МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Программное обеспечение информационных технологий»

ИНФОРМАТИКА

Методические рекомендации к лабораторным работам для студентов направления подготовки 13.03.02 «Электроэнергетика и электротехника» очной формы обучения

Часть 2

Могилев 2024

УДК 004
ББК 32.97
И74

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Программное обеспечение информационных технологий» «30» августа 2024 г., протокол № 1

Составитель канд. физ.-мат. наук, доц. П. Я. Чудаковский

Рецензент канд. техн. наук, доц. В. М. Ковальчук

В методических рекомендациях кратко изложены теоретические сведения, необходимые для выполнения лабораторных работ. Рекомендации составлены в соответствии с учебной программой по дисциплине «Информатика» для студентов направления подготовки 13.03.02 «Электроэнергетика и электротехника» очной формы обучения.

Учебное издание

ИНФОРМАТИКА

Часть 2

Ответственный за выпуск	В. В. Кутузов
Корректор	И. В. Голубцова
Компьютерная верстка	Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 16 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

1 Лабораторная работа № 5. Работа с главным меню системы C# в Visual Studio. Форматированный ввод-вывод информации. Программирование линейных алгоритмов. Работа с отладчиком

Цель работы: ознакомиться с языком программирования C# и системой MS Visual Studio; осуществить форматированный ввод-вывод информации.

1.1 Краткие теоретические сведения

Самая простая программа на языке C# не может быть написана без использования базовых понятий объектно-ориентированного программирования, прежде всего таких, как класс и метод. Приведем некоторые необходимые для начала работы понятия и термины без пояснений.

Рассмотрим структуру программы на языке C# на простейшем примере:

```
using System; // подключение пространства имен System
class Program // объявление класса Program
{
    static void Main() // объявление метода Main()
    {
        Console.WriteLine("Привет, C#");
    }
}
```

Первой строкой подключается пространство имен **System**, в котором содержатся описания встроенных объектов и методов среды **.NET**. Программный код записывается в фигурных скобках внутри класса (с именем **Program**) и содержит описания элементов класса: переменных, методов. Переменную класса в ООП называют **полем**, а функцию объекта – **методом**. Сразу отметим, что имена полей в C# принято начинать с малой (строчной) буквы, а имена классов и методов – с заглавной. Обязательным для любой программы на C# является метод **Main()**. Его имя предваряется модификаторами, уточняющими, что метод ничего не возвращает (**void**) и не требует создания экземпляра класса (**static**).

Язык программирования C# является строго типизированным. При объявлении переменной обязательно указывается ее тип, который в дальнейшем не может быть изменен. Для любого стандартного типа языка C# в библиотеке .NET содержится соответствующий класс с описанием его свойств и методов.

В дальнейшем при создании компьютерных программ будем пользоваться интегрированной средой разработки **Visual Studio**, которая используется для редактирования, отладки и сборки кода, а также для публикации приложения. В дополнение к стандартному редактору и отладчику, предоставляемых большинством интегрированных сред разработки, **Visual Studio** включает компи-

ляторы, средства завершения кода, графические конструкторы и многие другие функции для улучшения процесса разработки программного обеспечения.

Консоль – совокупность стандартных устройств ввода-вывода (клавиатура, монитор). Для работы с консолью предназначен класс **Console** в пространстве имен **System**. Основные методы: **Console.Write**, **Console.WriteLine**, **Console.Read**, **Console.ReadLine**.

Пусть в программе заданы: **int** d = 48; **double** y = 5,7412; **string s** = "бит". В простейшем выводе на консоль можно использовать конкатенацию, например,

```
Console.WriteLine("d = " + d + " " + s + " y = " + y);
```

Будет выведено: d = 48 бит y = 5,7412.

Заметим, что в коде программы разделителем целой и дробной части десятичной дроби является **точка**, а при вводе и выводе разделитель определяется локализацией операционной системы, т. е. для русифицированных ОС – **запятая**!

Форматный вывод – использование в строке вывода местозаполнителей (*placeholder*), которые включают параметры формата и управляющие символы:

{ номер [, к-во позиций] [: формат] }

Номера элементов в списке вывода могут идти не по порядку и повторяться. Количество позиций под выводимое значение может иметь знак «минус». Тогда оно выравнивается внутри отведенного места по левому краю, иначе – по правому. Формат вывода обозначается латинскими буквами, например: **F** или **f** – количество десятичных цифр дробной части числа (**f2** – две). В строке вывода могут использоваться управляющие символы (они предваряются косой чертой – слешем), например: **\n** – переход на новую строку, **\t** – табуляция. Заметим, что выводимую строку в кавычках разрывать нельзя, а список переменных после закрывающих кавычек можно записывать в новой строке:

```
Console.WriteLine("d = {0,6} {1,-8} y = {2} \n d = {0,-6} {1,-8} y = {2:f2} ", d, s, y);
```

Будет выведено следующее.

```
d=    48 бит      y = 5,7412
 d=48     бит      y = 5,74
```

Начиная с версии 6.0 C# для вывода данных можно использовать **интерполяцию строк**, размещая имена переменных прямо в строке, предварив ее символом $, например,

```
Console.WriteLine($"d = {d} y = {s} \n d = {d} y = {y}");
```

Для ввода данных с консоли используют методы **ReadLine** – возвращает строку типа **string** и **Read** – возвращает код символа.

Для дальнейшей работы требуется преобразование в нужный тип! Для этого используют метод **Parse**, который выполняет разборку (парсинг) строки или класс **Convert**, который содержит методы преобразования в требуемый тип:

```
    char c = (char)Console.Read(); // ввод кода и преобразование u1074 в
символ
    string st = Console.ReadLine(); // ввод строки
    int k = int.Parse(st); или int m = Convert.ToInt32(st); //
преобразование в целое
    Console.WriteLine("строка st={0} число k={1} m={2}", st, k, m); //
вывод
    double x = double.Parse(st); // преобразование в вещественный тип
    double y = Convert.ToDouble(st); // преобразование в вещественный
тип
```

Запись некоторых арифметических операций в языке C# отличается от принятой в математике: ***** умножение, / деление, **%** остаток от целочисленного деления.

Создание консольных приложений в Visual Studio

Далее представлены примеры простейших консольных приложений, созданных средствами **Visual Studio**. Для того чтобы создать консольное приложение, необходимо выполнить следующее:

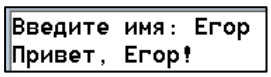
1) запустить **MS Visual Studio**;
2) на появившейся стартовой странице выбрать **Создать проект** (New Project);
3) на следующей странице выбрать тип приложения **Консольное** (Console Application) и шаблон **Visual C#**;
4) в поле ввода **Расположение** (Location) выбрать или задать рабочую папку, в которой будет сохраняться проект;
5) ввести имя проекта и **Решения** (Solution), например **app01**.

После выполнения пп. 1-5 откроется окно программного кода с шаблоном. В нем с помощью ключевых слов **using** объявлены пространства имен, из которых можно использовать стандартные классы непосредственно, без указания имени пространства (в данном примере используется пространство имен **System**). С помощью ключевого слова **namespace** создано собственное пространство имен, имя которого совпадает с именем проекта **app01**. Для упрощения шаблона можно сразу удалить «лишние» строки и выражения, которые не будут использоваться в программе, например неиспользуемые параметры метода **Main()** и пространства имен.

Основной код программы необходимо добавить в тело метода **Main**:

```
    using System;
    class Program
    {
        static void Main()
        {
            Console.Write("Введите имя: "); // вывод приглашения
            string nam = Console.ReadLine(); // ввод имени, тип string
            Console.WriteLine("Привет, " + nam + "!"); // вывод
приветствия
    // в конец добавляем строку с методом ReadKey,
// который заставляет программу ожидать нажатия любой клавиши
            Console.ReadKey();
        }
    }
```

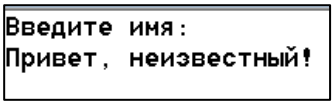
После набора кода программа обязательно тестируется (меню **Отладка** (Debug) или клавиша F5). При этом будет создан исполняемый PE-файл (Portable Executable) с расширением **.exe** и помещен в папку **\bin\Debug** проекта. Его можно переносить в другое место и запускать без системы Visual Studio при наличии виртуальной машины – общеязыковой среды исполнения CLR (**Common Language Runtime**) платформы .NET Framework. Протестировав программу, в окне вывода получим следующее.



Модифицируем программу, добавив условие: если не введено имя, то выводится строка «неизвестный»:

```
      if (nam == "")
      {
        nam = "неизвестный";// проверка условия
      }
```

После тестирования окончательного варианта программы, не выполнив ввода имени в консоль, получим следующее.



Приведем еще один пример консольного приложения, которое по введенному радиусу вычисляет и выводит на консоль длину окружности и площадь круга. Основной код программы, как всегда, находится в теле метода **Main**:

```
    using System;
    class Program
    {
        static void Main()
        {
```

```
            Console.Write("Введите радиус: ");
            // ввод и преобразование типа string в int
            int r = int.Parse(Console.ReadLine());
            double p = 2 * Math.PI * r; double s = Math.PI * r * r;
            // форматный вывод – три десятичных знака
            Console.WriteLine("Длина окружности {0:f3}, площадь круга
{1:f3}", p, s);
            Console.ReadKey();
        }
    }
```

После запуска приложения результат может выглядеть следующим образом.

```
Введите радиус: 5
Длина окружности 31,416, площадь круга 78,540
```

1.2 Практическое задание

1.2.1 Знакомство с базовыми типами данных C#.

1 Самостоятельно изучите базовые типы данных C#, операции и их приоритеты.

1.2.2 Разработка консольных приложений.

1 Создайте консольные приложения, которые вычисляют и выводят:

а) среднее арифметическое sred двух введенных чисел a и b;

б) площадь s и периметр p прямоугольника по сторонам a и b;

в) площадь поверхности $s = 4 \pi r^2$ и объем шара $v = 4/3 \pi r^3$ по радиусу r;

г) значение функции $z = x^2 + x y - y^2$ (ввод x и y);

д) стоимость товара в трех валютах по его стоимости в белорусских рублях (курсы вводятся с клавиатуры);

е) стоимость поездки на автомобиле (ввод: s – расстояние; b – расход бензина на 100 км; c – цена бензина за 1 л);

ж) смещение от положения равновесия точки, совершающей гармонические колебания, $x = A \sin(wt)$ (ввод: амплитуда A; угловая частота w; время t).

2 Самостоятельно рассмотрите приемы работы с отладчиком в Visual Studio.

Контрольные вопросы

1 Объясните, что такое тип.
2 Как можно определить понятие «переменная»?
3 Дайте определение идентификатора.
4 Объясните назначение отдельных частей простейшей программы на C#.

5 Каково назначение статического метода Main()?
6 Возможно ли написать программу на C#, не применяя классов?
7 Что такое тип void?
8 Какие методы класса Console применяются для ввода и вывода данных?
9 Что такое пространство имен?
10 Какое из слов конструкции System.Console.ReadLine() является названием пространства имен?
11 Для каких целей применяется директива **using?**

2 Лабораторная работа № 6. Программирование разветвляющихся алгоритмов. Оператор if. Программирование с использованием оператора switch

Цель работы: сформировать навыки реализации алгоритмов ветвления.

2.1 Краткие теоретические сведения

Разветвляющийся алгоритм – алгоритм, содержащий хотя бы одно условие, в результате проверки которого может осуществляться разделение на несколько параллельных ветвей алгоритма.

