6.2 Выполнить задания.

6.3 Оформить отчет.

7. Содержание отчета:

7.1 Наименование, цель занятия;

7.2 Выполненные задания (листинг программ);

7.3 Ответы на контрольные вопросы;

7.4 Вывод о проделанной работе.

8. Контрольные вопросы для зачета:

8.1 Что такое паттерны в программировании? Зачем они нужны?

8.2 Что такое порождающие паттерны? Примеры порождающих паттернов

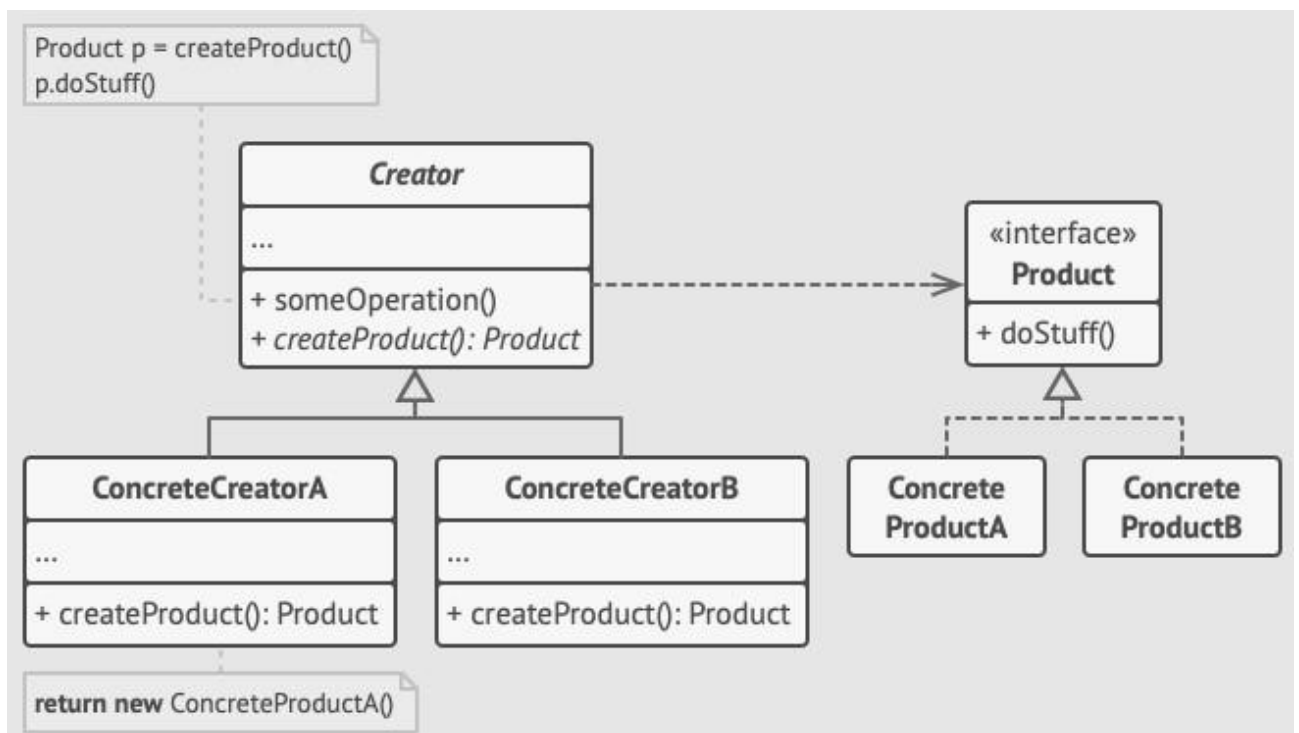
8.3 Что такое фабричный метод? Когда он используется, в чем его недостатки и достоинства

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Пример диаграммы, отражающей использование паттерна



Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.

Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.

Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов. Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт. Несмотря на название, важно понимать, что создание продуктов не является единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.

Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты. Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

Пример кода

```
using System;
```

```
namespace prkt
{
    // Класс Создатель объявляет фабричный метод, который должен возвращать
    // объект класса Продукт. Подклассы Создателя обычно предоставляют
    // реализацию этого метода.

    abstract class Creator
    {
        // Обратите внимание, что Создатель может также обеспечить реализацию
        // фабричного метода по умолчанию.

        public abstract IProduct FactoryMethod();
    }
}
```

```

// Также заметьте, что, несмотря на название, основная обязанность
// Создателя не заключается в создании продуктов. Обычно он содержит
// некоторую базовую бизнес-логику, которая основана на объектах
// Продуктов, возвращаемых фабричным методом. Подклассы могут косвенно
// изменять эту бизнес-логику, переопределяя фабричный метод и возвращая
// из него другой тип продукта.

public string SomeOperation()
{
    // Вызываем фабричный метод, чтобы получить объект-продукт.
    var product = FactoryMethod();

    // Далее, работаем с этим продуктом.
    var result = "Creator: The same creator's code has just worked with "
        + product.Operation();

    return result;
}
}

// Конкретные Создатели переопределяют фабричный метод для того, чтобы
// изменить тип результирующего продукта.

class ConcreteCreator1 : Creator
{
    // Обратите внимание, что сигнатура метода по-прежнему использует тип
    // абстрактного продукта, хотя фактически из метода возвращается
    // конкретный продукт. Таким образом, Создатель может оставаться
    // независимым от конкретных классов продуктов.

    public override IProduct FactoryMethod()
    {
        return new ConcreteProduct1();
    }
}

class ConcreteCreator2 : Creator
{
    public override IProduct FactoryMethod()
    {
        return new ConcreteProduct2();
    }
}

```

```

    }

}

// Интерфейс Продукта объявляет операции, которые должны выполнять все
// конкретные продукты.

public interface IProduct
{
    string Operation();
}

// Конкретные Продукты предоставляют различные реализации интерфейса
// Продукта.

class ConcreteProduct1 : IProduct
{
    public string Operation()
    {
        return "{Result of ConcreteProduct1}";
    }
}

class ConcreteProduct2 : IProduct
{
    public string Operation()
    {
        return "{Result of ConcreteProduct2}";
    }
}

class Client
{
    public void Main()
    {
        Console.WriteLine("App: Launched with the ConcreteCreator1.");
        ClientCode(new ConcreteCreator1());

        Console.WriteLine("");

        Console.WriteLine("App: Launched with the ConcreteCreator2.");
    }
}

```



```

        ClientCode(new ConcreteCreator2());
    }

    // Клиентский код работает с экземпляром конкретного создателя, хотя и
    // через его базовый интерфейс. Пока клиент продолжает работать с
    // создателем через базовый интерфейс, вы можете передать ему любой
    // подкласс создателя.
    public void ClientCode(Creator creator)
    {
        // ...
        Console.WriteLine("Client: I'm not aware of the creator's class," +
            "but it still works.\n" + creator.SomeOperation());
        // ...
    }
}

class Program
{
    static void Main(string[] args)
    {
        new Client().Main();
    }
}
}

```

ПРАКТИЧЕСКАЯ РАБОТА №40

НАИМЕНОВАНИЕ: Разработка программ с использованием структурных паттернов

1. ЦЕЛЬ: Закрепление понятий об структурных паттернах в C# - адаптер, знакомство с реализацией адаптера

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Создание объектно-ориентированных приложений на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

5.1 Набрать пример из приложения;

5.2 Зарисовать диаграмму по коду (пример диаграммы указан на рисунке);

5.3 Придумать не менее 3 ситуаций, когда использование паттерна адаптер будет полезно при проектировании приложения

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

6.1 Получить допуск к работе;

6.2 Выполнить задания.

