МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Белорусский национальный технический университет

Кафедра программного обеспечения
информационных систем и технологий

Г. А. Заборовский
В. В. Сидорик

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C#

Учебно-методическое пособие
для студентов и слушателей системы повышения квалификации
и переподготовки, преподавателей

*Рекомендовано учебно-методическим объединением
в области автоматизации технологических процессов,
производств и управления*

Минск
БНТУ
2020

УДК 004.438(075.8)
ББК 32.973.2-018.1я7
З-12

Рецензенты:
доцент кафедры «Информатика» Белорусского государственного университета информатики и радиоэлектроники, канд. физ.-мат. наук *В. А. Ганжа*;
доцент кафедры «Информационные технологии в образовании» ГУО Белорусский государственный педагогический университет им. М. Танка, канд. физ.-мат. наук *С. И. Чубаров*

Заборовский, Г. А.
З-12 Программирование на языке С# : учебно-методическое пособие / Г. А. Заборовский, В. В. Сидорик. – Минск, БНТУ, 2020. – 84 с.
ISBN 978-985-583-074-1.

Данное учебное пособие содержит теоретические сведения и методические указания по выполнению лабораторных работ на языке С# в среде Microsoft Visual Studio по дисциплинам «Языки программирования», «Основы алгоритмизации и программирования», «Объектно-ориентированное программирование». Приводятся основные сведения о языке С#, особенностях объектно-ориентированного программирования консольных и Windows-приложений. Затрагиваются вопросы визуального проектирования классов. Описание каждой лабораторной работы включает краткое теоретическое введение, примеры выполнения типовых заданий, задания для самостоятельной работы.

Предназначается студентам специальностей 1-40 01 01 «Программное обеспечение информационных технологий», 1-40 05 01-01 Информационные системы и технологии в проектировании и производстве», 1-40 05 01-04 «Информационные системы и технологии в обработке и представлении информации», а также слушателям системы повышения квалификации и переподготовки преподавателей.

УДК 004.438
ББК 32.973.2-018.1

ISBN 978-985-583-074-1

© Заборовский Г. А., Сидорик В. В., 2020
© Белорусский национальный технический университет, 2020

ВВЕДЕНИЕ

Объектно-ориентированное программирование (ООП) является одним из самых востребованных, но в то же время достаточно трудных для изучения. Формирование профессиональных компетенций будущего ИТ-специалиста в области ООП весьма актуально в условиях быстрого развития информационных технологий, смены парадигм и технологий программирования.

Данное учебное пособие нацелено на самостоятельную работу студентов и содержит теоретические сведения и методические указания по выполнению лабораторных работ по основам программирования на языке C#, сгруппированные в 3 раздела.

Работы первого раздела направлены на изучение основных понятий, объектов и методов языка C# путем реализации базовых алгоритмов на примерах программирования консольных приложений с использованием стандартных классов .NET. Второй раздел посвящен созданию собственных классов и практической реализации принципов ООП: инкапсуляции, наследования, полиморфизма. Более подробно, чем обычно, рассматриваются вопросы автоматической инкапсуляции с помощью инструментов рефакторинга, а также проектирования классов с помощью визуального конструктора. В третьем разделе на практических примерах изучается разработка Windows-приложений с графическим интерфейсом. Приводятся примеры использование возможностей среды Microsoft Visual Studio для разработки простых практико-ориентированных приложений.

Описание каждой лабораторной работы включает ее цель, краткое теоретическое введение и подробную пошаговую инструкцию с примерами выполнения типовых заданий. Завершается каждая работа заданиями разного уровня сложности для самостоятельной работы. Особое внимание уделяется применению технологий ООП при решении практических задач в контексте будущей деятельности ИТ-специалиста.

Перед выполнением каждой работы необходимо изучать используемые в ней теоретические положения. Основы ООП на языке C# доступно изложены в учебных пособиях [1–3], справочных материалах [4]. Для углубленного изучения рекомендуются книги [5–8].

1. ИСПОЛЬЗОВАНИЕ СТАНДАРТНЫХ ОБЪЕКТОВ И МЕТОДОВ C#

Любая, даже самая простая программа на языке C# не может быть написана без использования базовых понятий объектно-ориентированного программирования, и, прежде всего, таких, как класс и метод. Приведем некоторые необходимые для начала работы понятия и термины без пояснений (более подробно они рассматриваются в соответствующих работах).

Рассмотрим структуру программы на языке C# на простейшем примере.

```
using System;                          // подклюючение 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# (как и Pascal) 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.

Для успешного выполнения работ первого раздела необходимо самостоятельно познакомиться с основными понятиями и синтаксисом языка C#, встроенными типами данных и операциями над ними, методами ввода-вывода.

1.1. Основы работы в среде Microsoft Visual Studio

Цель работы: знакомство с языком программирования C# и системой MS Visual Studio.

Введение

Консоль – совокупность стандартных устройств ввода-вывода (клавиатура, монитор). Для работы с консолью предназначен класс **Console** в прост-

ранстве 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 резделителем 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:
```
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} \nd = {d} y = {y}");
```

Для совместимости с более старыми версиями во всех примерах данного пособия используется форматный вывод.

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 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
Console.WriteLine("строка st={0} число x={1}  y={2}", st, x, y); // вывод
```

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

Пример 1

Простейшее приложение: вывод на консоль приветствия.

1) Запустим **MS Visual Studio**.

2) На появившейся стартовой странице выберем **Создать проект** (New Project).

3) На следующей странице выберем тип приложения **Консольное** (Console Application) и шаблон **Visual C#**.

4) В поле ввода **Расположение** (Location) выберем или зададим рабочую папку, в которой будет сохраняться проект, например, **\rabota**.

5) Введем имя проекта и **Решения** (Solution), например, **con01.**

6) 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 нашем примере используется пространство имен **System**). С помощью ключевого слова **namespace** создано собственное пространство имен, имя которого совпадает с именем проекта **con01**.

7) Для упрощения шаблона можно сразу удалить «лишние» строки и выражения, которые не будут использоваться в нашей программе, например, неиспользуемые параметры метода **Main()** и пространства имен.

8) В теле метода **Main** шаблона наберем код программы:

```
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();
    }
  }
```

9) 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.

10) Протестируем программу. В окне вывода получим:

```
Введите имя:  Маша
Привет, Маша!
```

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

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

12) Протестируем окончательный вариант. Теперь без ввода имени получим:

```
Введите имя:
Привет, неизвестный!
```

Примечание. Свойства окна вывода (цвет фона, шрифт…) можно настраивать.

Пример 2

Приложение, которое по введенному радиусу вычисляет и выводит на консоль длину окружности и площадь круга

1) Создадим проект с именем **con02**.

2) В теле метода **Main** наберем код программы:

```
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();
 }
```

3) Протестируем программу (*F5*). При необходимости откорректируем программный код. Результат может выглядеть так:

```
Введите радиус:   4
Длина окружности  25,133 , площадь круга  50,265
```

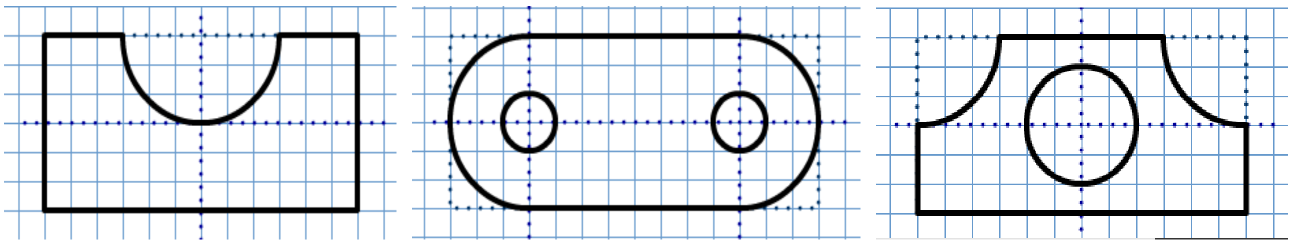
Задания для самостоятельной работы

Создайте консольные приложения, которые вычисляют и выводят:

1. Среднее арифметическое *sred* двух введенных чисел *a* и *b*.
2. Площадь *s* и периметр *p* прямоугольника по сторонам *a* и *b*.
3. Площадь поверхности $s = 4\pi r^2$ и объем шара $v = 4/3\ \pi r^3$ по радиусу *r*.
4. Значение функции $z = x^2 + x\,y - y^2$ (ввод *x* и *y*).
5. Стоимость товара в трех валютах по его стоимости в бел. рублях (курсы вводятся с клавиатуры).
6. Стоимость поездки на автомобиле (ввод: *s* – расстояние, *b* – расход бензина на 100 км, *c* - цена бензина за 1 литр).