Для реализации ветвлений в языке C# реализован условный оператор **if**. Его полная форма имеет вид

```
if (< условие >){
        // что делать, если условие верно
  }
    else{
        // что делать, если условие неверно
};
```

К особенностям оператора **if** стоит отнести:
– вторая часть (**else** …) может отсутствовать (неполная форма);
– если в блоке один оператор, то можно убрать скобки **{** и **}**.

Рассмотрим работу оператора на примере. Пусть требуется ввести два числа и вывести наибольшее из них. Алгоритм решения представлен на рисунке 2.1

Программная реализация алгоритма этой задачи может быть представлена в виде следующего кода:

```
using System;
class Program
{
    static void Main()
    {
        int a, b, max;
```

```
            Console.WriteLine("Введите два целых числа.");
            Console.Write("Первое число:");
            a = Convert.ToInt32(Console.ReadLine());
            Console.Write("Второе число:");
            b = Convert.ToInt32(Console.ReadLine());

            if (a > b)
            {
                max = a;
            }
            else
            {
                max = b;
            };

            Console.WriteLine("Наибольшее число {0}", max);
            Console.WriteLine("Для выхода из приложение нажмите Enter");
            Console.ReadLine();
        }
    }
```

В случае неполной формы условного оператора блок-схема представлена на рисунке 2.2.

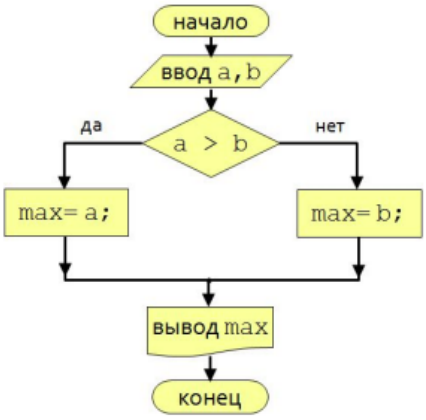


Рисунок 2.1 – Блок-схема поиска наибольшего из двух чисел

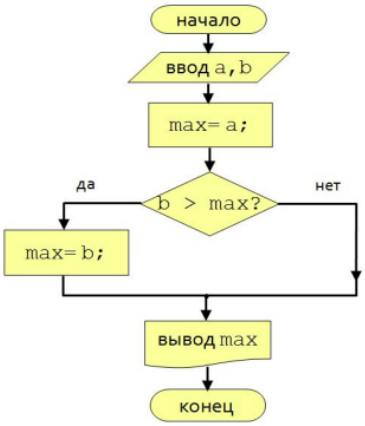


Рисунок 2.2 – Блок-схема неполного оператора **if**

```
using System;
class Program
{
    static void Main()
    {
        int a, b, max;
        Console.WriteLine("Введите два целых числа.");
        Console.Write("Первое число:");
        a = Convert.ToInt32(Console.ReadLine());
        Console.Write("Второе число:");
        b = Convert.ToInt32(Console.ReadLine());
```

```
            max = a;
            if (b > max)
            {
                max = b;
            }
            Console.WriteLine("Наибольшее число {0}", max);
            Console.WriteLine("Для выхода из приложения нажмите Enter");
            Console.ReadLine();
        }
    }
```

Как альтернатива оператору **if**…**else** может быть использован **тернарный оператор (?)**. Далее приведена общая форма этого оператора:

Выражение1* ? *Выражение2* : *Выражение3;

Если ***Выражение1*** – истинно, то берётся ***Выражение2***, иначе берётся ***Выражение3***. Далее представлен код, показывающий работу тернарного оператора, в задаче вывода на экран чётного или нечётного числа:

```
using System;
class Program
{
    static void Main()
    {
      int x;
       Console.WriteLine("Введите целое число.");
      x = Convert.ToInt32(Console.ReadLine());
      Console.WriteLine(x + " - " + ((x % 2 == 0) ? "четное число" :
      "нечетное число"));
      Console.WriteLine("Для выхода из приложение нажмите Enter");
      Console.ReadLine();
    }
}
```

Для разветвления процесса выполнения программы предназначен оператор **switch**:

```
switch (выражение)
{
  case  выражение_1:  блок_1;  break;
  case  выражение_2:  блок _2;  break;
  ……..
  case  выражение_n:  блок_n;  break;
  [default: блоки_XXX; ]}
```

Пример использования конструкции **switch…case**:

```
using System;
class Program
```

```
{
    static void Main()
    {
        int caseSwitch = 1;
        switch (caseSwitch)
        {
            case 1:
                Console.WriteLine("Case 1");
                break;
            case 2:
                Console.WriteLine("Case 2");
                break;
            default:
                Console.WriteLine("Default case");
                break;
        }
    }
}
```

Стоит отметить, что в случае переменной caseSwitch = 1 в консоль будет выведена строка “Case 1”. Если, например, переменная caseSwitch = 7, то управление передается блоку **default**, а в консоль будет выведена строка “Deafault case”.

2.2 Практическое задание

2.2.1 Работа с операторами ветвления и выбора.

Создайте консольные приложения, в которых выполняются заданные действия:

1) проверяется делимость введенного целого числа n на d (ввод: число n; делитель d);
2) по введенному номеру месяца выводится название поры года (зима, весна, лето, осень) и сообщение: сессия, каникулы, 1-й семестр, 2-й семестр;
3) проверяется соответствие веса и роста (ввод: рост и вес; вывод одного из сообщений: «Норма», «Нужно похудеть», «Нужно поправиться»). Нормальный вес (в кг) = (рост (в см) – 100) ± 10 %.;
4) выводится название дисциплины по введенной первой букве: ф – физика, м – математика, и – история, г – география, б – биология;
5) выводится название страны и ее столицы по введенной первой букве: б – Беларусь, Минск, р – Россия, Москва, л – Литва, Вильнюс;
6) выводится название дня недели по введенному номеру (1 – пн, 2 – вт, … и т. д.), сообщение «рабочий день» или «выходной»;
7) введенная цифра (от 0 до 5) выводится прописью;
8) проверяется правильность логина строго из пяти букв и пароля из шести цифр.

Контрольные вопросы

1 Назовите операторы выбора (ветвлений).
2 Какие операторы не могут входить в условный оператор?
3 Что такое сокращенная форма условного оператора?
4 Как устанавливается соответствие между **if** и **else** при вложениях условных операторов?

3 Лабораторная работа № 7. Операторы цикла for. Операторы цикла while и do…while

Цель работы: сформировать навыки реализации циклических алгоритмов.

3.1 Краткие теоретические сведения

Цикл – это многократное выполнение одинаковой последовательности действий. В C# доступны четыре разновидности цикла:

1) цикл while;
2) цикл do… while;
3) цикл for;
4) цикл foreach.

Форма оператора **while**:

```
while (условие)
{
оператор;
}
```

В операторе **while** можно использовать сложные условия. Если в теле цикла только один оператор, то скобки { и } можно не писать.

Рассмотрим цикл **while** в действии на примере. Пусть необходимо ввести целое положительное число (<2000000000) и определить количество цифр в нём. Блок-схема задачи представлена на рисунке 3.1.

Программная реализация этой задачи может быть представлена в следующем виде:

```
using System;
class Program
{
    static void Main()
    {
        int n = int.Parse(Console.ReadLine());
        int count = 0;
        while (n != 0)
        {
            count++;
```

```
            n = n / 10;// так как n-целое, деление целочисленное
        }
        Console.WriteLine(count);
    }
}
```

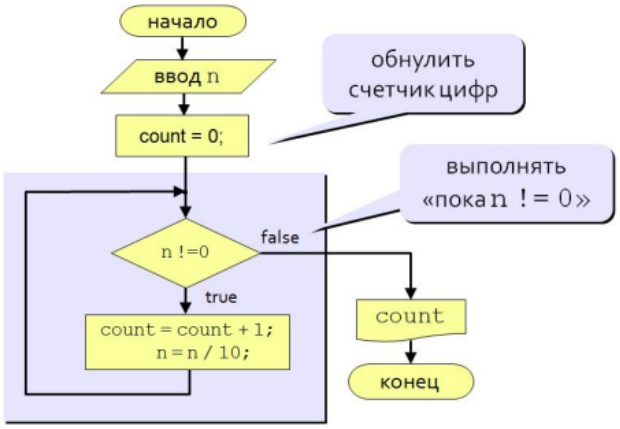


Рисунок 3.1 – Блок-схема подсчета количества цифр в числе

Цикл **do…while** является циклом с постусловием, т. е. условие проверяется в конце:

```
do
{
операторы
} while (условие);
```

Тело цикла **do…while** всегда выполняется хотя бы один раз. Далее приведен пример использования цикла в задаче по организации ввода числовых данных, ограничив значения числами от 1 до 99. Блок схема представлена на рисунке 3.2.

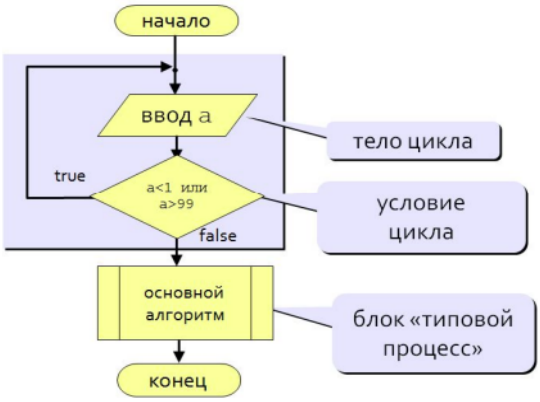


Рисунок 3.2 – Блок-схема организации ввода числовых данных

```
using System;
class Program
{
```

```
        static void Main()
        {
            int a, count = 0;
            do
            {
                Console.Write("Введите возраст:");
                a = int.Parse(Console.ReadLine());
                count++;
            }
            while (a < 1 || a > 99);// Повторять пока условие
истинно(true)
            Console.WriteLine("Вы сделали " + count + " попыток ввода");
        }
    }
```

Оператор **for** позволяет реализовать цикл с параметром:

```
for (инициализация; условие; итерация)
{
оператор;
}
```

Если тело цикла состоит из одного оператора, то операторные скобки { и } можно не писать. Условие цикла каждый раз пересчитывается. Блок-схема работы цикла **for** представлена на рисунке 3.3.