6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

7.1 Наименование, цель занятия;

7.2 Выполненные задания (листинг программ);

7.3 Ответы на контрольные вопросы;

7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

8.1 Что такое структурные шаблоны проектирования?

8.2 Что такое адаптер? Из чего, как правило, он состоит?

8.3 Чем адаптер объектов отличается от адаптера класса?

8.4 Перечислить ещё примеры структурных шаблонов

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Структурные шаблоны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример — использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот шаблон, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

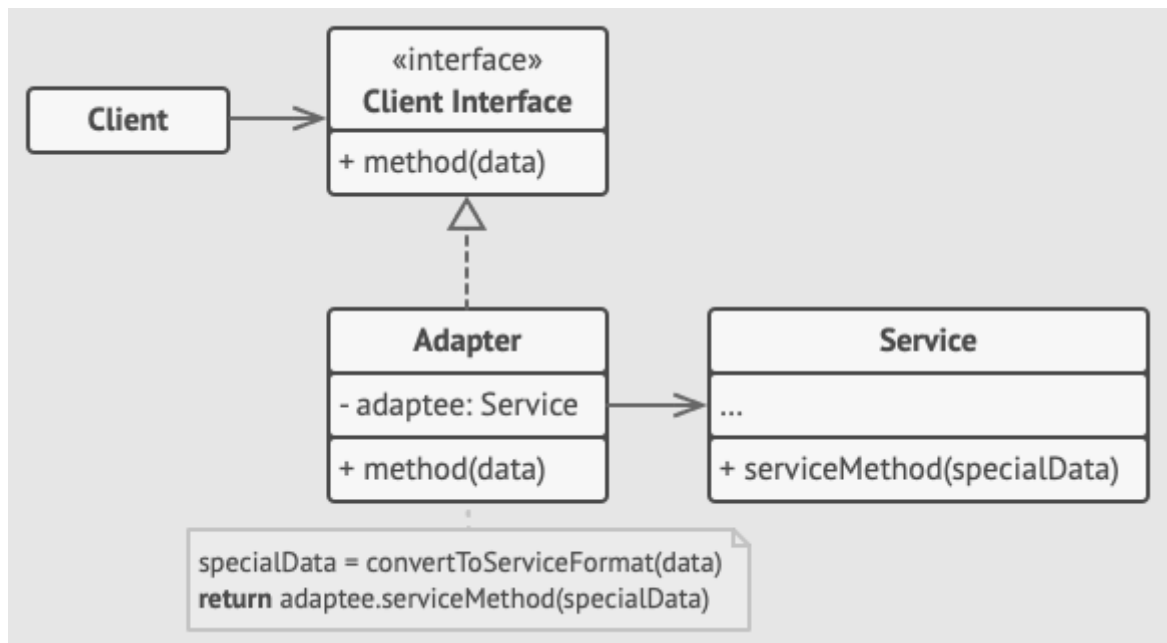
Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Примеры:

Адаптер объектов

Эта реализация использует агрегацию: объект адаптера «оборачивает», то есть содержит ссылку на служебный объект. Такой подход работает во всех языках программирования.



Клиент — это класс, который содержит существующую бизнес-логику программы.

Клиентский интерфейс описывает протокол, через который клиент может работать с другими классами.

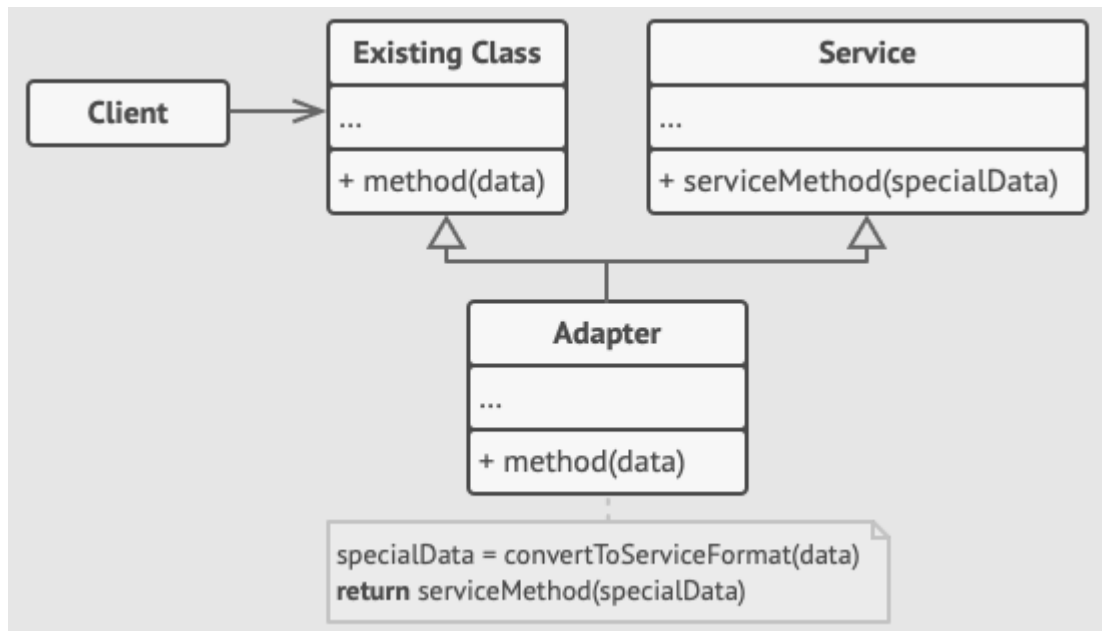
Сервис — это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.

Адаптер — это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.

Работая с адаптером через интерфейс, клиент не привязывается к конкретному классу адаптера. Благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки.