7. Смещение от положения равновесия точки, совершающей гармонические колебания x = A sin(w, t) (ввод: амплитуда A, угловая частота w, время t).

8. Расход материала s для изготовления детали по чертежу. Масштаб: 1 клетка = 4 см.



1.2. Алгоритмическая конструкция «ветвление»

Цель работы: формирование навыков реализации алгоритмов ветвления.

Введение

Алгоритмическая конструкция **ветвление** используется для выбора одной из двух (развилка) или нескольких альтернатив (переключатель).

Развилка реализуется с помощью условного оператора **if:**

if (условие) { блок_1 }
[**else** { блок_2 }]

Сначала проверяется условие. Если оно верно, выполняется блок_1, иначе выполняется блок_2.

В условиях используют операции сравнения, которые в языке C# записываются так: = = равно, **!=** не равно, < , >, <=, >= ; а также логические операции && “и”, || “или”, **!** отрицание “не”.

Блоки могут содержать фрагменты кода, заключенные в фигурные скобки, которые можно опустить, если фрагмент состоит из одного выражения.

В случаях выбора и присваивания только одного значения из двух альтернативных вариантов вместо конструкции **if…else** удобно использовать тернарный оператор условного присваивания (**= ? :**):

= (условие) ? вариант1 : вариант2 ;

При необходимости выбора из нескольких вариантов используется конструкция **switch ...case** (оператор выбора или переключатель)**:**

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

При отладке программ часто используют случайные числа. Методы их генерации описаны в классе **System.Random**. Для получения псевдослучайной

последовательности чисел сначала создается экземпляр класса **Random**, затем методом **Next(n1,n2)** генерируется число в диапазоне [n1, n2], например:

```
Random rnd = new Random();  int n = rnd.Next(-20, 70);
```

Пример 1

*Использование конструкции **if … else** для проверки знака случайного числа и тернарного оператора для проверки четности.*

1. Создадим проект **con111**. В теле метода **Main** наберем код программы:

```
string mes1 = "", mes2 = "";          // объявление и инициализация
Random rnd = new Random();         // создаем экземпляр класса Random
int n = rnd.Next(-40, 40);        // генерируем случайное число от –40 до 40
if (n>=0) mes1 ="положительное";          // используем конструкцию if…else
    else mes1 = "отрицательное";
                                        // используем тернарный оператор
string mes2 = (n % 2 == 0) ? "четное" : "нечетное";
Console.WriteLine("Число {0} {1} {2}", n, mes1, mes2);
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 2

*Использование конструкции выбора **switch…case**.*

1) Создадим проект **con112**. Наберем код программы в теле метода **Main**:

```
string mes;
Console.WriteLine("Введите день недели: Пн, Вт, Ср, Чт, Пт, Сб, Вс");
string day = Console.ReadLine();
  switch (day)
   {  case "Сб": mes = "Иду в гости"; break;
      case "Вс": mes = "Отдыхаю"; break;
      default:   mes = "Работаю"; break;
   }
Console.WriteLine(mes);
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 3

Проверка логина и пароля, введенных с клавиатуры.

Пусть эталонные логин длиной не меньше 6 символов и пароль хранятся в строковых переменных **myLog** и **myPas** в коде программы. Сначала проверяем длину и совпадение введенного логина с заданным. В случае правильности логина проверяем совпадение пароля.

1. Создадим проект **con113**. Наберем код программы в теле метода **Main**:

```
    // задаем эталонные логин и пароль, а также выводимые сообщения
string  myLog = "qwerty",  myPas = "asdf";
string  mes = "", mesYes = "Добро пожаловать!",  mesNo = "Вход воспрещен!";
Console.Write("Введите логин:   ");
string log = Console.ReadLine();                              // вводим логин
    // проверяем длину и совпадение логина
if (log.Length < 6)  mes = "Логин очень короткий!\n" + mesNo;
   else if (log != myLog)  mes = "Неверный логин!\n" + mesNo;
     else
    // в случае истинности логина вводим и проверяем пароль
     { Console.Write("Введите пароль:   ");
       string pas = Console.ReadLine();                       // вводим пароль

       mes = (pas == myPas) ? mesYes : "Неверный пароль!\n" + mesNo;
     }
Console.WriteLine(mes);                                  // выводим сообщения
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 4

Простейший калькулятор на 4 действия.

1. Создадим проект **con114**. Наберем код программы в теле метода **Main**:

```
//  инициализируем переменные res и ok.
double A, B, res = 0;  bool ok = true;
Console.Write("Введите число A:  ");
A = double.Parse(Console.ReadLine());       // ввод строки и преобразование
Console.Write("Введите число B:  ");
B = double.Parse(Console.ReadLine());
Console.Write("Введите знак операции (+-*/)  ");
string op = Console.ReadLine();
  switch (op)
  {  case "+": res = A + B; break;
     case "-": res = A - B; break;
     case "*": res = A * B; break;
     case "/": res = A / B; break;
     default: ok = false; break;}
if (ok)
    Console.WriteLine("{0} {1} {2} = {3}", A, op, B, res); // вывод
else Console.WriteLine("Недопустимая операция");
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

В рассмотренном примере простейшего калькулятора 4 полезно производить проверку вводимых данных и, если введено не число, присваивать значение по умолчанию, например 1.

Фрагмент кода с проверкой и преобразованием строки в число примет вид:

```
if (!double.TryParse(Console.ReadLine(), out A))
    {  Console.WriteLine("Не число!");  A = 1;  }
if (!double.TryParse(Console.ReadLine(), out B))
    {  Console.WriteLine("Не число!");  B = 1;  }
```

3. Протестируем окончательный вариант.

Задания для самостоятельной работы

Создайте консольные приложения, в которых выполняются заданные действия:

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

1.3. Алгоритмическая конструкция «цикл»

Цель работы: формирование навыков реализации циклических алгоритмов.

Введение

При необходимости многократного использования фрагмента программного кода используют алгоритмическую конструкцию **цикл** (повторение).

Возможны несколько вариантов реализации циклов:

– **с предусловием** – с помощью оператора **while;**

– **с постусловием** – с помощью оператора **do … while;**

– **с параметром** – с помощью оператора **for;**

– **перебора** элементов массива – с помощью оператора **foreach … in.**

При реализации циклов часто используют следующие операции:

++ инкремент, –– декремент (увеличение и уменьшение на 1); присваивание с операциями +=, – =, *=, /=. Например, k++ означает k = k + 1, а вместо z = z – 5 пишут z – = 5 (без пробела между – и =).

Используются также операторы передачи управления:

– **break** прерывает выполнение цикла и инициирует выход из блока;

– **continue** выполняет переход к следующей итерации внутри цикла;

– **return** завершает выполнение функции и передает управление в точку ее вызова;

– **goto** выполняет безусловную передачу управления.