Рисунок 3.3 – Блок-схема циклического вывода сообщения в консоль

Программная реализация блок схемы, представленной на рисунке 3.3, имеет вид

```
using System;
class Program
{
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
```

```
            {
                Console.Write(i + " ");
                Console.WriteLine("Привет");
            }
        }
    }
```

Оператор цикла **foreach** служит для циклического обращения к элементам коллекции, которая представляет собой группу объектов:

foreach (тип имя_переменной_цикла **in** коллекция**) оператор;**

```
using System;
class Program
{
    static void Main(string[] args)
    {
        string s = "Hello, Foreach";
        foreach (char c in s)
            Console.Write("{0} ", c);
    }
}
```

При работе с циклами используют операторы управления **continue**, **break**:
1) выполнение следующей итерации цикла – **continue**;
2) прерывание текущей итерации цикла – **break**.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        string s = "1. Привет, Foreach. \n2. А также break и
continue! А это не выведется";
        foreach (char c in s)
        {
            // Пропускаем цифры
            if (c >= '0' && c <= '9') continue;
            // Если встречаем !, прерываем цикл
            if (c == '!') break;
            Console.Write("{0} ", c);
        }
    }}
```

Довольно часто один цикл приходится вкладывать в другой. Далее приведен код программы с вложенным циклом на примере заполнения экрана звездочками:

```
using System;
class Program
{
```

```
        static void Main(string[] args)
        {
            //Внешний цикл
            for (int i = 0; i < 80; i++)
                //Внутренний цикл
                for (int j = 0; j < 24; j++)
                {
                    Console.SetCursorPosition(i, j); //  устанавливаем
позицию курсора
                    Console.Write('*');
                    System.Threading.Thread.Sleep(20);// делаем паузу
                    Console.Title = "i=" + i + " j=" + j;
                }
            Console.ReadKey();
        }
    }
```

3.2 Практическое задание

3.2.1 Работа с циклическими конструкциями.

Создайте консольные приложения, в которых выполняются заданные действия:

1) вычисляется сумма всех нечетных чисел от n_1 до n_2;
2) вычисляется сумма квадратов n натуральных чисел, начиная с k (ввод k и n);
3) повторяются вычисления площади круга по вводимому радиусу r до тех пор, пока не введена буква z или Z;
4) генерируется 8 случайных чисел в интервале (–30, 30). Выводятся эти числа и сообщения: отрицательное – положительное, четное – нечетное;
5) генерируется 10 случайных чисел в интервале (– 20, 20). Выводятся только положительные числа и сообщения: четное – нечетное;
6) генерируются случайные числа в интервале (–40, 40) до тех пор, пока очередное число не превышает 30. Выводятся только нечетные числа и сообщения: отрицательное – положительное;
7) генерируются случайные числа в интервале (0, 20) до тех пор, пока их сумма не превысит S (вводится с клавиатуры). Нумеруются и выводятся эти числа и их сумма.

Контрольные вопросы

1 Охарактеризуйте циклические конструкции языка C#.

4 Лабораторная работа № 8. Обработка одномерных массивов. Сортировка массивов

Цель работы: сформировать навыки работы с одномерными массивами в C#.

4.1 Краткие теоретические сведения

Массив – набор элементов одного и того же типа, объединённых общим именем. Массивы в C# можно использовать по аналогии с тем, как они используются в других языках программирования. Однако в C# массивы имеют существенные отличия: они относятся к ссылочным типам данных, более того – реализованы как объекты. Фактически имя массива является ссылкой на область кучи, в которой последовательно размещается набор элементов определённого типа. Выделение памяти под элементы происходит на этапе инициализации массива. За освобождением памяти следит система сборки мусора – неиспользуемые массивы автоматически утилизируются данной системой. Рассмотрим различные типы массивов.

Одномерный массив – это фиксированное количество элементов одного и того же типа, объединённых общим именем, где каждый элемент имеет свой номер. Нумерация элементов массива в C# начинается с нуля, т. е., если массив состоит из 10 элементов, они будут иметь следующие номера: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Одномерный массив в C# является объектом, поэтому он создаётся в два этапа. Сначала объявляется ссылочная переменная на массив, затем выделяется память под требуемое количество элементов, ссылочной переменной присваивается адрес на область памяти, в которой расположен массив. Тип массива определяет тип данных каждого элемента массива. Количество элементов, которые будут храниться в массиве, определяется размером массива.

В общем случае процесс объявления переменной типа массив и выделение необходимого объёма памяти может быть разделён. Кроме того, на этапе объявления массива можно произвести его инициализацию. Поэтому для объявления одномерного массива может использоваться одна из следующих форм записи:

```
class TestArraysClass
{
    static void Main()
    {
        // Описание объекта типа массив
        int[] array0;
        // Выделение места под массив
        array0 = new int[5];
        // Объявление одномерного массива из 5 элементов
        int[] array1 = new int[5];
        // Объявление массива и заполнение его элементами
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };
        // Альтернативный вариант создания заполненного массива
```

```
        int[] array3 = { 1, 2, 3, 4, 5, 6 };
    }
}
```

Обращение к элементам массива происходит с помощью индекса. Для этого нужно указать имя массива и в квадратных скобках его номер. Например, a[0], b[10], c[i]. Так как массив представляет собой набор элементов, объединённых общим именем, его обработка обычно производится в цикле. Рассмотрим несколько простых примеров работы с одномерными массивами.

Вывод массива:

```
using System;
class Program
{
    static void Main()
    {
        int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        int i;
        for (i = 0; i < 10; i++)
        {
            Console.Write("{0} ", a[i]);}
        Console.WriteLine(); }}
```

Заполнение каждого элемента массива квадратами его индекса:

```
using System;
class Program
{
    static void Main()
    {
        int[] a = new int[10];
        int i;
        for (i = 0; i < 10; i++) {a[i] = i * i;}
        for (i = 0; i < 10; i++) {Console.WriteLine(a[i]);}
        Console.WriteLine();
    }
}
```

Массив можно использовать как параметр. Так как имя массива фактически является ссылкой, он передаётся в метод по ссылке. Все изменения элементов массива, являющегося формальным параметром, отразятся на элементах соответствующего массива – фактического параметра. Рассмотрим пример передачи массива как параметра:

```
using System;
class Program
{
    static void Print(int n, int[] a)
    // n – размерность массива, a – ссылка на массив
    {
```

```
            for (int i = 0; i < n; i++) Console.Write("{0} ", a[i]);
            Console.WriteLine();
        }
        static void Change(int n, int[] a)
        {
            for (int i = 0; i < n; i++)
                if (a[i] < 0) a[i] = -a[i];
            // изменяются элементы массива
        }
        static void Main()
        {
            int[] myArray = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };
            Print(10, myArray);
            Change(10, myArray);
            Print(10, myArray);
        }
    }
```

Массивы в C# реализованы как объекты. Если говорить более точно, они реализованы на основе базового класса **Array**, определённого в пространстве имен System. Данный класс содержит различные свойства и методы. Например, свойство **Length** позволяет определять количество элементов в массиве. Преобразуем предыдущий пример:

```
using System;
class Program
{
    static void Print(int[] a) // передаем только ссылку на массив
    {
 for (int i = 0; i < a.Length; i++) { Console.Write("{0} ", a[i]); }
        Console.WriteLine();
    }
    static void Change(int[] a)
    {
        for (int i = 0; i < a.Length; i++)
        { if (a[i] < 0) a[i] = -a[i]; }
    }
    static void Main()
    {
        int[] A = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };
        Print(A);
        Change(A);
        Print(A);
    }
}
```

Относительно сортировки массивов укажем возможные способы её реализации: случайная сортировка, сортировка пузырьком, сортировка перемешиванием сортировка вставками, сортировка по частям, блинная сортировка, сортировка слиянием и др.

В классе **Array** содержится метод **Sort()**, позволяющий осуществить сортировку элементов одномерного массива.

4.2 Практическое задание

4.2.1 Работа с одномерными массивами.

1 Дан массив размера N. Вывести его элементы в обратном порядке.

2 Дан целочисленный массив размера N. Найти количество различных элементов в данном массиве.

3 Дан массив A размера N. Сформировать новый массив B того же размера по следующему правилу: элемент B_K равен сумме элементов массива A с номерами от K до N.

4 Дан массив размера N. Поменять местами его минимальный и максимальный элементы.

5 Дан массив A размера N (≤ 6). Упорядочить его по возрастанию методом сортировки простым обменом («пузырьковой» сортировкой): просматривать массив, сравнивая его соседние элементы (A1 и A2, A2 и A3 и т. д.) и меняя их местами, если левый элемент пары больше правого; повторить описанные действия N − 1 раз. Для контроля за выполняемыми действиями выводить содержимое массива после каждого просмотра.

6 Дан целочисленный массив A размера N. Назовем серией группу подряд идущих одинаковых элементов, а длиной серии – количество этих элементов (длина серии может быть равна 1). Сформировать два новых целочисленных массива B и C одинакового размера, записав в массив B длины всех серий исходного массива, а в массив C – значения элементов, образующих эти серии.

Контрольные вопросы

1 Что такое одномерный массив? Охарактеризуйте основные особенности при работе с одномерными массивами в C#.

2 Охарактеризуйте методы класса **Array**.

3 Можно ли массив использовать в качестве параметра?

5 Лабораторная работа № 9. Обработка двумерных массивов

Цель работы: сформировать навыки работы с двумерными массивами в C#.

5.1 Краткие теоретические сведения

Для работы с двумерными массивами используются два измерения и требуется два индекса. Таблица Excel – хороший аналог двумерного массива. Только в двумерных массивах и номер столбца, и номер строки – это числа. Для работы с двумерными, да и, вообще, с многомерными массивами чаще

всего используются вложенные циклы. Далее приведены способы задания массивов:

```
class Program
{
    static void Main(string[] args)
    {
       // Объявление двумерного массива
       int[,] multiDimensionalArray1 = new int[2, 3];
       // Объявление и заполнение двумерного массива
       int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
    }
}
```

Также можно создавать ступенчатые массивы или так называемые массивы массивов. Это одномерный массив. Каждый элемент в этом массиве является ссылкой на другой одномерный массив:

```
class Program
{
    static void Main(string[] args)
    {
        // Объявление массива массивов(ступенчатый массив)
        int[][] jaggedArray = new int[3][];
        // Пример заполнения первого элемента ступенчатого массива
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
        jaggedArray[1] = new int[5] { 1, 2, 3, 4, 5 };
        jaggedArray[2] = new int[3] { 1, 2, 3 };
    }
}
```

Приведем примеры программного кода работы с двумерным массивом, в котором реализованы методы поиска среднего значения, максимального элемента массива, минимального элемента массива, подсчета количества положительных элементов массива, вывода массива на экран и метод, возвращающий массив в виде строки.

```
using System;
namespace ArrayTwoDimensionClass
{
    class MyArrayTwoDim
    {
        int[,] a;
        public MyArrayTwoDim(int n, int el)
        {
            a = new int[n, n];
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    a[i, j] = el;
        }
```

```
        public MyArrayTwoDim(int n, int min, int max)
        {
            a = new int[n, n];
            Random rnd = new Random();
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    a[i, j] = rnd.Next(min, max);
        }
        public int Min
        {
            get
            {
                int min = a[0, 0];
             // Находим минимальный элемент
             // В двухмерном массиве для получения размерности нужно
                // использовать
                // метод GetLength, а в скобках указывать измерение
                for (int i = 0; i < a.GetLength(0); i++)
                    for (int j = 0; j < a.GetLength(1); j++)
                        if (a[i, j] < min) min = a[i, j];
                return min;
            }
        }
        public int Max
        {
            get
            {
                int max = a[0, 0];
                for (int i = 0; i < a.GetLength(0); i++)
                    for (int j = 0; j < a.GetLength(1); j++)
                        if (a[i, j] > max) max = a[i, j];
                return max;
            }
        }
    // Свойство - подсчет количества положительных элементов массива

        public int CountPositive
        {
            get
            {
                int count = 0;
                for (int i = 0; i < a.GetLength(0); i++)
                    for (int j = 0; j < a.GetLength(1); j++)
                        if (a[i, j] > 0) count++;
                return count;
            }
        }
        // Свойство - подсчет среднего арифметического
        public double Average
        {
            get
```