Адаптер классов

Эта реализация базируется на наследовании: адаптер наследует оба интерфейса одновременно. Такой подход возможен только в языках, поддерживающих множественное наследование, например, C++.



Адаптер классов не нуждается во вложенном объекте, так как он может одновременно наследовать и часть существующего класса, и часть сервиса.

Пример структуры паттерна на C#

```
using System;
```

```
namespace Prakt38
```

```
{
```

```
    // Целевой класс объявляет интерфейс, с которым может работать клиентский
```

```
    // код.
```

```
    public interface ITarget
```

```
    {
```

```
        string GetRequest();
```

```
    }
```

```
    // Адаптируемый класс содержит некоторое полезное поведение, но его
```

```
    // интерфейс несовместим с существующим клиентским кодом. Адаптируемый
```

```
    // класс нуждается в некоторой доработке, прежде чем клиентский код сможет
```

```
    // его использовать.
```

```
    class Adaptee
```

```
    {
```

```
        public string GetSpecificRequest()
```

```
        {
```

```
            return "Specific request.";
```

```

    }

}

// Адаптер делает интерфейс Адаптируемого класса совместимым с целевым
// интерфейсом.
class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee)
    {
        this._adaptee = adaptee;
    }

    public string GetRequest()
    {
        return $"This is '{this._adaptee.GetSpecificRequest()}';"
    }
}

class Program
{
    static void Main(string[] args)
    {
        Adaptee adaptee = new Adaptee();
        ITarget target = new Adapter(adaptee);

        Console.WriteLine("Adaptee interface is incompatible with the client.");
        Console.WriteLine("But with adapter client can call it's method.");

        Console.WriteLine(target.GetRequest());
    }
}
}

```

ПРАКТИЧЕСКАЯ РАБОТА №41

НАИМЕНОВАНИЕ: Разработка программ с использованием паттернов поведения

1. ЦЕЛЬ: Закрепление понятий об паттернах поведения в C# - снимок, знакомство с реализацией снимка

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Создание объектно-ориентированных приложений на языке C#».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

5.1 Набрать пример из приложения;

5.2 Зарисовать диаграмму по коду (пример диаграммы указан на рисунке);

5.3 Придумать не менее 3 ситуаций, когда использование паттерна снимок будет полезно при проектировании приложения

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

6.1 Получить допуск к работе;

6.2 Выполнить задания.

6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

7.1 Наименование, цель занятия;

7.2 Выполненные задания (листинг программ);

7.3 Ответы на контрольные вопросы;

7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

8.1 Что такое поведенческие шаблоны проектирования?

8.2 Зачем нужен шаблон снимок? Как он ещё называется?

8.3 Какие есть варианты реализации шаблона снимок?

8.4 Привести примеры ещё поведенческих шаблонов проектирования

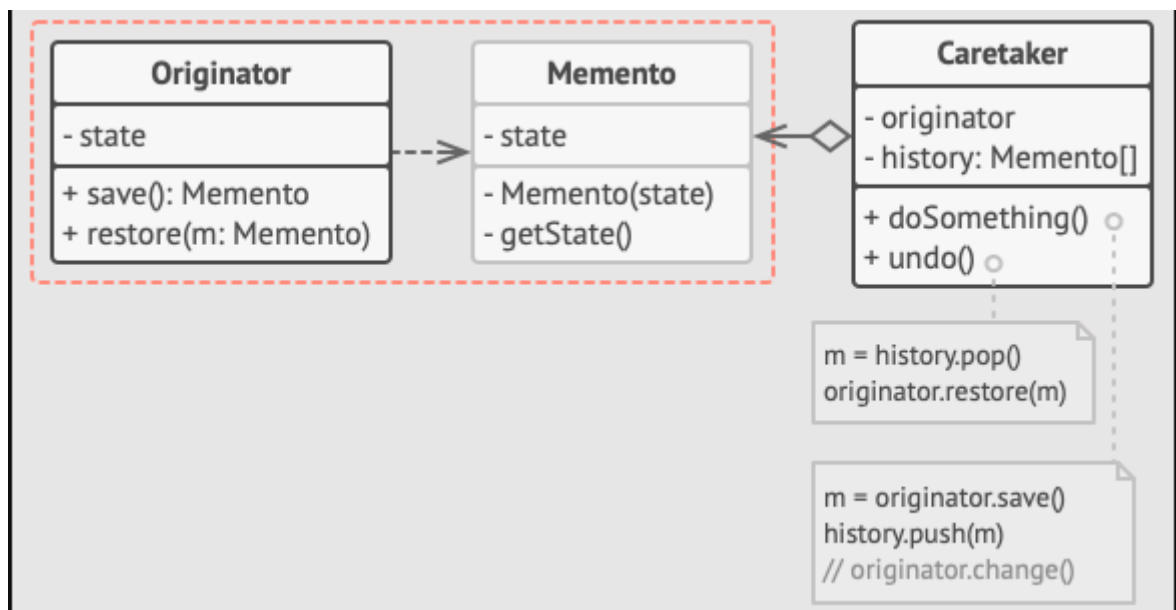
ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Хранитель (англ. Memento) — поведенческий шаблон проектирования, позволяющий, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта так, чтобы позднее восстановить его в это состояние.

Классическая реализация паттерна полагается на механизм вложенных классов, который доступен лишь в некоторых языках программирования (C++, C#, Java).

1. Классическая реализация на вложенных классах



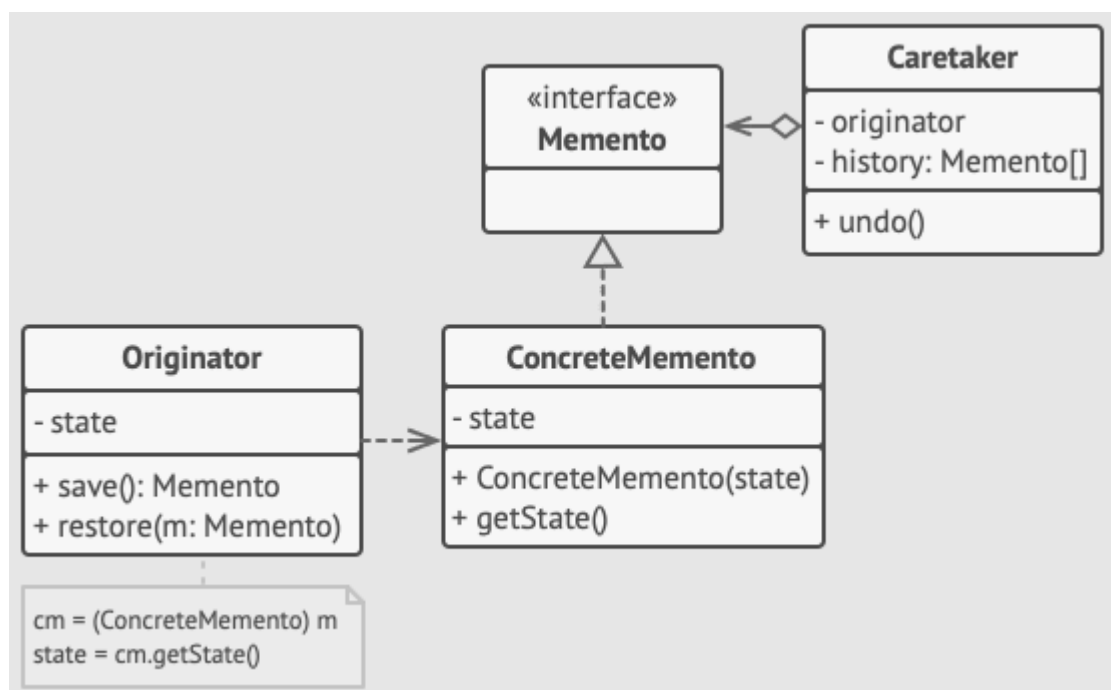
Создатель может производить снимки своего состояния, а также воспроизводить прошлое состояние, если подать в него готовый снимок.