Пример 1

Использование цикла for для вычисления суммы и произведения ***n*** *натуральных чисел.*

1. Создадим проект **con121**. Наберем код программы в теле метода **Main**:

```
Console.Write("Введите целое число (от 3 до 9)   ");
            // ввод строки, преобразование в int
int n = int.Parse(Console.ReadLine());
            // задание начальных значений sum и pro перед телом цикла
int sum = 0;   int pro = 1;
for (int i=1; i<=n; i++)
 { sum += i;  pro *= i;
            // тело цикла:  вычисления и вывод
   Console.WriteLine("шаг={0} сумма = {1} произведение = {2}", i, sum, pro);
 }
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 2

Использование цикла while. За один день уровень радиации уменьшается на 5 %. За сколько дней он уменьшится вдвое?

1. Создадим проект **con122**. Наберем код программы в теле метода **Main**:

```
Console.Write("Введите начальный уровень радиации(100 – 800 единиц:");
double rad0 = double.Parse(Console.ReadLine());
    // задание начальных значений  rad и day перед телом цикла
double rad=rad0; int day=0;
while (rad>rad0/2)              // проверка условия уменьшения радиации вдвое
 { rad *= 0.95; day++;          // ежедневное уменьшение радиации на 5 %
   Console.WriteLine("День {0}, радиация = {1}", day, rad);
 }
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 3

Использование циклов ***while*** *и* ***do…while*** *для моделирования движения тела. Тело брошено вертикально вверх с начальной скоростью* ***Vo***. *Найти максимальную высоту подъема и время полета.*

Будем считать тело материальной точкой, не учитывать сопротивление воздуха и пренебрегать изменением ускорения свободного падения с высотой. Выберем начало вертикальной оси координат 0Y в точке бросания тела. Свяжем начало отсчета времени с моментом бросания.

Уравнения движения в проекции на выбранную ось имеют вид:

$$\mathbf{y = Vo\ t - g\ t^2/2; \quad V = Vo - g\ t.}$$

Алгоритм решения задачи заключается в использовании исходных уравнений движения в циклах с достаточно малым шагом изменения времени, например, t += 0.001. Максимальная высота подъема определяется в цикле while при условии, что скорость V >= 0, а время полета в цикле do … while, который выполняется до тех пор, пока координата y > 0.

1. Создадим проект **con123**. Наберем код программы в теле метода **Main**:

```
Console.Write("Введите начальную скорость Vo (от 12 до 40) м/с :  ");
double Vo = double.Parse(Console.ReadLine());               // ввод и преобразование
double t = 0, y = 0, V = Vo, g = 9.81 ;               // задание начальных условий
        //  расчет максимальной высоты подъема тела в цикле с предусловием
while (V >= 0)
 {
   V = Vo - g*t;  y = Vo*t - g*t*t/2;  t += 0.001;
 }
Console.WriteLine("время = {0:f2} с,  макс высота = {1:f2} м", t, y);
//  расчет времени  полета тела в цикле с постусловием
do {
     t += 0.001;  y = Vo*t - g*t*t/2;
   }
while (y > 0);
Console.WriteLine("время полета = {0:f2} с", t);
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Многие задачи требуют многократного ввода данных. Для этого можно использовать бесконечный цикл **while(true)**, выход из которого выполняется в результате проверки вводимых данных оператором **if** в теле цикла, или цикл **do…while**() с проверкой в его постусловии. Сравним эти способы на примере ввода и суммирования чисел. Для завершения вводится буква «Q».

Пример 4

Реализация многократного ввода: бесконечный цикл ***while****.*

1. Создадим проект **con124**. Наберем код программы в теле метода **Main**:

```
    // объявляем и инициализируем переменные
double n = 0, sum = 0; string s = "";
while (true)                                       // бесконечный цикл while
 {
```

```
  Console.Write("Введите число: ");
  s = Console.ReadLine();                         // ввод
  if (s == "Q") break;                            // выход из цикла по условию
  if (double.TryParse(s, out n))                  // преобразование в double
    { sum += n;                                   // суммирование чисел
      Console.WriteLine("сумма = " + sum);  }
}
```

2. Протестируем программу. Откорректируем программный код.

Пример 5

Реализация многократного ввода: цикл ***do…while****.*

1. Создадим проект **con125**. Наберем код программы в теле методa **Main**:

```
          // объявляем и инициализируем переменные
double n = 0, sum = 0; string s = "";
do
 {
   Console.Write("Введите число: ");
   s = Console.ReadLine();                        // ввод
   if (double.TryParse(s, out n))                 // преобразование в double
      {  sum += n;                                // суммирование чисел
       Console.WriteLine("сумма = " + sum);   }
  // повторяем ввод до тех пор, пока не введен символ «Q»
  } while (s!="Q");                               // выход из цикла по условию
```

2. Протестируем программу. Откорректируем программный код.

Пример 6

Вклады в банках с простыми и сложными процентами (цикл ***while****).*

1. Создадим проект **con126**. Наберем код программы в теле метода **Main**:

```
Console.Write("Введите начальный вклад (от 100 до 500 руб:  ");
double vklad = double.Parse(Console.ReadLine());
Console.Write("Введите процентную ставку (от 10 до 30):  ");
double proc = double.Parse(Console.ReadLine());
       // задаем начальные значения
double sum1=vklad, sum2=vklad;  int god=0;
while (sum1 < 2*vklad)                          // условие - удвоения суммы
 {     // выполняем вычисления накопленных сумм и задаем приращение года
  sum1 += proc*vklad/100; sum2 *= (1+proc/100);  god++;
  Console.WriteLine("год {0}, банк1 = {1}, банк2 = {2}", god, sum1, sum2);
 }
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Задания для самостоятельной работы

Создайте консольные приложения, в которых выполняются заданные действия:

1. Вычисляется сумма всех нечетных чисел от ***n*1** до ***n*2** (ввод ***n*1**, ***n*2**, цикл for).
2. Вычисляется сумма квадратов **n** натуральных чисел, начиная с **k** (ввод **k** и **n**, цикл for).
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 (for, if).
5. Генерируется 10 случайных чисел в интервале (– 20, 20). Выводятся только положительные числа и сообщения: четное – нечетное (for, if).
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 (while, if).
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 (цикл while).
8. Ежедневно количество бактерий увеличивается на *p* %. Через сколько дней количество бактерий увеличится в n раз (ввод *p* и *n*).

1.4. Работа с массивами

Цель работы: формирование навыков работы с одномерными массивами в C#.

Введение

Массив – именованная упорядоченная последовательность однотипных элементов, к каждому из которых можно обратиться по индексу. Для обращения к элементу массива после его имени указывается индекс элемента в квадратных скобках: w[4], z[i]. Во многих случаях индекс можно считать порядковым номером элемента. В C# начальный индекс = 0 (элементы нумеруются с нуля).

Виды массивов в C#: одномерные, многомерные (например, двумерные, прямоугольные), массивы массивов (jagged – используются термины: ступенчатые, изрезанные и др.).

Массив относится к ссылочным типам данных (располагается в хипе), поэтому создание массива начинают с выделения памяти под его элементы. Элементами массива могут быть величины как значимых, так и ссылочных типов (в том числе массивы), например:

```
int[] w = new  int[10];                  // массив из 10 целых чисел;
string[] z = new  string[100];           // массив из 100 строк;
Car[] s = new  Car[5];                   // массив из 5 объектов типа Car;
double[,]  tb = new double[2, 10];       // прямоугольный массив 2 × 10;
int[][] a = new int[2][];                // массив массивов.
```

Элементы массивов значимых типов хранят значения, массивы ссылочных типов – ссылки на элементы. При объявлении массива его элементам присваиваются значения по умолчанию: 0 для значимых типов и *null* для ссылочных.