```
            {
                double sum = 0;
                for (int i = 0; i < a.GetLength(0); i++)
                    for (int j = 0; j < a.GetLength(1); j++)
                        sum += a[i, j];
                return sum / a.GetLength(0) / a.GetLength(1);
            }
        }
        // Метод, который возвращает массив в виде строки
        public string ToString()
        {
            string s = "";
            for (int i = 0; i < a.GetLength(0); i++)
            {
                for (int j = 0; j < a.GetLength(1); j++)
                    s += a[i, j] + " ";
                s += "\n"; // Переход на новую строчку
            }
            return s;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyArrayTwoDim a = new MyArrayTwoDim(2, 0, 10);
            Console.WriteLine(a.ToString());
            Console.WriteLine("\nМаксимальный элемент:" + a.Max);
            Console.WriteLine("Минимальный элемент:" + a.Min);
            Console.WriteLine("Среднее значение элементов:" +
a.Average);
            Console.WriteLine("Количество положительных элементов: "
+  a.CountPositive);
        }
    }
}
```

5.2 Практическое задание

5.2.1 Работа с двумерными массивами.

1 Дан вещественный массив N × N. Найти произведение элементов массива, расположенных между максимальным и минимальным элементами.

2 Дан целочисленный массив N × M. Найти произведение элементов массива с четными номерами.

3 Дан целочисленный массив N × M. Найти сумму элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.

4 Дан вещественный массив N × M. Найти сумму положительных элементов.

5 Дан целочисленный массив N × N. Найти сумму элементов массива с

нечетными номерами.

6 Дан вещественный массив N × N. Найти сумму элементов массива, расположенных между первым и вторым нулевым элементами.

7 Дан вещественный массив N × N. Найти произведение элементов массива, расположенных после максимального элемента.

8 Дан целочисленный массив N × M. Найти произведение отрицательных элементов массива.

9 Дан целочисленный массив N × N. Найти сумму модулей элементов массива, расположенных после первого элемента, равного нулю.

10 Дан вещественный массив N × M. Найти сумму элементов массива, расположенных после первого положительного элемента.

11 Дан целочисленный массив N × M. Найти сумму модулей элементов массива, расположенных после первого элемента, равного s.

12 Дан вещественный массив N × M. Найти в заданном своим номером столбце максимальный по модулю элемент.

Контрольные вопросы

1 В чем состоит отличие одномерных массивов от двумерных?
2 Как создать двумерный массив в C#?
3 Опишите основные особенности при работе с двумерными массивами.

6 Лабораторная работа № 10. Строковые типы. Обработка текстов и строк

Цель работы: сформировать навыки работы со строками.

6.1 Краткие теоретические сведения

Для работы со строками предназначены классы **String** и **StringBuilder**. Строки типа **string** в **C#** – **неизменяемый** тип данных. Методы, изменяющие содержимое строки, на самом деле создают новую копию строки, а неиспользуемые «старые» копии автоматически удаляются сборщиком мусора.

Примеры создания строк:

```
string s; // переменная объявлена, инициализация отложена;
string st = "строка"; // инициализация строковым литералом;
string u = new string(“ “, 20); // создание с помощью конструктора;
char[] a = { 'e', 'n', 'd' }; // создание массива символов;
string v = new string(a); // строка из массива символов.
```

Основные операции для работы со строками: присваивание = ; проверка на равенство = =; на неравенство **!**=; сцепление (конкатенация) строк +; обращение к символу строки по индексу **[k]**.

Строки равны, если имеют одинаковое количество символов и совпадают посимвольно. Обращаться к отдельному символу строки по индексу можно только для получения значения, но не для его изменения.

Длина строки (количество символов) определяется свойством **Length**. Пустая строка – это экземпляр класса **String**, содержащий **0** символов: **string s = ""**.

Приведем некоторые методы класса **System.String**:

– **Compare**, **CompareTo** – сравнение строк;

– **Concat** – конкатенация (сцепление) строк;

– **Copy** – создание копии строки;

– **IndexOf(s)**, **LastIndexOf(s)** – определение позиции первого (последнего) вхождения подстроки или символа **s**;

– **Substring(k1, k2)** – выделение подстроки с позиции **k1** по **k2**;

– **Replace(s, z)** – замена всех вхождений подстроки **s** новой подстрокой или символом **z**;

– **Insert(k)** – вставка подстроки с позиции **k**;

– **Remove (k,n)** – удаление n символов с позиции **k**;

– **Trim()**, **TrimStart()**,**TrimEnd()** – удаление концевых пробелов;

– **Split(d)** – разделение строки в массив строк по символу-разделителю **d** (или массиву символов);

– **Join(d, mas)** – слияние массива строк **mas** в единую строку с разделителем **d**;

– **Format** – форматирование с заданными спецификаторами формата.

Некоторые спецификаторы формата строк:

– **C** или **c** – вывод значений в денежном (currency) формате;

– **F** или **f** – вывод значений с фиксированной точностью;

– **G** или **g** – формат общего вида;

– **P** или **p** – вывод числа в процентном формате.

Для создания изменяемых строк предназначен класс **StringBuilder**, который определен в пространстве имен **System.Text**. Требует создания экземпляра! Позволяет **изменять** значение своих экземпляров. При создании экземпляра обязательно использовать **new** и конструктор, например:

– StringBuilder a = **new** StringBuilder();

– StringBuilder b = **new** StringBuilder("Privet");

– StringBuilder d = **new** StringBuilder("Privet", 10).

Некоторые методы класса **StringBuilder**:

– **Append(s)** – добавление в конец строки;

– **AppendFormat(…)** – добавление форматированной строки;

– **Insert(k)** – вставка подстроки с позиции **k**;

– **Remove (k, n)** – удаление **n** символов с позиции **k**;

– **Replace(s, z)** – замена всех вхождений подстроки **s** новой подстрокой или символом **z;**

– **ToString()** – преобразование в строку типа **string**;

– **Capacity** – получение или установка емкости буфера. Если устанавливаемое значение меньше текущей длины строки или больше максимального, генерируется исключение ArgumentOutOfRangeException;

– **MaxCapacity** – максимальный размер буфера.

6.2 Практическое задание

6.2.1 Работа со строками.

Создайте консольные приложения, в которых выполняются заданные действия над введенными строками:

1) все пробелы в строке заменяются на символы подчеркивания _;
2) стоящие рядом две точки заменяются на три звездочки ***;
3) из строки удаляется заданное слово;
4) во введенной строке заданное слово заменяется на другое;
5) выводится подстрока, расположенная до последней запятой;
6) выводится подстрока, расположенная после первого двоеточия;
7) выводится первое и последнее слова строки;
8) подсчитывается количество слов в строке;
9) подсчитывается количество слов в строке, начинающихся с заданной буквы;
10) меняются местами соседние слова.

Контрольные вопросы

1 Перечислите способы создания объектов типа **string**.

2 Перечислите операции над строками.

3 Перечислите методы класса **String**.

4 В чем заключается различие между строковыми объектами классов **String** и **StringBuilder**?

5 Объясните назначение класса **StringBuilder**.

6 В каких случаях рекомендуется использовать строковые объекты класса **StringBuilder**? В чем заключается главная особенность строковых объектов класса **StringBuilder**?

7 Перечислите основные методы и свойства класса **StringBuilder**.

7 Лабораторная работа № 11. Разработка классов по индивидуальным вариантам. Работа с файлами

Цель работы: ознакомиться с основными принципами ООП; сформировать навыки создания класса, объекта, методов, конструкторов; сформировать навыки работы с файлами.

7.1 Краткие теоретические сведения

Каждый объект реального мира обладает свойствами и поведением – набором статических и динамических характеристик. Поведение объекта зависит от его состояния и внешних воздействий. Понятие объекта в программировании похоже на обыденный смысл этого слова. **Объект** – это совокупность данных, характеризующих его состояние, и методов, моделирующих его поведение.

В объектно-ориентированном программировании (ООП) предметная область представляется как совокупность взаимодействующих объектов. Реализуется **событийная модель** взаимодействия: объекты обмениваются сообщениями и, реагируя на них, выполняют определенные действия.

Основные **принципы ООП**: абстрагирование, инкапсуляция, полиморфизм, наследование.

Абстрагирование – выделение существенных для данной задачи характеристик объекта и отбрасывание второстепенных. Любой программный объект – это абстракция. Детали реализации объекта, как правило, скрыты, они используются через его интерфейс – совокупность правил доступа.

Инкапсуляция – сокрытие деталей реализации. Позволяет представить программу в укрупненном виде и защитить от нежелательных вмешательств.

Полиморфизм – использование одного имени (методов, операций, объектов) для решения нескольких схожих задач или для обращения к объектам разного типа. Идея полиморфизма – «один интерфейс, множество методов». Возможны различные способы реализации полиморфизма: перегрузка методов, перегрузка операций, виртуальные методы, переопределение методов, параметризованные классы. Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

Наследование – это процесс, посредством которого один объект может приобретать свойства другого. Для объекта можно определить потомков, которые наследуют, корректируют или дополняют его поведение. Наследование дает возможность многократного использования программного кода.

Создание класса и объекта. Методы. Конструкторы

Класс – обобщенное понятие, описывающее характеристики и поведение множества сходных объектов (называемых **экземплярами** или просто объектами этого класса). В программе класс является пользовательским **типом данных** и представляет собой блок кода, в котором описывается одна сущность,

например, модель реального объекта или процесса. Основными элементами класса являются данные и методы их обработки.

Заголовок описания класса обязательно содержит служебное слово **class** и **Имя**, которое по правилам языка C# начинается с заглавной буквы. В теле класса в фигурных скобках **{ ... }** описываются его элементы. Тело может быть пустым.

[модификаторы] **class Имя** [: предки]
{тело класса}

Перед заголовком класса могут указываться **модификаторы** (modifier), которые определяют некоторые общие свойства класса, а также доступность для других элементов программы: **internal** – внутренний (задается по умолчанию, можно не писать); **public** – общедоступный; **private** – закрытый; **protected** – защищенный (закрыт для посторонних, но доступен для наследников); **static** – статический; **abstract** – абстрактный; **sealed** – запечатанный (или бесплодный, не может иметь наследников).

Класс может содержать следующие функциональные элементы (*members*):

- **поля** – переменные класса;
- **свойства** – обеспечивают «умный» доступ к полям;
- **методы** – определяют поведение класса;
- **конструкторы** – служат для инициализации объектов (экземпляров класса);
- **исключения** – используются для обработки исключительных ситуаций;
- набор **операций**, позволяющих производить различные действия.

Простейший пример – описание общедоступного класса с одним методом:

```
public class Computer
{
    public void ShowInfo()
    { Console.WriteLine("RAM = 8 Гбайт");
    }
}
```

Все классы .NET имеют общего предка – класс **Object** – и организованы в единую иерархическую структуру. Классы логически группируются в **пространства имен**, которые служат для упорядочивания имен классов и предотвращения конфликтов имен: в разных пространствах имена могут совпадать. Пространства имен могут быть вложенными. Любая программа использует пространство имен **System**, в котором собрано множество стандартных классов и методов.

Рассмотрим теперь описание экземпляра класса. Напомним, что класс является обобщенным понятием, определяющим характеристики и поведение множества конкретных объектов, называемых **экземплярами** (или просто объектами) этого класса. Классы создаются программистом до выполнения программы. Экземпляры класса (объекты) создаются системой во время выпол-

нения. Программист задает создание экземпляра класса с помощью инструкции **new**, например: Computer comp1 = new Computer(“Asus”, 2048) или Computer comp2 = new Computer(“IBM”, 4096). При этом для каждого объекта выделяется отдельная область памяти для хранения его данных. Класс может содержать также и статические элементы, которые существуют в единственном экземпляре.

Содержащиеся в классе данные могут быть переменными или константами. Описанные в классе переменные называются **полями класса**. При описании полей обязательно указывают **тип** и **имя** (которое начинается с малой буквы):

[модификаторы] **тип имя** [= начальное_значение]

Можно также указывать модификаторы, определяющие доступность, а также задавать начальное значение (т. е. инициализировать поле).

Все поля в C# сначала автоматически инициализируются нулем соответствующего типа. Например, полям типа int присваивается 0, полям типа double – 0.0, а ссылкам на объекты – значение null. После этого полю присваивается значение, заданное при его явной инициализации.

Метод – именованный функциональный элемент класса, выполняющий вычисления и другие действия. Метод определяет поведение класса. В программе метод представляет собой блок кода, содержащий ряд инструкций, к которому можно обратиться по имени. Он описывается один раз, а вызываться может многократно по необходимости.

При описании метода в заголовке обязательно указывают его **тип** (который соответствует типу возвращаемого методом значения) и **Имя** (с заглавной буквы). В круглых скобках после имени указывают параметры метода, которых может и не быть, но скобки обязательны:

[модификаторы] **тип Имя(** [параметры] **)**
{ тело метода }

В теле метода в фигурных скобках **{ ... }** описываются выполняемые этим методом действия. Тело метода может быть пустым. Доступность и другие характеристики метода задаются модификаторами (смысл которых будет раскрываться по мере выполнения работ): **public**, **private**, **protected**, **internal**, **abstract**, **virtual**, **override**, **new**, **static**, **sealed**.

Методы класса имеют доступ к его полям непосредственно или через свойства (**set** – **get**). Поскольку поля хранят данные, а методы выполняют действия, для облегчения чтения и понимания кода программы рекомендуется поля называть существительными, а методы глаголами.

Параметры метода определяют множество значений аргументов, которые можно передавать в метод. Для каждого параметра обязательно задавать его **тип** и **имя**. Передаваемые в метод аргументы должны соответствовать объявленным параметрам по количеству, типам и порядку. Имя метода вместе с

количеством и типами его параметров составляет **сигнатуру** метода. В сигнатуру не входит тип возвращаемого методом значения. Методы различают благодаря сигнатурам. В классе не должно быть методов с одинаковыми сигнатурами.

В языке C# возможны четыре типа параметров: параметры-значения, параметры-ссылки (ref), выходные параметры (out), параметры-массивы (params).

При вызове метода сначала выделяется память под его параметры. Каждому из параметров сопоставляется соответствующий аргумент. Проверяется соответствие типов аргументов и параметров, и производятся необходимые преобразования типов (или выдается сообщение о невозможности). После этого выполняются вычисления и/или другие действия (тело метода). В результате значение заявленного типа передается в точку вызова метода. Если методу задан тип **void**, он ничего не возвращает. После выполнения метода управление передается на выражение, следующее после его вызова.

Методы реализуют функционал класса. Хорошо спроектированный метод должен решать только одну задачу, а не все сразу. Необходимо четко представлять, какие параметры должен получать метод и какие результаты выдавать. Необходимо стремиться к максимальному сокращению области действия каждой переменной. Это упрощает отладку программы, поскольку ограничивает область поиска ошибки.

Заметим, что элементы (поля, методы), характеризующие класс в целом, следует описывать как **статические**. Статический метод (с модификатором *static*) может обращаться только к статическим полям класса. Статический метод вызывается через имя класса, а обычный – через имя экземпляра.

Конструктор – особый вид метода, предназначенный для инициализации объекта (конструктор экземпляра) или класса (статический конструктор). Конструктор объекта вызывается при создании экземпляра класса с помощью ключевого слова **new**. Имя конструктора **совпадает** с именем класса. Конструктор не имеет никакого типа, даже **void**.

Класс может иметь несколько конструкторов с разными параметрами для разных вариантов инициализации. Если не указано ни одного конструктора или некоторые поля не были инициализированы, полям значимых типов присваивается ноль (**0** или **0.0**), полям ссылочных типов – значение null. Конструктор, вызываемый без параметров, называется **конструктором по умолчанию**. Пример с использованием конструкторов представлен в следующих разделах (см. ***Инкапсуляция. Сокрытие полей, создание свойств***).

Перегрузка методов

Перегрузкой методов (*overloading*) называется использование методов с одним и тем же именем, но различным количеством и типами параметров.

Компилятор по типу фактических параметров сам определяет, какой именно метод требуется вызвать. Это называется **разрешением** (*resolution*) перегрузки.

Перегрузка методов – одна из простейших реализаций полиморфизма.

Широко используется также перегрузка конструкторов. Большинство стандартных операций тоже можно переопределять, что позволяет использовать объекты своих типов в составе выражений аналогично переменным стандартных типов.

Далее представлен пример класса с перегруженными методами:

```
using System;
class Program
{
    static void Perim(int a, int b) // два параметра
  {
  Console.WriteLine("Периметр прямоугольника = {0}", 2 * a + 2 * b);
    }
    static void Perim(int a, int b, int d) // три параметра
{
        Console.WriteLine("Периметр треугольника = {0}", a + b + d);
    }
   static void Perim(params int[] ar) // переменное число параметров
    {
      int p = 0; foreach (int x in ar) p += x;
    Console.WriteLine("Периметр {0}-угольника = {1}", ar.Length, p);
    }

    static void Main()
    {
        Perim(10, 20); Perim(3, 4, 5); Perim(2, 3, 4, 5, 6, 7, 9);
        Console.ReadKey();
    }
}
```

Инкапсуляция. Сокрытие полей, создание свойств

Для защиты от нежелательных вмешательств доступ к полям и методам классов приходится закрывать или ограничивать (например, с помощью модификаторов **private**, **protected**). Для организации управления доступом к полям класса служат **свойства** (properties). Как правило, свойство определяет методы доступа к закрытому полю и имеет следующий синтаксис:

[модификаторы] **тип Имя**
{
{ get код_доступа **}** // получение значения
{ set код_доступа **}** // установка значения
}

В C# свойство имеет то же имя, что и соответствующее скрытое поле, только первая буква заглавная, например: **поле** private int **age**, свойство public int **Age**. При обращении к свойству автоматически вызываются указанные в нем блоки чтения **get** и установки **set**. Может отсутствовать либо часть get, либо set,

но не обе одновременно. Если отсутствует set, свойство доступно только для чтения (read only), если отсутствует get – только для записи (write only).

Пример сокрытия полей и создания свойств представлен далее:

```
using System;
namespace ConsoleApp1
{
   class Student
    {
        public string fam; //поля сначала public
        public int kurs;
        public Student() { } // конструктор без параметров
        public Student(string fam, int kurs) // конструктор с
параметрами
        {
            this.fam = fam; this.kurs = kurs; // fam и kurs сначала
с малых букв
        }
        public void ShowInfo() // метод ShowInfo
        {
            Console.WriteLine("Студент {0} курса {1}", kurs, fam);
        }
    }
}
```

```
using System;
namespace ConsoleApp1
{

    class Program
    {
        static void Main()
        {
            Student st1 = new Student("Иванов", 3); st1.ShowInfo();
            Student st2 = new Student("", -7); st2.ShowInfo();
            Console.ReadKey();
        }
    }
}
```

Согласно примеру, в едином пространстве имен **ConsoleApp1** с шаблоном класса **Program** создан класс **Student**. В этом классе объявлены два поля: **fam** (фамилия) и **kurs** (курс). Созданы конструкторы и метод **ShowInfo()**, который выводит информацию о студенте.

Первоначальная общедоступность полей и методов класса **Student** позволяет при отладке программы в методе **Main()** класса **Program** создавать объекты класса **Student** (т. е. описывать конкретных студентов по шаблону класса **Student**), а также вызывать метод **ShowInfo()**.

Существенный недостаток кода в классе **Student** – незащищенность от ввода абсурдных данных (например, в классе **Program** можно ввести, что студент st2 не имеет фамилии и учится на отрицательном курсе –7).

Для устранения этого недостатка **инкапсулируем** данные (скроем и защитим поля), создав свойства с методами **set** и **get** для управления доступом к полям. Система Visual Studio позволяет автоматизировать этот процесс.

Устанавливаем курсор на **имя_поля** (например, **fam**), в меню **Рефакторинг** (Refactoring) выбираем пункт **Инкапсулировать поле** (Incapsulate field), в пункте **Обновление ссылок** указываем **Все** и нажимаем **OK**. В появившемся диалоговом окне **Просмотр изменений ссылок** показываются предлагаемые замены полей на свойства (как правило, это все методы и объекты, использующие значения полей, кроме конструкторов!). Соглашаемся с предложением, нажимая **Применить**. Доступ к полю **fam** будет изменен на **private** (закрытый) и методом **Fam** сгенерирован шаблон общедоступного **свойства** с именем **Fam** (с большой буквы), которое и будет использоваться теперь вместо поля **fam**.

Создадим свои правила доступа. Для этого будем вводить необходимый код в автоматически сгенерированные шаблоны **set** и **get**. Например, все фамилии будем хранить большими буквами (преобразование зададим в блоке **set**). Если фамилия не введена (поле **fam** пустое), то блок **get** будет возвращать значение «неизвестный». Измененный фрагмент кода будет выглядеть так:

```
private string fam; //поле private
public string Fam
        {
            get { return (fam != "") ? fam : "неизвестный"; }
            set { fam = value.ToUpper(); }
        }
```

Аналогичным образом инкапсулируем поле **kurs**. Защитим его от ввода и хранения абсурдных значений. Например, при вводе чисел <1 или >4 в поле **kurs** будет сохранено значение 0 (поступай на подготовительный курс!).

```
private int kurs;
public int Kurs
        {
            get { return kurs; }
            set { kurs = (value < 1 || value > 4) ? 0 : value; }
        }
```

На завершающем этапе заменим в конструкторе **имена скрываемых полей** (с малой буквы) на **имена открытых свойств** (с большой буквы):

```
public Student(string fam, int kurs)
{this.Fam = fam; this.Kurs = kurs;}
```

Автоматически такая замена в конструкторе не производится, поскольку окончательное решение о правах доступа должен принимать программист.

Протестируйте самостоятельно окончательный вариант кода с разными параметрами.

Наследование

Наследование (inheritance) – это процесс приобретения состояния и поведения одного класса (называемого **базовым** или **предком**) другим классом (называемым **производным**, **наследником** или **потомком**). Для любого класса, кроме бесплодного (модификатор **sealed)**, можно задать классы-наследники, в которых повторяется и дополняется состояние и поведение предка. Синтаксис объявления производного класса (потомка)

[модификаторы] **class Имя : класс-предок, интерфейсы…**
{ тело класса }

Класс в C# может иметь произвольное количество потомков и только один класс-предок. В то же время класс может наследоваться от произвольного количества интерфейсов. Наследование позволяет многократно использовать программный код, исключать из программы повторяющиеся фрагменты; упрощает модификацию программ и создание новых классов на основе существующих. Благодаря наследованию можно, например, использовать объекты, исходный код которых недоступен, но в поведение которых требуется внести изменения.

Наследование позволяет строить иерархии объектов. Они представляются в виде деревьев, в которых более общие объекты (предки) располагаются ближе к корню, а более специализированные (потомки) – на ветвях и листьях (рисунок 7.1).

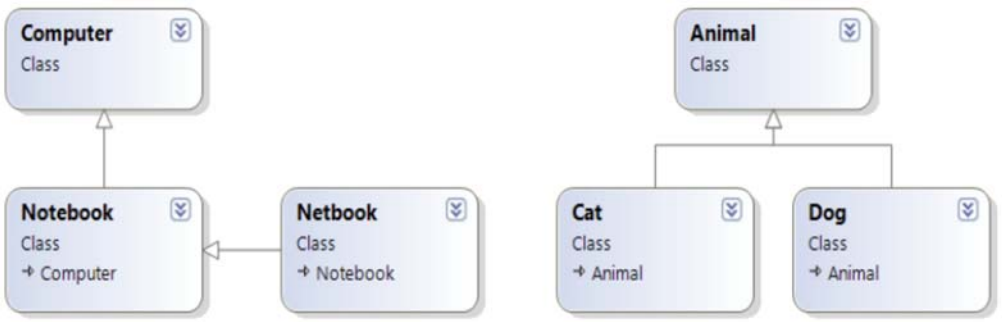


Рисунок 7.1 – Иерархии классов

Фрагменты программного кода для этих иерархий:

class **Computer { ... }** // базовый класс class **Animal { ... }**

class **Notebook : Computer** // потомки class **Cat : Animal**
{ ... } **{ ... }**
class **Netbook : Notebook** class **Dog : Animal**
{ ... } **{ ... }**

Возможны различные стратегии классического наследования:

– функционал потомка остается неизменным;

– функционал методов базового класса скрывается и замещается в производном классе (модификатор **new**);

– функционал методов базового класса, называемых **виртуальными** (модификатор **virtual**), переопределяется в производном классе (модификатор **override**).

Отметим важные особенности классического наследования:

– наследуются поля, методы и свойства класса;

– конструкторы не наследуются! Класс – потомок должен иметь собственные конструкторы;

– объекту базового класса можно присвоить объект производного, например:

```
public class Notebook : Computer { … }
        Computer comp = new Notebook( … );
```

Таким образом, методы, которые у потомков должны реализовываться поразному, при описании базовых классов следует определять виртуальными. Если во всех классах иерархии метод будет выполняться одинаково, его лучше определить как обычный метод. Виртуальные методы базового класса задают поведение всей иерархии, которое может изменяться и дополняться в потомках за счет добавления новых виртуальных методов. С помощью виртуальных методов реализуется один из основных принципов ООП – полиморфизм.

Абстрактные классы. Интерфейсы

Абстрактные (abstract) классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс задает общее поведение для всей иерархии, при этом его методы могут не выполнять никаких действий. Такие методы называются абстрактными. Они имеют пустое тело и объявляются с модификатором **abstract.**

Абстрактный метод – это виртуальный метод без реализации. Он должен быть переопределен в любом неабстрактном производном классе. Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть объявлен как абстрактный.

Абстрактный класс может содержать и полностью определенные методы (в отличие от интерфейса).

Абстрактный класс не разрешает создавать свои экземпляры. Он служит только для порождения потомков. Как правило, в нем лишь объявляются методы, которые каждый из потомков будет реализовывать по-своему. Это бывает полезным, если использовать переопределение методов базового класса в потомках затруднительно или неэффективно.

Интерфейс – крайний случай абстрактного класса, у которого нет полей и ни один метод не реализован. В нем объявляются только абстрактные методы,

которые должны быть реализованы в производных классах. Синтаксис интерфейса аналогичен синтаксису класса:

[модификаторы] **interface Имя** [**: предки**]
{ тело_интерфейса }

Тело интерфейса составляют только абстрактные методы, шаблоны свойств, а также события. Элементы интерфейса по умолчанию общедоступны. Интерфейс не может иметь обычных методов – все элементы интерфейса должны быть абстрактными.

В языке C# разрешено одиночное наследование для классов и множественное – для интерфейсов. Это позволяет придать производному классу свойства нескольких интерфейсов, реализуя их по-своему. Сигнатуры методов в интерфейсе и реализации должны полностью совпадать.

Интерфейс можно представлять как «контракт» о реализации объявленных методов потомками. Таким образом, наследование интерфейса заключается в его **реализации** (implementation) потомками. Основная идея использования интерфейса состоит в том, чтобы к объектам разных классов можно было обращаться одинаковым образом. Каждый класс может определять элементы интерфейса по-своему. Так реализуется полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода. Отметим, что в библиотеке классов **.NET** определено множество стандартных интерфейсов, декларирующих желаемое поведение объектов.

Работа с файлами

Обмен данными с устройствами в C# выполняется с помощью подсистемы ввода-вывода (**IO**) и классов библиотеки .NET. Реализуется с помощью потоков. **Поток** (*stream*) – абстрактное понятие, относящееся к любому переносу данных от источника к приемнику и наоборот. Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен. Потоки обеспечивают единообразие при работе со стандартными типами данных и с типами, определяемыми пользователем.

Обмен с потоком для повышения скорости передачи данных производится, как правило, через **буфер**, который выделяется для каждого открытого файла.

Чтение (ввод – input) – передача данных с внешнего устройства в оперативную память, обратный процесс – **запись** (вывод – output).

Для работы с потоками и файлами необходимо подключать пространство имен **System.IO.**

Выполнять обмен с внешними устройствами можно на уровне:

– двоичного представления данных – классы **BinaryReader**, **BinaryWriter**;
– байтов – класс **FileStream**;
– текста (строк) – классы **StreamWriter**, **StreamReader**.

Доступ к файлам может быть:

– последовательным – очередной элемент можно прочитать (записать)

только после предыдущего элемента;

– произвольным или прямым – выполняется чтение (запись) произвольного элемента по заданному адресу.

Прямой доступ при отсутствии дальнейших преобразований обеспечивает более высокую скорость получения нужной информации. В двоичных и байтовых потоках можно использовать оба метода доступа. Для текстовых файлов возможен только последовательный доступ.

Для открытия файла на запись текста создается поток – объект класса **StreamWriter**. Параметрами конструктора служат имя файла **imf** и режим записи (true – дозапись, false – перезапись):

StreamWriter **fw** = new StreamWriter(**imf**, true);

Отметим, что символы, которые можно ошибочно принять за управляющие, в имени файла необходимо экранировать, например, слешами "**D:\\testFile\\my.txt**", или все имя целиком с помощью символа «собака» **@**"**D:\testFile\my.txt**".

Для дальнейшей работы используется имя (дескриптор) созданного объекта **fw** и стандартный метод вывода строки, например:

fw.WriteLine("Запись в файл");

По завершении работы поток вывода закрывается **fw.Close()**.

Для открытия файла на чтение создается поток – объект класса **StreamReader**:

StreamReader **fr** = new StreamReader(**imf**);

Текст файла можно читать в строковую переменную **s** целиком (одной строкой):

string **s = fr.ReadToEnd();** или string s = **fr.ReadAllText(imf);**

а также построчно, и сразу выводить на консоль:

```
string s; int i = 0;
while ( ( s = fr.ReadLine() ) != null ) Console.WriteLine( "{0} : {1}", ++i, s);
```

Можно также читать строки файла в массив для дальнейшего вывода, например:

```
string[] stroki = fr.ReadAllLines(imf);
foreach (string s in stroki) Console.WriteLine(s);
```

Здесь важно отметить, что при чтении-записи могут возникать критические ошибки, поэтому следует обрабатывать исключения, помещая соответст-

вующие фрагменты кода в блок **try**, например:

```
try
{
 StreamReader fr = new StreamReader(imf);
 string s = fr.ReadToEnd(); Console.WriteLine(s);
 fr.Close();
}
catch (FileNotFoundException ex)
{
 Console.WriteLine(ex.Message);
 Console.WriteLine("Проверьте имя файла!"); return;
}
```

Пространство имен **System.IO** содержит классы **File**, **FileInfo**, **Directory**, **DirectoryInfo** для работы с файлами и каталогами (папками), например: создание, удаление, перемещение файлов и каталогов, получение свойств. Приведем примеры некоторых методов:

– **Create**, **CreateSubDirectory** – создать каталог (подкаталог) по указанному пути;

– **MoveTo** – переместить каталог и все его содержимое на новый адрес;

– **Delete** – удалить каталог со всем его содержимым;

– **GetFiles** – получить файлы текущего каталога как массив объектов **FileInfo**;

– **GetDirectories** – получить массив подкаталогов.

Далее представлен пример работы с текстовым файлом my.txt (запись-чтение строк).

```
using System;
using System.IO; // подключаем пространство имен System.IO
class Program
{
    static void Main()
    {
        string s1 = "Привет"; int a = 15; int b = 12;
        // создаем объект класса StreamWriter, открываем файл на
дозапись
        StreamWriter fw = new StreamWriter(@"D:\testFile\my.txt",
true);
        // записываем строки 1 и 2
        fw.WriteLine("1: Работа с файлом"); fw.WriteLine("2: " +
s1);
        // добавляем строки 3, 4, 5
        fw.Write("3: a = " + a); fw.WriteLine(", b = " + b); //
строка 3
        fw.WriteLine("4: 'a + b' = " + a + b); // строка 4
        fw.WriteLine("5: a + b = " + (a + b)); // строка 5
        fw.Close(); // закрываем записанный файл
```