Снимок — это простой объект данных, содержащий состояние создателя. Надёжнее всего сделать объекты снимков неизменяемыми, передавая в них состояние только через конструктор.

Опекун должен знать, когда делать снимок создателя и когда его нужно восстанавливать. Опекун может хранить историю прошлых состояний создателя в виде стека из снимков. Когда понадобится отменить выполненную операцию, он возьмёт «верхний» снимок из стека и передаст его создателю для восстановления.

В данной реализации снимок — это внутренний класс по отношению к классу создателя. Именно поэтому он имеет полный доступ к полям и методам создателя, даже приватным. С другой стороны, опекун не имеет доступа ни к состоянию, ни к методам снимков и может всего лишь хранить ссылки на эти объекты.

2. Реализация с пустым промежуточным интерфейсом



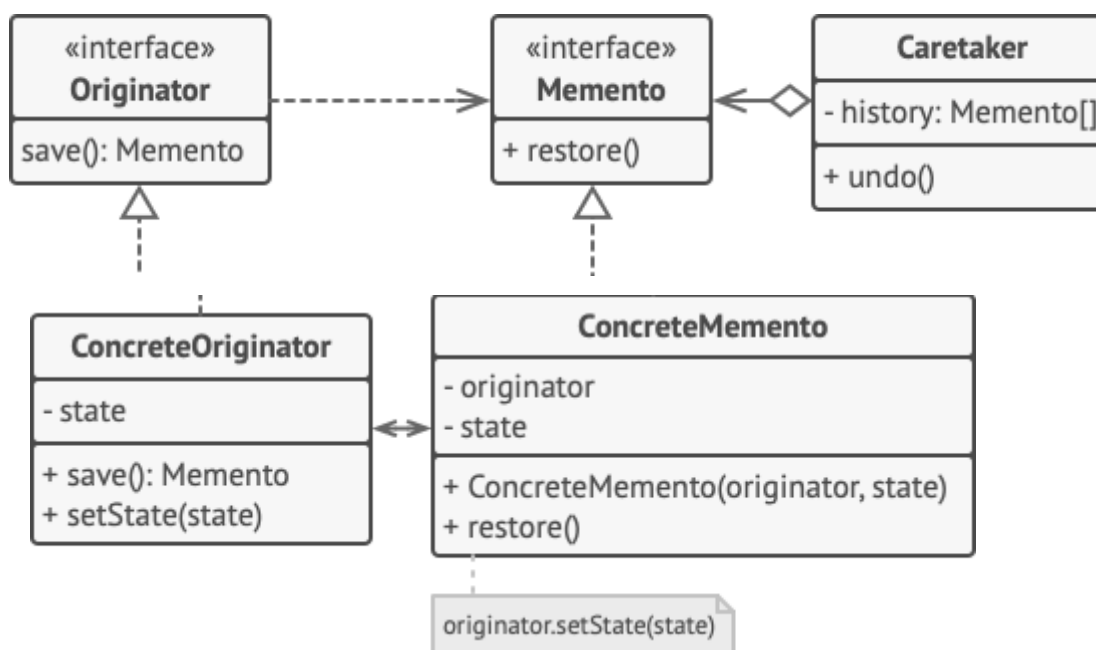
Подходит для языков, не имеющих механизма вложенных классов (например, PHP).

В этой реализации создатель работает напрямую с конкретным классом снимка, а опекун — только с его ограниченным интерфейсом.

Благодаря этому достигается тот же эффект, что и в классической реализации. Создатель имеет полный доступ к снимку, а опекун — нет.

3. Снимки с повышенной защитой

Когда нужно полностью исключить возможность доступа к состоянию создателей и снимков.



Эта реализация разрешает иметь несколько видов создателей и снимков. Каждому классу создателей соответствует свой класс снимков. Ни создатели, ни снимки не позволяют другим объектам прочесть своё состояние.

Здесь опекун ещё более жёстко ограничен в доступе к состоянию создателей и снимков. Но, с другой стороны, опекун становится независим от создателей, поскольку метод восстановления теперь находится в самих снимках.

Снимки теперь связаны с теми создателями, из которых они сделаны. Они по-прежнему получают состояние через конструктор. Благодаря близкой связи между классами, снимки знают, как восстановить состояние своих создателей.

Пример

```
using System;
using System.Collections.Generic;

using System.Linq;

using System.Threading;
```


namespace prakt

```
{

    // Создатель содержит некоторое важное состояние, которое может со временем
    // меняться. Он также объявляет метод сохранения состояния внутри снимка и
    // метод восстановления состояния из него.

    class Originator
    {
        // Для удобства состояние создателя хранится внутри одной переменной.

        private string _state;

        public Originator(string state)
        {
            this._state = state;

            Console.WriteLine("Originator: My initial state is: " + state);
        }

        // Бизнес-логика Создателя может повлиять на его внутреннее состояние.
        // Поэтому клиент должен выполнить резервное копирование состояния с
        // помощью метода save перед запуском методов бизнес-логики.

        public void DoSomething()
        {
            Console.WriteLine("Originator: I'm doing something important.");
            this._state = this.GenerateRandomString(30);
            Console.WriteLine($"Originator: and my state has changed to: {_state}");
        }

        private string GenerateRandomString(int length = 10)
        {
            string allowedSymbols = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
            string result = string.Empty;

            while (length > 0)
            {
                result += allowedSymbols[new Random().Next(0, allowedSymbols.Length)];

                Thread.Sleep(12);

                length--;
            }
        }
    }
}
```

```

        return result;
    }

    // Сохраняет текущее состояние внутри снимка.
    public IMemento Save()
    {
        return new ConcreteMemento(this._state);
    }

    // Восстанавливает состояние Создателя из объекта снимка.
    public void Restore(IMemento memento)
    {
        if (!(memento is ConcreteMemento))
        {
            throw new Exception("Unknown memento class " + memento.ToString());
        }

        this._state = memento.GetState();
        Console.WriteLine($"Originator: My state has changed to: {_state}");
    }
}

// Интерфейс Снимка предоставляет способ извлечения метаданных снимка, таких
// как дата создания или название. Однако он не раскрывает состояние
// Создателя.
public interface IMemento
{
    string GetName();

    string GetState();

    DateTime GetDate();
}

// Конкретный снимок содержит инфраструктуру для хранения состояния
// Создателя.
class ConcreteMemento : IMemento
{
    private string _state;