Размерность массива (количество элементов в массиве) задается при объявлении (выделении памяти) и **не может** быть изменена впоследствии.

Размерность может задаваться числом или выражением, например:

```
int n = 5;  string[] z = new string[2*n + 1].
```

Варианты и примеры описания одномерного массива:

```
int[] a; // объявлена только ссылка на массив, память под элементы не выделена;
int[] b = new int[4];                          // 4 элемента, значения равны 0;
int[] c = { 61, 2, 5, -9 };                    // размерность вычисляется;
int[] e = new int[4] { 61, 2, 5, -9 };         // избыточное описание.
```

С элементами массива можно делать все, что допустимо для переменных того же типа. При работе с массивом автоматически выполняется контроль выхода за его границы: если значение индекса выходит за границы массива, генерируется исключение IndexOutOfRangeException (см. работу 1.6).

Наиболее распространенные задачи работы с массивами:

– поиск индекса и значений заданного элемента (например, первого, последнего, положительного, отрицательного, максимального и т. п.);

– определение количества, суммы, произведения, среднего арифметического заданных элементов (например, положительных, отрицательных, от 2 до 8);

– сортировка и анализ возможных вариантов расположения элементов.

При работе с массивами используются циклы. Многие действия с массивами упрощаются при использовании алгоритмической конструкции перебора элементов массива:

foreach (*тип перем* **in** *имя_массива*)
{ тело_цикла **}**

где *перем* – имя локальной переменной цикла, которая по очереди принимает все значения элементов массива.

Например для массива:

int[] **mas** = { 24, 50, 18, 3, 16, –7, 9, –1 };
foreach (int **x** in **mas**) Console.WriteLine(**x**);

будут последовательно выведены все числа.

Все массивы в C# имеют общий базовый класс **Array**, определенный в пространстве имен **System**.

Некоторые свойства и методы класса **Array**:

– **Length** (свойство) – количество элементов массива (по всем размерностям);

– **IndexOf (LastIndexOf)** (статический метод) – поиск первого (последнего) вхождения элемента в одномерный массив;

– **Sort** (статический метод) – упорядочивание элементов одномерного массива;

– **Reverse** – изменение порядка следования элементов на обратный.

Пример 1

Формирование и вывод массивов чисел.

1. Создадим проект **con131**. Наберем код программы в теле метода **Main**:

```
Console.Write("Введите размерность массивов (от 5 до 20)  ");
int n = int.Parse(Console.ReadLine());
int[] a = new int[n];  int[] b = new int[n];        // объявление массивов
for (int i = 0; i < n; i++)
 {
   a[i] = i; b[i] = a[i]*a[i];                      // заполнение массивов
   Console.WriteLine("a[{0}] = {1}, b[{0}] = {2}", i, a[i], b[i] );
 }
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 2

Формирование и вывод массива строк.

1. Создадим проект **con132**. Наберем код программы в теле метода **Main**:

```
     // объявляем и заполняем массив дней недели
string[] dw = {"Вс","Пн","Вт","Ср","Чт","Пт","Сб"};
     // выводим рабочие дни с помощью цикла for
for (int i = 1; i < dw.Length - 1; i++)
   Console.WriteLine(i + " рабочий день " + dw[i]);
     // перебираем и выводим все элементы с помощью цикла foreach
foreach (string day in dw)
          Console.WriteLine(day);
    // сначала только объявляем массив месяцев, затем заполняем два элемента
string[] ms = new string[12];
ms[6] = "Июль"; ms[7] = "Август";
int j = 0;
     // перебираем и выводим элементы с помощью цикла foreach
foreach (string m in ms)
     Console.Write("{0}-{1}  ", ++j, m);
Console.ReadKey();
```

2. Протестируем программу. Проанализируем результат.

Пример 3

*Свойства и методы класса **Array**.*

1. Создадим проект **con133**. Наберем код программы в теле метода **Main**:

```
Console.Write("Введите размерность массивов (от 5 до 20)  ");
int n = int.Parse(Console.ReadLine());
Random r = new Random();                  // создание экземпляра класса Random
```

```
int[] a = new int[n];                          // объявление массива
for (int i = 0; i < n; i++)
  {
   a[i] = r.Next(-20, 20);        //заполнение массива случайными числами
   Console.Write("{0,4}", a[i]);              // вывод исходного массива
  }
   // подсчет суммы и количества отрицательных чисел
int sum = 0, num = 0;                          // задание начальных значений
foreach (int x in a)
  if (x < 0)  {  sum += x; num++;  }
Console.WriteLine("\n Сумма отрицательных = {0}, к-во = {1}", sum, num);
int max = a[0];                               // поиск максимального элемента
foreach (int x in a) if (x > max) max = x;
Console.WriteLine("max = " + max);
Array.Sort(a);                                // сортировка элементов массива
foreach (int x in a) Console.Write("{0,4}", x);
Console.WriteLine();
Array.Reverse(a);                             // изменение порядка следования
foreach (int x in a) Console.Write("{0,4}", x);
Console.ReadKey();
```

2. Протестируем программу. Сравним вывод элементов двумя способами.

Задания для самостоятельной работы

Создайте консольные приложения, в которых выполняются заданные действия:

1. По введенному порядковому номеру выводится название дня недели и количество дней до Вс.

2. По введенному обозначению транспортного средства (a – автомобиль, v – велосипед, m – мотоцикл, p – поезд, s – самолет) выводится его название и средняя скорость.

3. Заданы диагонали мониторов в дюймах: 10.1; 11.6; 14; 15.6; 17; 19; 24; 27. Формируется и выводится таблица перевода диагоналей в сантиметры.

4. Формируется массив из *n* натуральных нечетных чисел. Выводятся числа кратные 3.

5. Формируется массив из *n* целых случайных чисел от 10 до 99. Выводятся четные числа и их количество.

6. Формируется массив из *n* целых случайных чисел от –40 до 40. Выводятся нечетные отрицательные числа и их количество.

7. Формируется массив из *n* целых случайных чисел от –50 до 50. Массив упорядочивается: а) выводится сумма и количество положительных чисел; б) Выводятся числа от –20 до +20.

8. *По введенному порядковому номеру месяца и дате выводится: название месяца, количество дней в нем, количество дней, оставшихся до конца текущего месяца, название следующего месяца.

1.5. Работа с двумерными массивами

Цель работы: формирование навыков работы с двумерными массивами в C#.

Введение

В языке C# различают двумерные массивы двух видов: прямоугольные (таблица с одинаковым количеством элементов в строках) и ступенчатые. Примеры описания **прямоугольных** двумерных массивов:

```
int[,] a;                 // объявлена только ссылка, память под элементы не выделена;
int[,] b = new int[2, 3];          // 2 строки, 3 столбца, элементы равны 0
int[,] c = { {1, 2, 3}, {4, 5, 6} };           // размерность вычисляется
int[,] d = new int[2,3] { {1, 2, 3}, {4, 5, 6} };  // избыточное описание.
```

К элементу двумерного массива обращаются, указывая номер строки и столбца, на пересечении которых он расположен **b[i,j]**. Первый индекс всегда воспринимается компилятором, как номер строки.

Пример 1

Формирование и вывод двумерного массива заданных чисел.

1. Создадим проект **con141**. Наберем код программы в теле метода **Main**:

```
    // формируем прямоугольный массив чисел из двух строк
int[,] ar = { { 11,12,13,14,15 }, { 21,22,23,24,25 } };
      // с помощью foreach все элементы выводятся в одну строку
foreach (int x in ar) Console.Write("{0,4}", x);
   Console.WriteLine();
    // выводим числа построчно (в форме таблицы)
for (int i = 0; i < 2; i++)
  {
   for (int j = 0; j < 5; j++)  Console.Write("{0,4}", ar[i,j]);  // j-я строка
    Console.WriteLine();                       // переход на следующую строку
  }
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 2

Формирование и вывод таблицы чисел.