```
                        // создаем объект класса StreamReader –
открываем файл на чтение
            StreamReader fr = new StreamReader("D:\\testFile\\my.txt");
            string str; int i = 0;
            while ((str = fr.ReadLine()) != null)
                Console.WriteLine("{0} - {1} ", ++i, str);
            fr.Close(); // закрываем прочитанный файл
            Console.ReadKey();
        }
    }
```

7.2 Практическое задание

7.2.1 Разработка индивидуальных классов.

1 Для заданной преподавателем предметной области разработайте классы. Продумайте их функциональность (т. е. методы). Придерживайтесь инкапсуляции. При необходимости реализуйте наследование классов, интерфейсов.

2 На основании разработанных классов создайте консольное приложение, в котором будут задействованы объекты данных классов. Приложение должно содержать программные конструкции, осуществляющие работу с файлами (чтение и запись).

Контрольные вопросы

1 Что такое класс и объект класса?
2 Какое отличие между описанием класса и объектом класса?
3 Что объявляется в классе?
4 Что называется полями класса?
5 Какая общая форма описания класса?
6 Какие существуют типы (модификаторы) доступа к членам класса?
7 Что такое статические члены класса?
8 Как используются статические и экземплярные методы?
9 Что представляет собой конструктор? Для чего он используется?
10 Какие бывают конструкторы?
11 Как создать объект?
12 Что такое инкапсуляция, наследование и полиморфизм?
13 Что такое перегрузка методов?
14 Опишите классы, предназначенные для работы с файлами. Как осуществить запись в файл и чтение из него?

8 Лабораторная работа № 12. Элементы форм Windows Forms – основные компоненты. Разработка приложений с формой

Цель работы: сформировать навыки создания приложений Windows Forms.

8.1 Краткие теоретические сведения

Система Microsoft Visual Studio содержит удобные средства визуальной разработки Windows-приложений, которые позволяют в интерактивном режиме конструировать программы с графическим интерфейсом, используя готовые компоненты и шаблоны. В процессе разработки выделяют два основных этапа:

1) **визуальное проектирование** – создание внешнего облика приложения, которое заключается в помещении на форму компонентов (элементов управления) и задании их свойств, а также свойств самой формы;

2) **программирование логики работы** приложения путем написания методов обработки событий.

Рассмотрим подробнее этапы разработки и структуру Windows-приложения.

1 После запуска **MS Visual Studio** выбирается тип **Приложение Windows Forms** (Windows Forms Application) и шаблон **Visual C#**. Задается имя и расположение проекта и решения, например **myWin.**

2 В результате открывается окно с формой в режиме конструктора (Design) (рисунок 8.1). Слева по умолчанию располагается **Панель элементов** (Toolbox), а справа – **Обозреватель решений** (Solution Explorer). При отсутствии их можно вызвать с помощью меню **Вид** (View).

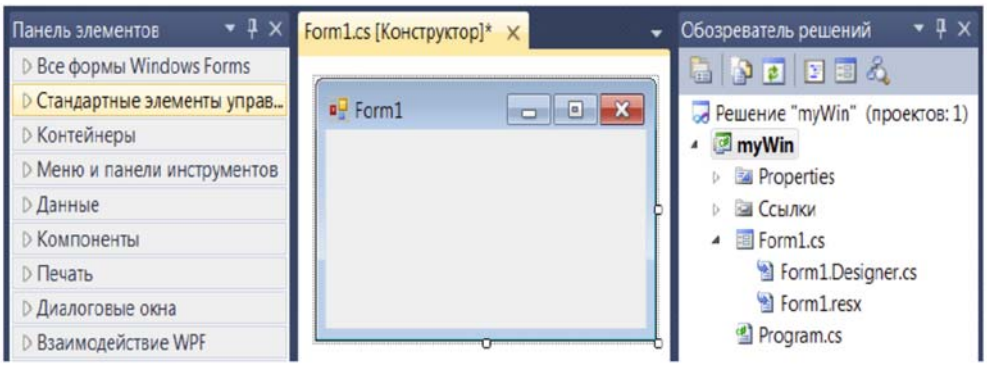


Рисунок 8.1 – Окно с формой в режиме конструктора

3 Требуемые элементы (например, кнопка) перетаскиваются мышью с **Панели элементов** на форму (рисунок 8.2). Корректируется их расположение и размеры. На панели **Свойства** задаются их характеристики (имя, внешний вид, поведение). Необходимые значения вводятся или выбираются из имеющихся в списке вариантов. Значок ▸ около имени свойства означает, что это свойство содержит другие, которые становятся доступными после щелчка на значке. При

размещении элемента на форме автоматически создается **экземпляр** соответствующего класса и шаблон программного кода.

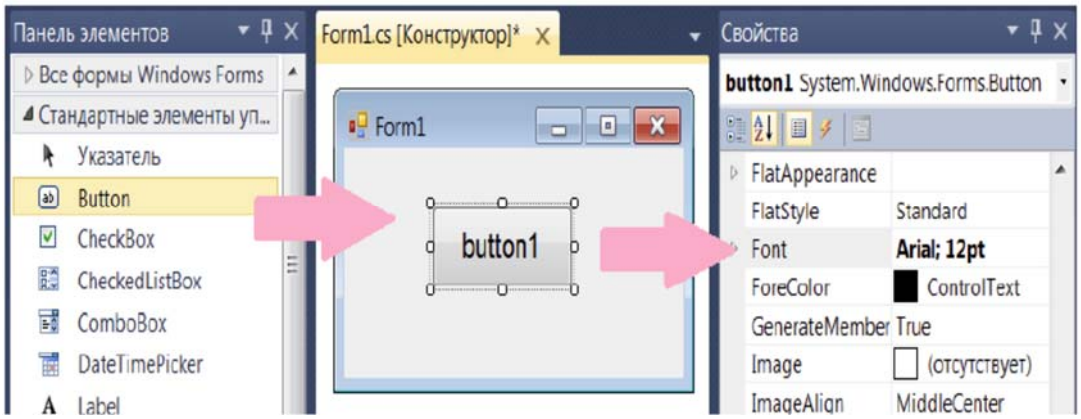


Рисунок 8.2 – Окно с формой в режиме конструктора

Проект Windows-приложения включает ряд файлов. Его структура отображается на панели **Обозреватель решений**.

4 Файл **Program.cs** содержит класс **Program**. Его метод **Main** является точкой входа, обеспечивает запуск приложения **Application.Run(new Form1())** и задает визуальный стиль (рисунок 8.3). Для других целей при визуальном проектировании Windows-приложения его обычно не используют.

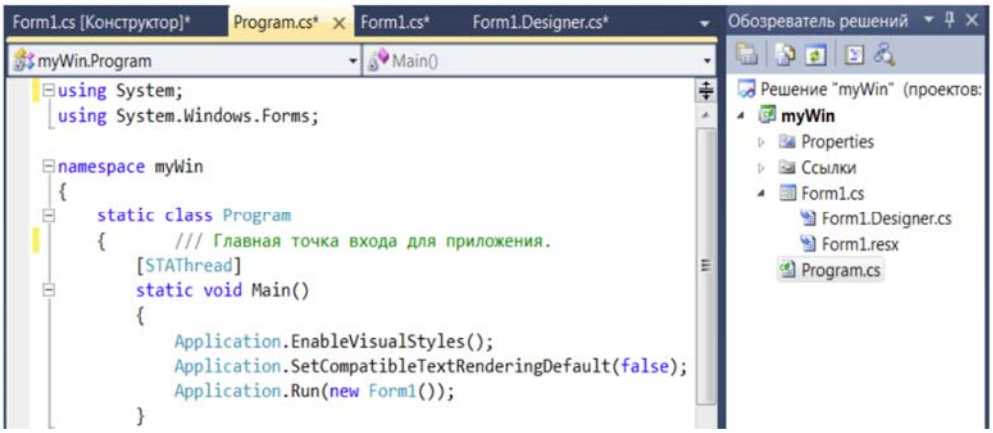


Рисунок 8.3 – Окно с кодом класса **Program**

Основной программный код с описанием используемых компонентов, объектов, методов, а также обработчиков событий находится в классе **Form1**, который для удобства программирования разделен на две части (модификатор **partial** – частичный).

5 Файл **Form1.cs** содержит часть класса **Form1** – конструктор с вызовом метода инициализации компонентов **InitializeComponent()** и обработчики событий (рисунок 8.4). Именно в обработчиках событий программируется логика работы приложения.

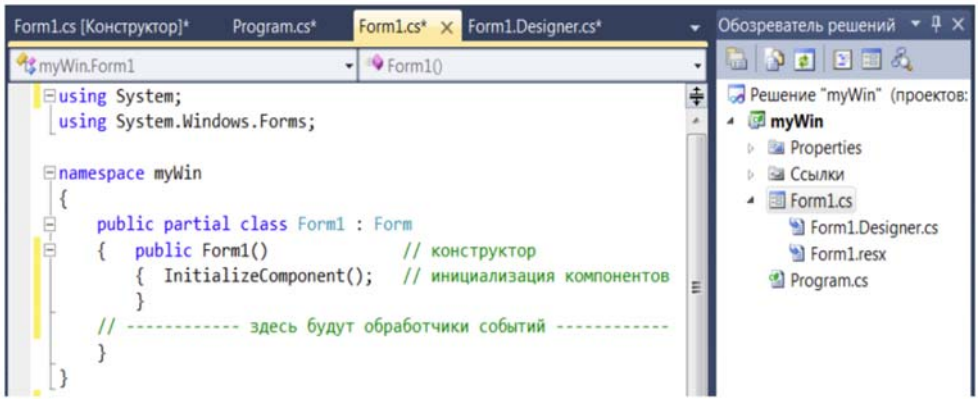


Рисунок 8.4 – Окно с программируемой частью кода класса **Form1**

6 Файл **Form1.Designer.cs** в области #region … #endregion содержит код метода **InitializeComponent()**, автоматически создаваемый конструктором форм при установке элементов и регистрации событий (рисунок 8.5).

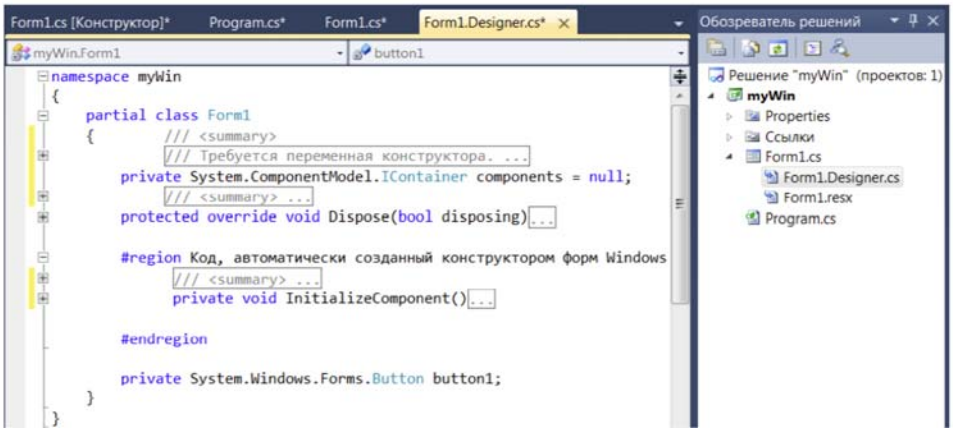


Рисунок 8.5 – Окно с автоматически создаваемым кодом класса **Form1**

Логика работы Windows-приложения основана на **объектно-событийной модели**. Определение поведения объектов начинается с принятия решений, какие действия должны выполняться при нажатии кнопки, вводе текста, перемещении курсора мыши, выборе пунктов меню, т. е. по каким событиям будут выполняться действия, реализующие функциональность программы. Для каждого класса определен свой набор событий, на которые он может реагировать.

Нужное событие для выбранного объекта сначала необходимо зарегистрировать в методе **InitializeComponent()** (файл **Form1.Designer.cs**) или даже непосредственно в конструкторе формы (файл **Form1.cs**), а затем запрограммировать ответные действия в обработчике этого события.

Регистрацию события (подписку на событие) выполняют на вкладке **События** (Events) панели **Свойств**а двойным щелчком мыши на поле, расположенном справа от имени соответствующего события (рисунок 8.6).

В методе **InitializeComponent()** (файл Form1.Designer.cs) появляется строка:\

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

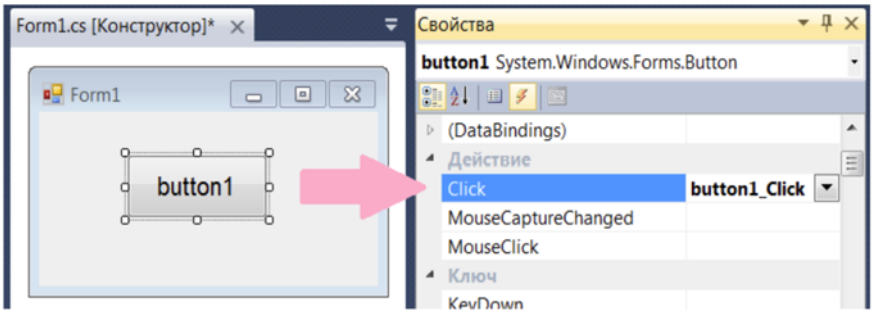


Рисунок 8.6 – Регистрация события нажатия кнопки Click

В файле Form1.cs автоматически создается шаблон соответствующего **метода-обработчика** (его имя формируется из имен объекта и события), в который предполагается вводить необходимый программный код. Обработчику передаются два параметра – объект-источник события и тип события:

```
private void button1_Click(object sender, EventArgs e)
{ ... }
```

Итак, разработка Windows-приложений в системе MS Visual Studio сводится к визуальному конструированию графического интерфейса в интерактивном режиме и программированию логики работы приложения путем написания методов обработки событий.

Библиотека классов .NET включает пространство имен **System.Windows.Forms**, содержащее огромное количество типов строительных блоков Windows-приложений. Приведем лишь наиболее часто используемые с указанием некоторых свойств и событий:

– **Form** – форма, служит контейнером, на который устанавливаются все элементы;

– **Button** – кнопка;

– **Label** – надпись (метка) служит для вывода текста;

– **TextBox** – поле ввода-вывода текста; свойство **Text**, событие **TextChanged**;

– **CheckBox** – флажок, включатель; свойство **Checked**, событие **CheckedChanged**;

– **RadioButton** – переключатель, выбор одного варианта из нескольких; свойство **Checked**, событие **CheckedChanged**;

– **PictureBox** – контейнер для изображений (**bmp**, **jpg**, **gif**, **png**…); свойства: **Image** – объект типа **Image**, **Visible** – видимость (true – false), **SizeMode** – позиционирование (Normal, StretchImage, Zoom, AutoSize).

Базовым классом (предком) для всех элементов управления является класс **Control**, который позволяет задавать общее поведение и свойства любого объекта графического интерфейса пользователя, например:

– **BackColor**, **ForeColor** – цвет фона и переднего плана;

– **BackgroundImage** – фоновый рисунок;

– **Text** – текст (строковые данные, ассоциированные с элементом);

– **Font** – шрифт;
– **Cursor** – вид курсора над элементом;
– **Enabled**, **Focused**, **Visible** – состояние элемента (true – false);
– **Opacity** – прозрачность элемента (0.0 прозрачный, 1.0 непрозрачный).

Основные **события** элементов управления:

– **Click**, **DoubleClick** – одинарный или двойной щелчки мышью;
– **MouseDown**, **MouseUp** – нажатие или отпускание кнопки мыши;
– **MouseMove** – перемещение мыши;
– **MouseHover** – мышь над элементом;
– **MouseEnter**, **MouseLeave** – мышь входит или покидает некоторую область;
– **KeyDown**, **KeyUp** – нажатие или отпускание любой клавиши;
– **KeyPress** – нажатие клавиши, имеющей ASCII-код;
– **DragDrop**, **DragEnter**, **DragLeave**, **DragOver** – события перетаскивания.

Положение, размеры и поведение элементов относительно контейнеров, в которые они вложены (например, **Form**, **Panel**, **GroupBox**, **PictureBox**), задаются свойствами **позиционирования**. Наиболее важные из них:

– **Location**, **Top**, **Left**, **Bottom**, **Right** – положение элемента;
– **Size**, **Width**, **Height** – размеры элемента;
– **Anchor** – привязка стороны элемента к сторонам контейнера. Если растягивать контейнер, то вместе с ним будет растягиваться и вложенный элемент (рисунок 8.7, *а*). По умолчанию это свойство равно Top, Left;
– **Dock** – прикрепление элемента к определенной стороне контейнера (рисунок 8.7, *б*). По умолчанию имеет значение None.

а) *б*)

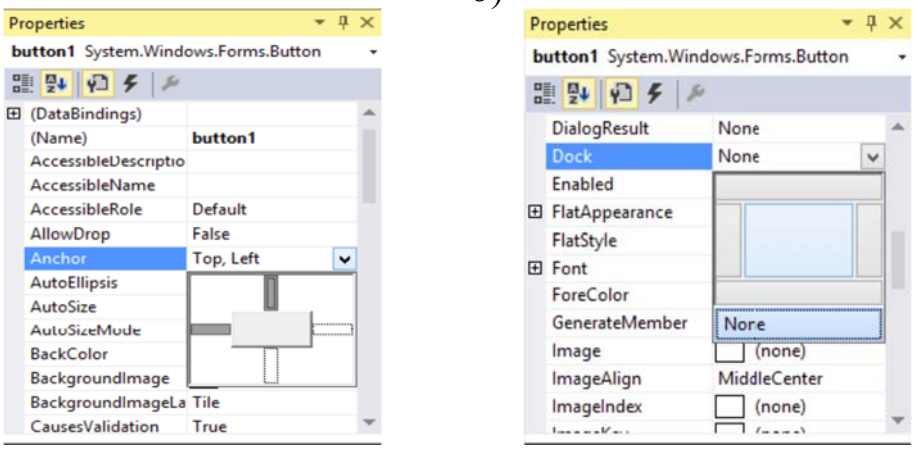


Рисунок 8.7 – Пример задания свойств Anchor (*а*) и Dock (*б*) для кнопки Button

В заключение раздела создадим Windows-приложение, которое приглашает ввести в элемент **TextBox** имя и по нажатию кнопки выводит в элемент **Label** приветствие. Для этого выполним следующие действия.

1 Запустим MS Visual Studio. Создадим новый проект.

2 Выберем тип приложения **Windows Forms** и шаблон **Visual C#**.

3 В поле ввода **Расположение (Location)** зададим рабочую папку, в которой будет сохраняться проект. Введем имя проекта, например **myWinProject**.

4 Откроется окно с формой **Form1** в режиме конструктора. По умолчанию слева располагается **Панель элементов** (Toolbox), а справа – **Обозреватель решений** (Solution Explorer).

5 Мышью перетащим с **Панели элементов** на форму две надписи **Label**, поле ввода текста **TextBox** и кнопку **Button**. При этом будут созданы **экземпляры** объектов, которым по умолчанию присваиваются имена соответствующих классов (с малой буквы) с номерами: **label1**, **label2**, **textBox1**, **button1**.

6 По очереди выделяем установленные элементы и на панели **Свойства** изменяем предлагаемые по умолчанию значения свойства **Text**: у формы на «**Приветствие»**, у надписи_1 на «**Введите имя»**, у кнопки на «**Нажмите»** (рисунок 8.8). Подберем размеры шрифта (свойства **Font** 10–12 пт).

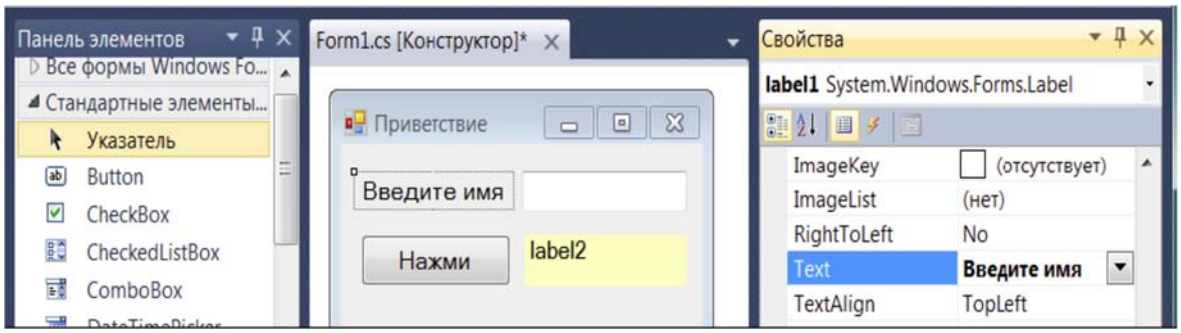


Рисунок 8.8 – Задание свойств элементов формы

7 Зарегистрируем событие нажатия кнопки. Для этого выделим кнопку и на вкладке **События** (**Events**) панели свойств выберем **Click**. Отметим, что регистрировать события, связанные с элементами по умолчанию, можно и двойным щелчком мыши по выбранному элементу.

8 Откроется окно **Form1.cs**, в котором автоматически будет создан шаблон обработчика. Введем в него код:

```
private void button1_Click(object sender, EventArgs e)
{ label2.Text = "Привет, " + textBox1.Text;
}
```

9 Протестируем программу (рисунок 8.9, *а*). Откорректируем программный код и свойства элементов (например, зададим желтый фон **BackColor** надписи_2).

10 Модифицируем наш проект. Разместим на форме элемент **pictureBox1** (рисунок 8.9, *б*). Импортируем изображение (свойство **Image**) из файла **x.jpg**.

Подберем размеры и установим значения свойств **Visible = false** (в исходном состоянии изображение невидимо) и **SizeMode = StretchImage** (растягивается по размеру контейнера).

а) *б*) *в*)

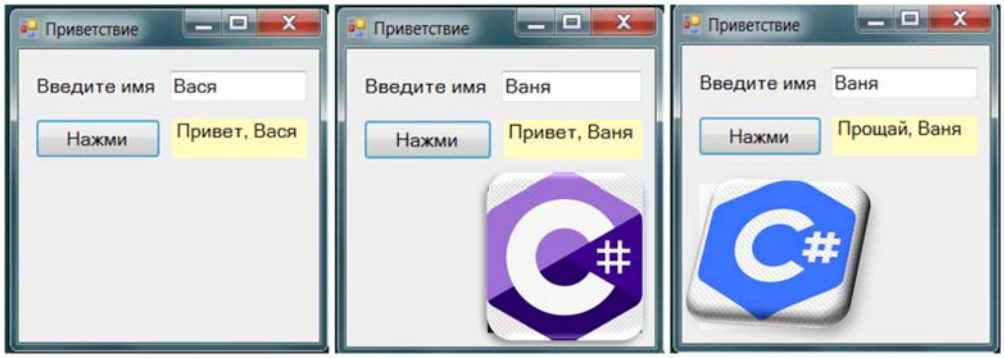


Рисунок 8.9 – Интерфейс приложения

11 В обработчик события кнопки добавим код: `pictureBox1.Visible = true;`

12 Протестируем программу. Теперь при вводе имени и нажатии кнопки появляется приветствие и изображение.

13 Еще раз модифицируем проект так, чтобы по щелчку мыши на **pictureBox1** это изображение исчезало, а появлялось новое из файла **xx.gif** и надпись «**Привет,**» заменялась на «**Прощай,**» (рисунок 8.9, *в*).

14 Разместим на форме элемент **pictureBox2** и импортируем в него изображение **xx.gif**. Настроим размеры и свойства.

15 Зарегистрируем событие **MouseClick** (щелчок мыши) на **pictureBox1**.

В созданный шаблон обработчика введем код:

```
private void pictureBox1_MouseClick(object sender, MouseEventArgs e)
{ label2.Text = "Прощай, " + textBox1.Text;
pictureBox1.Visible = false;
pictureBox2.Visible = true;
}
```

16 Протестируем программу.

8.2 Практическое задание

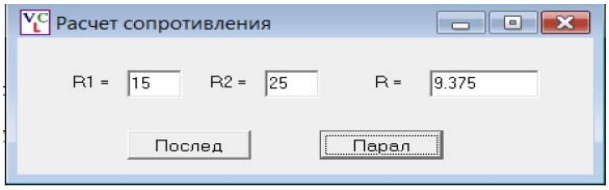
8.2.1 Разработка Window-приложений.

1 Разработайте Windows-приложение для вычисления сопротивления при последовательном или параллельном соединении резисторов. Ввод и вывод в текстовые поля по нажатию кнопок (рисунок 8.10, *а*).

2 Создайте калькулятор, выполняющий пять действий. Ввод и вывод в текстовые поля. Счет по нажатию кнопок (рисунок 8.10, б).

3 Разработайте приложение в соответствии с индивидуальным вариантом, выданным преподавателем.

а)



б)

Рисунок 8.10 – Шаблон интерфейсов приложений

Контрольные вопросы

1 Что понимают под термином «событийно-управляемое программирование»?

2 Назовите основные структурные элементы Windows-приложения.

3 Опишите процесс создания Windows-приложения с графическим интерфейсом на основе Windows Forms.

4 Что такое обработчик события?

5 Как создаются обработчики событий?

6 Опишите синтаксис обработчика событий.

7 Какие параметры передаются в обработчик событий?

8 Перечислите основные свойства стандартных визуальных элементов управления.

9 Каким образом можно добавить элемент управления на форму?

10 Объясните назначение класса Form.

11 Назовите основные свойства, методы, события класса Form.

Список литературы

1 Основы алгоритмизации и программирования на языке C# : учеб. пособие для вузов / Е. В. Кудрина [и др.]. – М. : Юрайт, 2024. – 322 с.

2 **Тузовский, А. Ф.** Объектно-ориентированное программирование : учеб. пособие для вузов / А. Ф. Тузовский. – М. : Юрайт, 2024. – 213 с.

3 **Гуриков, С. Р**. Введение в программирование на языке Visual C# : учеб. пособие / С. Р. Гуриков. – М.: ФОРУМ; ИНФРА-М, 2020. – 447 с.

4 **Мартыненко, Т. В.** Основы визуального программирования в среде Visual Studio на базе C# : учеб. пособие / Т. В. Мартыненко, В. В. Турупалов, Н. К. Андриевская; под общ. ред. В. В. Турупалова. – М. ; Вологда : Инфра-Инженерия, 2023. – 232 с.

5 **Заборовский, Г. А.** Программирование на языке C# : учеб.-метод. пособие / Г. А. Заборовский. – Минск: БНТУ, 2020. – 84 с.