```

```

private DateTime _date;

public ConcreteMemento(string state)
{
    this._state = state;
    this._date = DateTime.Now;
}

// Создатель использует этот метод, когда восстанавливает своё
// состояние.
public string GetState()
{
    return this._state;
}

// Остальные методы используются Опекуном для отображения метаданных.
public string GetName()
{
    return $"{this._date} / ({this._state.Substring(0, 9)})...";
}

public DateTime GetDate()
{
    return this._date;
}
}

// Опекун не зависит от класса Конкретного Снимка. Таким образом, он не
// имеет доступа к состоянию создателя, хранящемуся внутри снимка. Он
// работает со всеми снимками через базовый интерфейс Снимка.
class Caretaker
{
    private List<IMemento> _mementos = new List<IMemento>();

    private Originator _originator = null;

    public Caretaker(Originator originator)
    {
        this._originator = originator;
    }
}

```

```

public void Backup()
{
    Console.WriteLine("\nCaretaker: Saving Originator's state...");
    this._mementos.Add(this._originator.Save());
}

public void Undo()
{
    if (this._mementos.Count == 0)
    {
        return;
    }

    var memento = this._mementos.Last();
    this._mementos.Remove(memento);

    Console.WriteLine("Caretaker: Restoring state to: " + memento.GetName());

    try
    {
        this._originator.Restore(memento);
    }
    catch (Exception)
    {
        this.Undo();
    }
}

public void ShowHistory()
{
    Console.WriteLine("Caretaker: Here's the list of mementos:");

    foreach (var memento in this._mementos)
    {
        Console.WriteLine(memento.GetName());
    }
}

```

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код.

        Originator originator = new Originator("Super-duper-super-puper-super.");
        Caretaker caretaker = new Caretaker(originator);

        caretaker.Backup();
        originator.DoSomething();

        caretaker.Backup();
        originator.DoSomething();

        caretaker.Backup();
        originator.DoSomething();

        Console.WriteLine();
        caretaker.ShowHistory();

        Console.WriteLine("\nClient: Now, let's rollback!\n");
        caretaker.Undo();

        Console.WriteLine("\nClient: Once more!\n");
        caretaker.Undo();