1. Создадим проект **con142**. Наберем код программы в теле метода **Main**:

```
Console.Write("Введите количество строк (n<9):  ");
int n = int.Parse(Console.ReadLine());
Console.Write("Введите количество столбцов (m<9):  ");
int m = int.Parse(Console.ReadLine());
```

```
        // объявляем массив чисел из n строк и m столбцов
int[,] mas = new int[n, m];
        // построчно заполняем массив целыми числами
for (int i = 0; i < n; i++)
 {
   for (int j = 0; j < m; j++) mas[i,j] = 10*i + j;          // i-я строка
 }
        // построчно выводим элементы массива (в форме таблицы)
for (int i = 0; i < n; i++)
 {
  for (int j = 0; j < m; j++)
      Console.Write("{0,4}", mas[i,j]);        // вывод элементов i-й строки
  Console.WriteLine();                         // переход на следующую строку
 }
  Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

В **ступенчатых** массивах количество элементов в разных строках может различаться. В памяти хранится ступенчатый массив иначе, чем прямоугольный – в виде нескольких внутренних массивов, каждый из которых имеет свой размер (массив массивов). Кроме того, выделяется отдельная область памяти для хранения ссылок на каждый из внутренних массивов. Под каждый из массивов, составляющих ступенчатый массив, память требуется выделять явным образом.

Примеры описания ступенчатых двумерных массивов:

```
int[][] a = new int[3][];    // выделение памяти под ссылки на 3 строки
a[0] = new int[5];           // выделение памяти под 0-ю строку (5 элементов)
a[1] = new int[3];           // выделение памяти под 1-ю строку (3 элемента)
a[2] = new int[4];           // выделение памяти под 2-ю строку (4 элемента)
```

Сокращенный вариант: int[][] a = { new int[5], new int[3], new int[4] };

Обращение к элементам ступенчатого массива: a[0][3], a[1][2], a[i][j].

Пример 3

Формирование и вывод ступенчатого массива чисел.

1. Создадим проект **con143**. Наберем код программы в теле метода **Main**:

```
 // построчно объявляем и заполняем массив из 3-х одномерных массивов
int[][] jag = new int[3][]
 {  new int[] {3,7,9,5,12},   new int[] {2,4},  new int[] {1,3,5}  };
          // построчно выводим три внутренних одномерных массива
foreach (int[] arr in jag)
  { foreach (int a in arr) Console.Write("{0,4}", a);
   Console.WriteLine();
  }
  Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Задания для самостоятельной работы

Создайте консольные приложения, в которых выполняются заданные действия:

1. Формируется и выводится прямоугольный массив (5 строк и 6 столбцов) целых случайных чисел от –40 до 40. Вычисляется и выводится: а) сумма чисел в каждой строке; б) среднее арифметическое чисел в каждой строке;

2. Формируется и выводится прямоугольный массив (n строк и m столбцов) целых случайных чисел от –50 до 50. Вычисляется и выводится: а) среднее арифметическое отрицательных чисел в каждой строке; в) сумма и среднее арифметическое положительных четных чисел в каждой строке; д) сумма и среднее арифметическое всех чисел.

3. Формируется и выводится прямоугольный массив – таблица умножения двух чисел от 1 до 10.

```
       Таблица умножения
1   2   3   4   5   6   7   8   9  10
2   4   6   8  10  12  14  16  18  20
3   6   9  12  15  18  21  24  27  30
4   8  12  16  20  24  28  32  36  40
5  10  15  20  25  30  35  40  45  50
```

4. Формируется и выводится прямоугольный массив (6 строк и 8 столбцов) целых случайных чисел от –70 до 70. Вычисляется и выводится: а) максимальный элемент в кажой строке; б) минимальный положительный элемент в каждой строке.

5. Формируется и выводится прямоугольный массив (*n* строк и *m* столбцов) целых случайных чисел от –90 до 90. Вычисляется и выводится: а) минимальный из всех отрицательных элементов; б) максимальный из модулей всех элементов массива.

6. *Формируется и выводится прямоугольный массив (*n* строк и *m* столбцов) целых случайных чисел от –80 до 80. Вычисляется и выводится: а) в каждой строке находится минимальный элемент и заменяется нулем; б) в каждом столбце находится максимальный элемент и заменяется единицей.

7. Формируется массив и выводится треугольная таблица, заполненная: а) единицами; б) нулями.

```
1
1 1
1 1 1
1 1 1 1
1 1 1 1 1
1 1 1 1 1 1
```

```
0 0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0
0 0 0 0
0 0 0
0 0
0
```

1.6. Обработка исключений

Цель работы: формирование навыков обработки исключений.

Введение

Важным показателем эффективности компьютерных программ является защита от сбоев. Для повышения надежности необходимо предусмотреть обработку критических ситуаций, которые могут вызывать ошибки в работе приложений. Перечислим основные способы защиты от сбоев.

– Контроль формата вводимых данных – «умный» парсинг (метод **TryParse**):

```
int a;      if ( !int.TryParse(Console.ReadLine(), out a) )
                 Console.WriteLine("Неверный формат");
             Console.WriteLine(a);
```

– Контроль допустимых значений вводимых данных, например, при инкапсуляции полей с помощью «умных» свойств (**set – get**).

– Контроль операций и результатов, например, путем локальной проверки использования встроенных методов с помощью условий, например:

```
if ( a < 0 )  Console.WriteLine("Неверное значение < 0");
```

– Контроль критических (часто непредвиденных) ситуаций.

Исключительная ситуация, или **исключение** – это возникновение события, (часто непредвиденного или аварийного), которое порождается некорректным использованием команд или аппаратуры. Например: деление на ноль, обращение по несуществующему адресу памяти, попытка прочитать несуществующий файл.

Использование механизма исключений позволяет логически разделить вычислительный процесс на две части – обнаружение аварийной ситуации и ее обработку. Типичный синтаксис:

```
try {  контролируемый блок  }
catch {  блок обработки исключения  }
finally {  блок завершения  }
```

В контролируемый блок **try** включаются потенциально опасные фрагменты программного кода. Все функции, прямо или косвенно вызываемые из этого блока, также считаются ему принадлежащими.

В одном или нескольких блоках обработки исключений **catch** описывается, как обрабатываются ошибки различных типов:

```
catch (тип имя)   {  обработка конкретной ошибки  }
catch (тип)   {  обработка ошибки заданного типа  }
catch   {  обработка любой, часто неизвестной ошибки  }
```

Необязательный блок завершения **finally** выполняется независимо от того, возникла ли ошибка в контролируемом блоке или нет. Собственные исключения можно создавать оператором **throw.**

Механизм обработки исключений:

– функция или операция, в которой возникла ошибка, генерирует исключение;

– выполнение текущего блока прекращается, ищется соответствующий обработчик исключения и ему передается управление;

– если обработчик не найден, вызывается стандартный обработчик;

– наконец, выполняется блок finally, если он присутствует.

Для работы с исключениями предназначен класс **Exception**. Приведем некоторые стандартные исключения:

– **FormatException** – попытка передать в метод аргумент неверного формата;

– **InvalidCastException** – ошибка преобразования типа;

– **ArithmeticException** – ошибка арифметических операций или преобразований;

– **DivideByZeroException** – попытка деления на ноль;

– **OverFlowException** – переполнение выполнения арифметических операций;

– **IndexOutOfRangeException** – индекс массива выходит за границы диапазона;

– **OutOfMemoryException** – недостаточно памяти для создания нового объекта;

– **StackOverFlowException** – переполнение стека;

Пример типичной последовательности обработки исключений:

try { контролируемый блок }
catch (DivideByZeroException) { обработка деления на 0 }
catch (OverflowException) { обработка переполнения }
catch (IndexOutOfRangeException) { обработка неверного индекса массива }
catch (FormatException) { обработка неверного формата }
catch { обработка всех остальных исключений }

Некоторые важные свойства класса **Exception**:

– **Message** – текстовое описание ошибки (только для чтения);

– **Source** – имя объекта, сгенерировавшего ошибку;

– **TargetSite** – метод, сгенерировавший ошибку;

– **InnerException** – ссылка на исключение, послужившее причиной текущего.