        Console.WriteLine();
    }
}
```


ПРАКТИЧЕСКАЯ РАБОТА №42

НАИМЕНОВАНИЕ: Создание приложения на основе технологии Windows Forms

1. ЦЕЛЬ: Познакомиться с технологией Windows Forms, получить представление о способах создания элементов графического интерфейса приложения на языке C#

2. ПОДГОТОВКА К ЗАНЯТИЮ: Повторить теоретический материал по теме «Визуальное событийно-управляемое программирование».

3. ЛИТЕРАТУРА:

3.1 Сергеев Р.А. Основы алгоритмизации и программирования. Учебное пособие, - Самара: КС ПГУТИ, 2021 г.

4. ПЕРЕЧЕНЬ ОБОРУДОВАНИЯ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ:

4.1. Персональный IBM PC.

5. ЗАДАНИЯ:

5.1 Установить среду разработки Visual Studio Community;

5.2 Создать проект Windows Forms App;

5.3 Добавить в основную форму не менее трех новых элементов с помощью графического редактора;

5.4 Добавить кнопку в форму;

5.5 Установить обработчик нажатия кнопки, выводящий сообщение с фамилией студента;

5.6 Добавить ещё одну форму в проект;

5.7 Осуществить взаимодействие между формами.

6. ПОРЯДОК ПРОВЕДЕНИЯ ЗАНЯТИЯ:

6.1 Получить допуск к работе;

6.2 Выполнить задания.

6.3 Оформить отчет.

7. СОДЕРЖАНИЕ ОТЧЕТА:

7.1 Наименование, цель занятия;

7.2 Выполненные задания (листинг программ);

7.3 Ответы на контрольные вопросы;

7.4 Вывод о проделанной работе.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ ЗАЧЕТА:

8.1 Что такое Visual Studio? В каких редакторах вы ещё работали на занятиях? В чем преимущества Visual Studio?

8.2 Что такое Windows Forms?

8.3 Объяснить, как установить обработчик нажатия кнопки, в чем смысл его параметров.

8.4 Как добавить ещё одну форму в проект? Как осуществляется взаимодействие между формами?

ПРИЛОЖЕНИЕ:

Краткие теоретические сведения:

Для создания графических интерфейсов с помощью платформы .NET применяются разные технологии - Window Forms, WPF, UWP. Однако наиболее простой и удобной платформой до сих пор остается Window Forms или сокращенно WinForms.

Для создания графических приложений на C# будем использовать бесплатную и полнофункциональную среду разработки - Visual Studio Community. Версию можно использовать Visual Studio Community 2022.

Чтобы добавить в Visual Studio поддержку проектов для Windows Forms и C# и .NET 6, в программе установки среди рабочих нагрузок нужно выбрать только пункт Разработка

классических приложений .NET. Можно выбрать и больше опций или вообще все опции, однако стоит учитывать свободный размер на жестком диске - чем больше опций будет выбрано, соответственно тем больше места на диске будет занято.

После установки среды и всех ее компонентов, запустим Visual Studio и создадим проект графического приложения. На стартовом экране выберем Create a new project (Создать новый проект) (см. рис. 1)

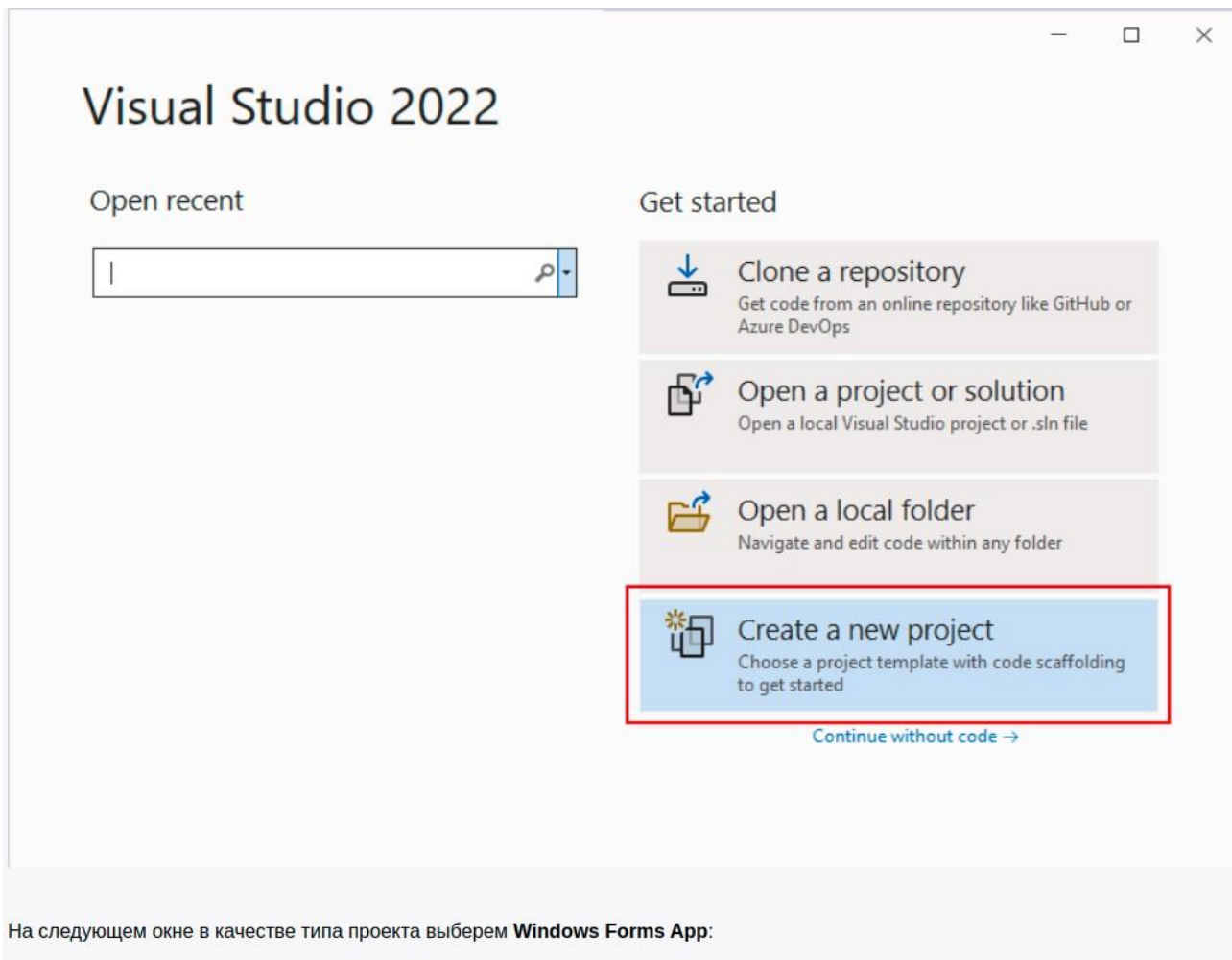


Рисунок 1. Создание нового проекта

На следующем окне в качестве типа проекта выберем Windows Forms App. Стоит отметить, что среди шаблонов можно увидеть еще тип Windows Forms App (.NET Framework) - его НЕ надо выбирать, необходим именно тип Windows Forms App.

Далее на следующем этапе нам будет предложено указать имя проекта и каталог, где будет располагаться проект.

В поле Project Name дадим проекту какое-либо название.

На следующем окне Visual Studio предложит нам выбрать версию .NET, которая будет использоваться для проекта. По умолчанию здесь выбрана последняя на данный момент версия - .NET 6.0. Оставим и нажмем на кнопку Create (Создать) для создания проекта.

После этого Visual Studio откроет наш проект с созданными по умолчанию файлами (рис. 2):

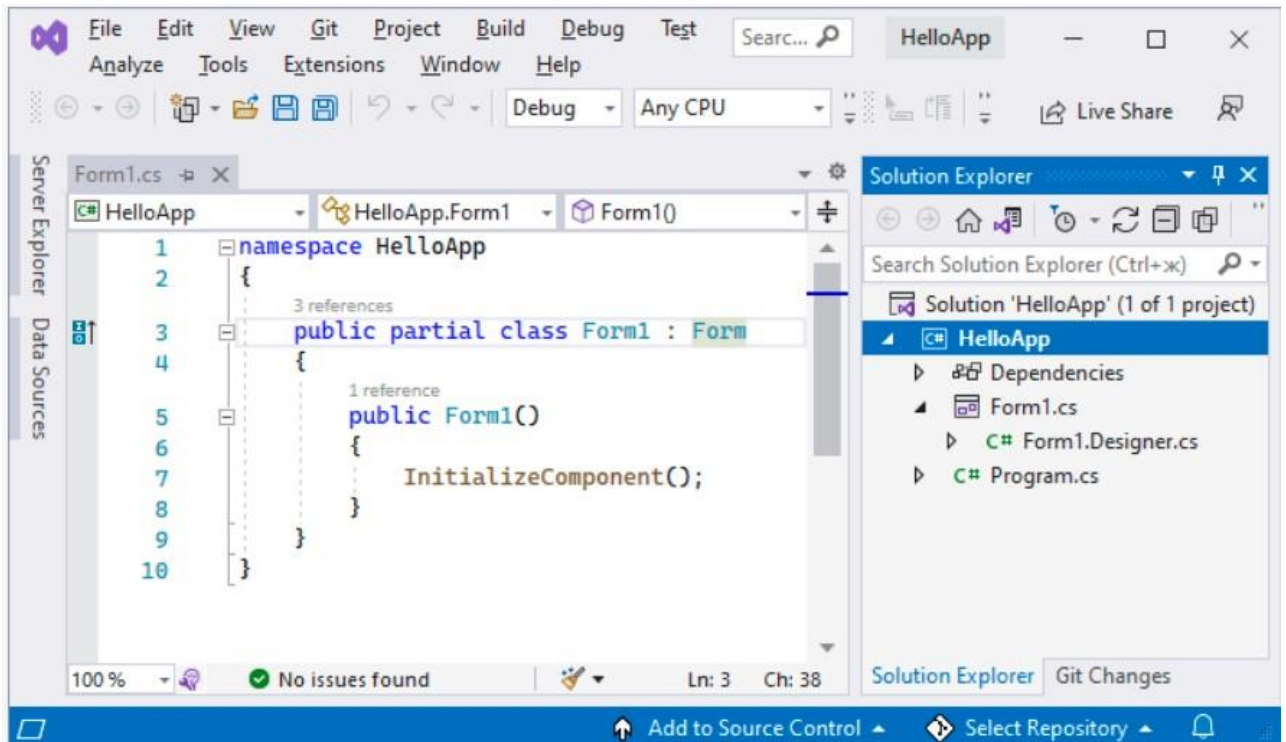


Рисунок 2. Структура нового проекта

В данном случае у нас сгенерированная по умолчанию структура:

8 Dependencies - это узел содержит сборки dll, которые добавлены в проект по умолчанию. Эти сборки как раз содержат классы библиотеки .NET, которые будет использовать C#

9 Далее идет файл единственной в проекте формы - Form1.cs, который по умолчанию открыт в центральном окне;

Класс формы – Form1 представляет графическую форму - фактически то окно, которое мы увидим на экране при запуске проекта.

Этот класс определяется как частичный (с модификатором partial) и наследуется от встроенного класса Form, который содержит базовую функциональность форм.

В самом классе Form1 определен по умолчанию только конструктор, где вызывается метод InitializeComponent(), который выполняет инициализацию компонентов формы из файла дизайнера.

Рядом с этим элементом можно заметить другой файл формы - Form1.Designer.cs. Это файл дизайнера - он содержит определение компонентов формы, добавленных на форму в графическом дизайнера и именно его код по сути передается выше через вызов InitializeComponent

10 Program.cs определяет точку входа в приложение;

В нем используется метод ApplicationConfiguration.Initialize() который устанавливает некоторую базовую конфигурацию приложения. Затем вызывается метод Application.Run(new Form1());

в который передается объект отображаемой по умолчанию на экране формы. То есть, когда мы запустим приложение, сработает метод Main, в котором будет вызван метод Application.Run(new Form1()), благодаря чему мы увидим форму Form1 на экране.

Чтобы запустить приложение в режиме отладки, нажмем на клавишу F5 или на зеленую стрелочку на панели Visual Studio.

Одним из преимуществ разработки в Visual Studio приложений Windows Forms является наличие графического редактора, который позволяет в графическом виде представить создаваемую форму и в принципе упрощает работу с графическими компонентами.

Для открытия формы в режиме графического дизайнера нажмем на в структуре проекта на файл Form1.cs либо левой кнопкой мыши двойным кликом, либо правой кнопкой мыши и в появившемся контекстном меню выберем View Designer (также можно использовать комбинацию клавиш Shift+F7)

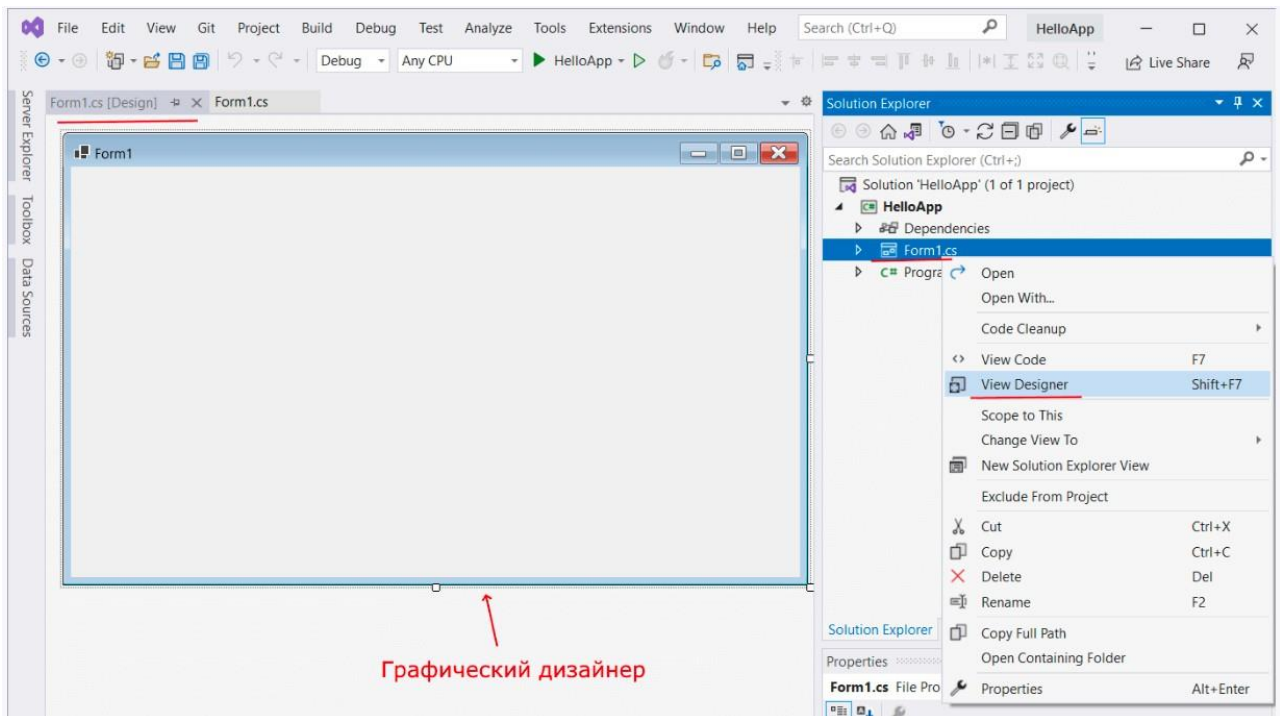


Рисунок 3. Окно графического дизайнера

При выборе формы в окне дизайнера внизу справа под структурой проекта мы сможем найти окно Properties(Свойства). Здесь мы можем поменять свойства формы, которые доступны в окне свойств

Но Visual Studio имеет еще одну связанную функциональность. Она обладает панелью графических инструментов. И мы можем, вместо создания элементов управления в коде C#, просто переносить их на форму с панели инструментов с помощью мыши. Так, перенесем на форму какой-нибудь элемент управления, например, кнопку. Для этого найдем в левой части Visual Studio вкладку Toolbox (Панель инструментов). Нажмем на эту вкладку, и у нас откроется панель с элементами, откуда мы можем с помощью мыши перенести на форму любой элемент:

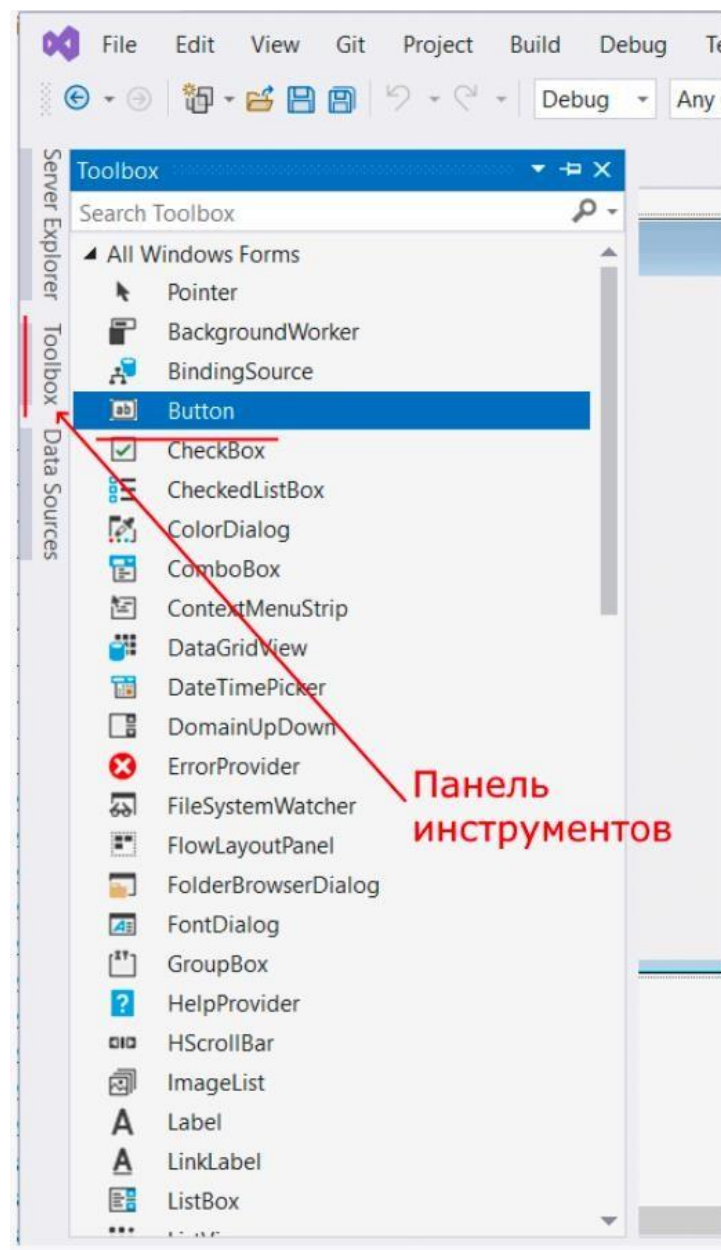


Рисунок 4. Панель инструментов для добавления

Причем при выборе кнопки она открывается в окне свойств и, как и для всей формы, для кнопки в окне свойств мы можем изменить значения различных свойств. Кроме того, в файле Form1 произошли изменения – добавился код для кнопки.

Добавим простейший код на языке C#, который бы выводил сообщение по нажатию кнопки. Для этого перейдем в файл кода Form1.cs, который связан с этой формой. По умолчанию после создания проекта он имеет код типа следующего: namespace HelloApp