Пример 1

Обнаружение и обработка ошибка деления на 0.

1. Создадим проект **con151**. Наберем код программы в теле метода **Main**:

```
Console.Write("Введите делимое ");
int a = int.Parse(Console.ReadLine());
Console.Write("Введите делитель ");
int b = int.Parse(Console.ReadLine());
// в проверяемый блок включим операцию деления и вывод результата
try {  Console.WriteLine(a / b);   }
    // обработка ошибки деления на 0 и вывод системного сообщения
catch (DivideByZeroException e)
    {  Console.WriteLine(e.Message); }
```

// обработка неопознанной ошибки, вывод собственного сообщения

```
catch {  Console.WriteLine("Произошла ошибка");  }
Console.ReadKey();
```

2. Протестируем программу. Сравним результаты обработки исключений при вводе различных целых чисел и 0.

3. Закомментируем первый блок **catch**. Сравним результаты в этом случае.

4. Снимем комментирование. Будем поочередно изменять тип переменных **a** и **b** на **double**: `double a = double.Parse(Console.ReadLine());`

5. Сравним результаты обработки исключений.

Пример 2

Задан строковый массив дней недели. Составим программу, которая по введенному номеру выводит название дня недели и количество дней до Вс, а также обрабатывает ошибки формата ввода и выхода индекса за границы массива.

1. Создадим проект **con152**. Наберем код программы в теле метода **Main**:

```
   // создадим массив дней недели
string[] dw = { "Вс", "Пн", "Вт", "Ср", "Чт", "Пт", "Сб" };
Console.Write("Введите номер дня недели (0 - 6)   ");
   // в проверяемый блок включим ввод номера и вывод сообщения
try {
     int i = int.Parse(Console.ReadLine());
     Console.WriteLine("{0}, до Вс {1} дней", dw[i], 6 - i);
    }
   // обработка ошибки выхода индекса за границы диапазона
catch (IndexOutOfRangeException e)
    {  Console.WriteLine(e.Message);   }
   // обработка ошибки формата ввода
catch (FormatException e)
    {  Console.WriteLine(e.Message);   }
   // обработка неопознанной ошибки
catch {  Console.WriteLine("Ошибка");    }
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

3. Сравним результаты обработки исключений при вводе чисел > 6 и букв.

4. Исследуем, как влияет порядок следования блоков **catch** на обработку исключений от одного блока **try.**

Пример 3

Составим программу, в которой обрабатывается ошибка переполнения.

1. Создадим проект **con153**. Наберем код программы в теле метода **Main**:

// в проверяемый блок **try** включим вычисление и вывод;

// проверяемое выражение заключим в **checked**

```
int a = 1000; int b = 3000;
try {
     int pro = checked (a*a*b);
     Console.WriteLine("Произведение = {0}", pro);
    }
   // обработка ошибки переполнения
catch(OverflowException e)
   { Console.WriteLine(e.Message);  }
   // обработка неопознанной ошибки
catch { Console.WriteLine("Ошибка");  }
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

3. Удалим **checked**. Сравним результат в этом случае.

Задания для самостоятельной работы

Создайте консольные приложения, в которых выполняются заданные действия и обрабатываются исключения:

1. Вычисляется сила тока **i = u/r** по введенным значениям напряжения и сопротивления (тип **int**):

2. Расстояния до звезд **a** = 5 и **b** = 8000 световых лет вычисляется в **км** и **м** sa = a*365*24*3600*300 000 (км).

3. Проверяется правильность ввода логина и пароля, состоящих только из цифр.

4. Задан строковый массив месяцев. По введенному порядковому номеру месяца выводится название месяца и количество дней в нем.

5. Вычисляются значения функций от целочисленных аргументов:

а) y = (x + 5) / (x - 7) *ввод* **x**

в) z = (x – y) / cos(x) *ввод* **x** и **y**

6. Задан строковый массив городов: Брест, Витебск, Гомель, Минск, Полоцк. По введенной первой букве или номеру выводится название города.

1.7. Работа с символами

Цель работы: формирование навыков работы с символами в C#.

Введение

Для работы с символами предназначен тип **char.** Ему соответствует базовый класс **System.Char**. В памяти переменная типа **char** представляется числовым кодом символа. Требуется преобразование в символ:

```
char ch = 'A';                    // в переменной ch код 65
int k = 100;   c = (char)k;       // преобразование в символ 'd' с кодом 100
```

При вводе символа с клавиатуры методом **Read()** и нажатии клавиши Enter в буфер попадает его код, а также коды нажатия клавиши (13) и конца строки (10).

```
int k = Console.Read();           // в переменной код символа  +13+10
char ch = (char) Console.Read();  // в переменной символ  +10+13
```

Лучше использовать метод **ReadLine()** и разборку строки:

```
char s  = char.Parse(Console.ReadLine()); // в переменной только символ
```

Некоторые методы класса **Char**:

– **IsDigit(s), IsLetter(s), IsLower(s), IsUpper(s), IsPunctuation(s), IsControl(s)** возвращают **true**, если символ **s** является соответственно: цифрой, буквой, строчной, заглавной буквой, знаком пунктуации, управляющим символом;

– **GetNumericValue(s)** возвращает числовое значение, если символ **s** цифра;

– **ToLower(s), ToUpper(s)** изменяют регистр букв (в строчные или заглавные);

– **ToCharArray(str)** преобразуют строку в массив символов.

Пример 1

Определение категории введенного символа.

1. Создадим проект **con161**. Наберем код программы в теле метода **Main**:

```
int k; char ch; string mes = "";                //объявление переменных
do {                       // повторение ввода в цикле с постусловием
  Console.WriteLine("Введите символ: ");
  k = Console.Read(); ch = (char)k;     // ввод и преобразование кода в символ
                           // проверка категории символа
  if (char.IsLetter(ch)) mes = "буква";
  else if (char.IsDigit(ch)) mes = "цифра";
  else if (char.IsPunctuation(ch)) mes = "знак пунктуации";
  else mes = "управляющий символ";
  Console.WriteLine("Введен символ {0}, его код {1}, это {2}", ch, k, mes);
   } while (ch != 'Q');               // завершение цикла при вводе буквы Q
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 2

Подсчет количества и суммы всех содержащихся в строке цифр.

Заметим, что символам цифр от 0 до 9 соответствуют коды от 48 до 57.

1. Создадим проект **con162**. Наберем код программы в теле метода **Main**:

```
Console.WriteLine("Введите строку:   ");
string str = Console.ReadLine();          // ввод строки
int k = 0, sum = 0;                       // задание начальных значений
foreach (char s in str)                   // перебор всех символов строки
  {
   if (char.IsDigit(s))                   //если очередной символ цифра,
    {  k++;  sum += s-48;                 //считаем к-во и сумму цифр
       Console.Write(s +" ");             //выводим цифры
    }
  }
```

```
Console.WriteLine("\nВ строке {0} цифр, их сумма = {1}", k, sum);
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 3

Перестановка соседних символов строки.