```
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Изменим этот код следующим образом:

```
namespace HelloApp
{
```

```

public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

        button1.Click += button1_Click;
    }
    private void button1_Click(object? sender, EventArgs e)
    {
        MessageBox.Show("Привет");
    }
}

```

Кнопка обладает событием Click, которое генерируется при нажатии. В данном случае в конструкторе формы мы подвязываем к кнопке button1 в качестве обработчика события нажатия метод button1_Click, в котором с помощью метода MessageBox.Show выводит сообщение. Текст сообщения передается в метод в качестве параметра.

Добавление форм. Взаимодействие между формами

Чтобы добавить еще одну форму в проект, нажмем на имя проекта в окне Solution Explorer (Обозреватель решений) правой кнопкой мыши и выберем Add(Добавить)->Windows Form... Итак, у нас в проект была добавлена вторая форма. Теперь попробуем осуществить взаимодействие между двумя формами. Допустим, первая форма по нажатию на кнопку будет вызывать вторую форму. Во-первых, добавим на первую форму Form1 кнопку и двойным щелчком по кнопке перейдем в файл кода. Итак, мы попадем в обработчик события нажатия кнопки, который создается по умолчанию после двойного щелчка по кнопке:

```

private void button1_Click(object sender, EventArgs e)
{

}

```

Теперь добавим в него код вызова второй формы. У нас вторая форма называется Form2, поэтому сначала мы создаем объект данного класса, а потом для его отображения на экране вызываем метод Show:

```

private void button1_Click(object sender, EventArgs e)
{
    Form2 newForm = new Form2();
    newForm.Show();
}

```

Теперь сделаем наоборот - чтобы вторая форма воздействовала на первую. Пока вторая форма не знает о существовании первой. Чтобы это исправить, надо второй форме как-то передать сведения о первой форме. Для этого воспользуемся передачей ссылки на форму в конструкторе.

Итак перейдем ко второй форме и перейдем к ее коду - нажмем правой кнопкой мыши на форму и выберем View Code (Просмотр кода). Пока он пустой и содержит только конструктор. Поскольку C# поддерживает перегрузку методов, то мы можем создать несколько методов и конструкторов с разными параметрами и в зависимости от ситуации вызывать один из них. Итак, изменим файл кода второй формы на следующий:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;

```

```

using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace HelloApp
{
    public partial class Form2 : Form
    {
        public Form2()
        {
            InitializeComponent();
        }

        public Form2(Form1 f)
        {
            InitializeComponent();
            f.BackColor = Color.Yellow;
        }
    }
}

```

Фактически мы только добавили здесь новый конструктор `public Form2(Form1 f)`, в котором мы получаем первую форму и устанавливаем ее фон в желтый цвет. Теперь перейдем к коду первой формы, где мы вызывали вторую форму и изменим его на следующий:

```

private void button1_Click(object sender, EventArgs e)
{
    Form2 newForm = new Form2(this);
    newForm.Show();
}

```

Поскольку в данном случае ключевое слово `this` представляет ссылку на текущий объект - объект `Form1`, то при создании второй формы она будет получать ее (ссылку) и через нее управлять первой формой.

Теперь после нажатия на кнопку у нас будет создана вторая форма, которая сразу изменит цвет первой формы.

Мы можем также создавать объекты и текущей формы:

```

private void button1_Click(object sender, EventArgs e)
{
    Form1 newForm1 = new Form1();
    newForm1.Show();

    Form2 newForm2 = new Form2(newForm1);
    newForm2.Show();
}

```

При работе с несколькими формами надо учитывать, что одна из них является главной - которая запускается первой в файле `Program.cs`. Если у нас одновременно открыта куча форм, то при закрытии главной закрывается все приложение и вместе с ним все остальные формы.