1. Создадим проект **con163**. Наберем код программы в теле метода **Main**:

```
Console.WriteLine("Введите строку: ");
string str = Console.ReadLine();
int k = str.Length-1;                        // определение  длины строки
string str2 = "";                            // объявление новой пустой строки
          // сформируем строку str2  с переставленными соседними символами
for (int i = 0; i < k ; i += 2)
  { str2 += str[i+1];    str2 += str[i]; }
          // добавим финальный нечетный символ (с четным индексом)
if (k % 2 == 0) str2 += str[k];
 Console.WriteLine(str2);                    //вывод строки str2
 Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 4

*Работа с массивом символов. Использование методов класса **Array.***

1. Создадим проект **con164**. Наберем код программы в теле метода **Main**:

```
Console.WriteLine("Введите строку: ");
string str = Console.ReadLine();                    // ввод строки
          // преобразование строки в массив символов
char[] ch = str.ToCharArray();
Array.Reverse(ch);  Console.WriteLine(ch);          // реверсирование и вывод
Array.Sort(ch);     Console.WriteLine(ch);          // сортировка и вывод
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Задания для самостоятельной работы

Создайте консольные приложения, в которых выполняются заданные действия над символами введенных строк:

1. Подсчитывается количество строчных, заглавных букв, знаков препинания в строке.
2. Подсчитывается количество вхождений заданной буквы (цифры).
3. Заданные строчные латинские буквы преобразуются в заглавные.
4. Проверяется, имеются ли в строке рядом стоящие удвоенные символы.
5. Выводятся все повторяющиеся в строке буквы и подсчитывается количество каждой из них.

6. Подсчитывается количество и сумма всех содержащихся в строке четных (нечетных) цифр.
7. Первые строчные буквы слов строки преобразуются в заглавные.
8. *Содержащиеся в строке цифры 0–5 заменяются на слова «ноль»…«пять».

1.8. Работа со строками

Цель работы: формирование навыков работы со строками.

Введение

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 – 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.

Пример 1

*Использование методов класса **String**.*

1. Создадим проект **con171**. Наберем код программы в теле метода **Main**:

```
        //  исходная строка заимствована из пособии [1]
string str = "прекрасная королева";
  Console.WriteLine(str);                          // вывод
        //выделяем подстроку str2, удаляем символы  "ле"
string str2 = str.Substring(3).Remove(12, 2);
  Console.WriteLine(str2);                         // вывод:  красная корова
        // расщепляем строку str в массив слов mas (разделитель – пробел)
string[] mas = str.Split(' ');
        // выводим все слова s массива mas
  foreach (string s in mas) Console.WriteLine(s);
Array.Sort(mas);                  // сортируем слова массива mas по алфавиту
  foreach (string s in mas) Console.WriteLine(s);          // вывод
        // соединяем слова массива mas в одну строку str3
string str3 = string.Join(" !!! моя ", mas);
  Console.WriteLine(str3);              // вывод: королева !!! моя прекрасная
 string str4 = str3.Replace("!", "?");                 // заменяем все ! на ?
  Console.WriteLine(str4);              // вывод: королева ??? моя прекрасная
        // определяем позицию k вхождения подстроки моя
int k = str4.IndexOf("моя");
        //удаляем из строки str4  символы начиная с k-го до конца
string str5 = str4.Remove(k);
  Console.WriteLine(str5);                              // вывод: королева ???
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

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** – **м**аксимальный размер буфера.

Пример 2

Использование методов класса StringBuilder.

1. Создадим проект **con172**. Наберем код программы в теле метода **Main**:

```
using System;
using System.Text;              // подключение пространства имен System.Text
class Program
{
 static void Main()
 {
  Console.Write("Введите зарплату: ");
  double zar = double.Parse(Console.ReadLine());
  StringBuilder str = new StringBuilder();          // создание объекта
  str.Append("зарплата ");
        // добавляем строку в денежном формате  (тыс и млн отделяются)
  str.AppendFormat("{0,6:C} - за год {1,6:C}", zar, zar*12);
    Console.WriteLine(str);                                // вывод
  str.Replace("р.", "$.");                                 // замена р. на $
    Console.WriteLine("А было бы лучше: " + str);          // вывод
  Console.ReadKey();
 }
}
```

2. Протестируем и откорректируем программу.

Задания для самостоятельной работы

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. Меняется местами соседние слова.

11. Выводятся слова, заключенные в кавычки « » (в угловые скобки < >).

12. Выводятся все слова, начинающиеся с заданной буквы.

1.9. Использование регулярных выражений

Цель работы: формирование навыков использования регулярных выражений в C#.

Введение

Регулярное выражение (*regular expression*) – шаблон, по которому выполняется поиск соответствующего ему фрагмента текста.

Регулярные выражения предназначены для поиска в тексте фрагментов (символов, строк) по заданному шаблону с целью дальнейшей обработки.

Язык описания регулярных выражений содержит символы двух видов: обычные и метасимволы. Обычный символ представляет в шаблоне сам себя. Метасимвол представляет: класс символов (например, **\d** обозначает цифру), уточняющий символ или квантификатор (например, **A{3}** означает, что букву **A** необходимо повторить три раза). Чтобы одиночный метасимвол в шаблоне представлял сам себя, его необходимо экранировать обратным слешем **“\”**, а целое выражение – символом **@,** например: \\ воспринимается как один слеш.

Приведем примеры часто используемых метасимволов.

<u>Метасимволы, представляющие класс символов:</u>

. (точка) любой символ;

[ ] любой *одиночный* символ из набора внутри квадратных скобок;

[^] любой *одиночный* символ, не входящий в набор внутри скобок;

\w любой алфавитно-цифровой символ, то есть буква или десятичная цифра;

\W любой **не** алфавитно-цифровой символ, то есть кроме букв и цифр;

\s любой пробельный символ, то есть пробел, табуляция (\t, \v), перевод строки (\n, \r), новая страница (\f);

\S любой **не** пробельный символ;

\d любая десятичная цифра;

\D любой символ, не являющийся десятичной цифрой.

<u>Уточняющие метасимволы</u>

^ искать только с начала строки, **$** искать с конца строки;

\b встречается только в начале или конце слова; **\B** только внутри слова.

Квантификаторы (задают количество повторений предыдущего элемента):

***** 0 или более повторений; **?** 0 или одно; **+** одно или более повторений;

{n} ровно **n** повторений; **{n,m}** от **n** до **m** повторений.

При задании квантификаторов для последовательности применяется **группирование** с помощью круглых скобок.

Примеры простых шаблонов для поиска:
семизначного номера телефона вида NNN-NN-NN:

[0-9]{3}-[0-9]{2}-[0-9]{2} или **\d{3}-\d{2}-\d{2} ;**

номера автомобиля (4 цифры, 2 буквы, дефис, цифра от 1 до 7):

\d{4}[ABEIKMHOPCTX]{2}-[1-7] ;

Регулярные выражения в .NET реализуются несколькими классами пространства имен **System.Text.RegularExpressions**. Основной класс **Regex.** Возможны два способа обработки текста: 1) вызов статических методов класса **Regex** с передачей в качестве параметров исходной строки и шаблона; 2) создание объекта **Regex** с передачей шаблона в конструктор класса.

Некоторые методы класса **Regex.**

IsMatch(str, reg) – поиск вхождения подстроки с шаблоном **reg** в строку **str;**

Split(text) – разбиение текста **text** на массив строк по шаблону-разделителю;

Match или **Matches** – извлечение из текста одного или всех вхождений.

Метод Match возвращает одноименный объект Match с информацией о совпадении. Метод Matches возвращает коллекцию MatchCollection, в которую входят объекты Match для всех совпадений в проанализированном тексте.

Пример 1

Проверка телефонного номера вида NNN-NN-NN.

1. Создадим проект **con181**. Наберем код программы:

```
using System;
// подключаем пространство имен System.Text.RegularExpressions
using System.Text.RegularExpressions;
class Program
{
  static void Main()
  {
    Console.WriteLine("Введите номер телефона");
    String str= Console.ReadLine();
// создаем шаблон регулярного выражения
    String reg = @"^\d{3}-\d{2}-\d{2}$";
// проверяем совпадение введенной строки с шаблоном
    if (Regex.IsMatch(str, reg))
        Console.WriteLine("{0} похоже", str);
    else Console.WriteLine("{0} ошибка", str);
    Console.ReadKey();
  }
}
```

2. Протестируем программу. Откорректируем программный код.

3. Модифицируем шаблон для определения номера телефона с разделением групп цифр пробелом или дефисом

```
String reg = @"^\d{3}[ -]\d{2}[ -]\d{2}$";
```

Пример 2

Разбиение текста на слова.

1. Создадим проект **con182**. Наберем код программы в теле метода **Main**:

```
// задаем исходную строку и шаблон разделителя
string text  = " салат - 4 руб, борщ - 10 руб, чай - 1 руб.";
string reg  = "[-,.]+";
Regex  r = new Regex(reg);                // создаем объект класса Regex
// разбиваем строку на слова по шаблону
string[] words = r.Split(text);
foreach ( string wrd in words )
   Console.WriteLine(wrd);
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Пример 3

Замена фрагментов текста.

1. Создадим проект con183. Наберем код программы в теле метода Main:

```
// задаем исходную строку и шаблоны
string text = "телефон 400 рб, часы 120 грн, компьютер 560 $";
string reg = "рб|грн|\\$";                // шаблон для поиска
string zam = "руб";                       // шаблон для замены
// выполняем поиск и замену по шаблонам и сразу вывод
Console.WriteLine(Regex.Replace(text, reg, zam));
Console.ReadKey();
```

2. Протестируем программу. Откорректируем программный код.

Задания для самостоятельной работы

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

1. Проверяется, есть ли в строке заданное слово (в любом регистре).
2. Проверяется, есть ли в строке слово из N букв.
3. Проверяется, содержит ли строка число из N цифр.
4. Проверяется, содержит ли строка цифры (знаки препинания, только буквы).
5. Все пробелы в строке заменяются на два символа подчеркивания __
6. Все знаки препинания (**. , ; : ? !**) в тексте заменяются на символ звездочка*.

7. Подсчитывается количество строчных (прописных) букв.

8. Подсчитывается количество удвоенных согласных (гласных).

9. Выводятся только слова, которые содержат заданное количество букв.

10. Выводятся слова, которые содержат не менее заданного количества букв.

11. Строка разбивается на слова, которые выводятся в обратном порядке.

12. Проверяется корректность ввода номера автомобиля.

13. Проверяется, содержит ли введенный логин не менее n букв, а пароль не менее m цифр.

14. *Проверяется корректность ввода адреса электронной почты.

15. *Проверяется корректность IP-адреса.

16. *Подсчитывается количество и сумма всех содержащихся в строке цифр.

1.10. Работа с файлами

Цель работы: формирование навыков работы с файлами.

Введение

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);.

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

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

fw.WriteLine("Запись в файл");

По завершению 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 соответствующие фрагменты кода в блок **try**, например:

```
try {    StreamReader  frr = new StreamReader(imf);
         string s = frr.ReadToEnd();   Console.WriteLine(s);
         frr.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.

Пример 1

*Запись в текстовый файл **my.txt** и чтение строк.*

1. Создадим проект **con191**. Наберем код программы:

```
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:\work\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:\\work\\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();
  }
}
```

2. Протестируем программу. Сравним варианты вывода.

Пример 2

*Составить программу, которая подсчитывает, выводит на консоль и записывает в файл **money.txt** (папка **Data**) ежемесячный баланс. Доходы вводятся еженедельно (4 раза за месяц), а количество источников расхода неизвестно (для завершения вводить три нуля).*

1. Создадим проект **con192**. Наберем код программы в теле метода **Main**:

```
    // создаем объект класса  DirectoryInfo
DirectoryInfo di = new DirectoryInfo("Data");
    // если папка уже существует, удаляем ее вместе с содержимым
if (di.Exists)  di.Delete(true);
    // создаем новую папку Data в текущей с exe-файлом папке ( ..bin/debug/)
di.Create();
    // создаем файл money.txt в папке Data (экранируем слеш)
StreamWriter sw= File.CreateText("Data\\money.txt");
```

```
int sumD = 0;                         // начальное значение суммы доходов за месяц
     // вводим доходы за каждую из 4-х недель месяца
for (int i = 1; i < 5; i++)
  {  Console.Write("Введите доход за {0} неделю: ", i);
    string debit = Console.ReadLine();
    sumD += int.Parse(debit);
  }
Console.WriteLine("=== Доход за месяц: " + sumD);      // вывод на консоль
     sw.WriteLine("=== Доход за месяц: " + sumD);      // запись в файл
Console.WriteLine();
int sumR = 0; string credit = "";     // начальное значение суммы расходов
     // организуем многократный ввод расходов (пока не введено три нуля 000)
while (credit != "000")
  { Console.Write("Введите расход  ");
    credit = Console.ReadLine();
    sumR += int.Parse(credit);
  }
Console.WriteLine("=== Суммарный расход: " + sumR);    // вывод на консоль
     sw.WriteLine("=== Суммарный расход: " + sumR);    // запись в файл
Console.WriteLine();
int balans = sumD - sumR;                  // расчет и вывод баланса
string mes = (balans >= 0) ? "хорошо" : "плохо";
Console.WriteLine("=== Баланс: {0}. Это {1}!", balans.ToString(), mes);
     sw.WriteLine("=== Баланс: {0}. Это {1}!", balans.ToString(), mes);
sw.Close();                                //закрываем файл  money.txt
 Console.ReadKey();
```

2. Протестируем программу. Проанализируем вывод даходов и расходов.

Задания для самостоятельной работы

1. Модифицируйте программу **con192** так, чтобы в файл записывались все доходы и расходы.
2. Составьте программу, которая дописывает в файл **spis.txt** фамилию, имя и возраст студента.
3. Составьте программу, которая выводит на консоль и записывает в файл **week.txt** день недели, месяц и год.
4. Составьте программу, которая считывает из файла **stroki.txt** и выводит на консоль случайную пословицу.
5. Составьте программу, которая выводит на консоль и записывает в файл **tab.txt** таблицу умножения.
6. *Добавьте обработку исключений в программу **con192** (выводится сообщение «Повторите ввод», если введено не число).
7. Составьте программу, которая считывает из списка в файле **citata.txt** и выводит на консоль цитату дня по указанному номеру.
8. *Добавьте обработку исключений в программу задания 7 (выводит сообщение «Повторите», если введен номер больше, чем количество цитат в списке).

2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ СОБСТВЕННЫХ КЛАССОВ

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** – 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.

2.1. Создание класса и объекта. Методы. Конструкторы

Цель работы: формирование навыков создания класса, объекта, методов. Создание и использование конструкторов.

Введение

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 (смотри раздел 1).

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 выполнения. 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 (т. е. является процедурой в терминологии Pascal). 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 1

Создание класса и объекта. Создание и вызов метода. Конструкторы.

1. Создадим проект **con211**.

2. В едином пространстве имен **namespace con211** с шаблоном класса **Program** создадим класс **Build** (проект строения). По умолчанию он имеет модификатор доступа **internal.**

3. В этом классе объявим три поля **name** (имя), **area** (площадь), **kvo** (количество жильцов) с модификаторами доступа **public**. Создадим метод **ShowInfo()**, который вычисляет площадь на одного жильца и выводит информацию о строении.

```
class Build
{ public string name; public double area; public int kvo;
  public void ShowInfo()
  { Console.WriteLine("В доме {0} площадью {1} живет {2} чел, на человека {3:f2}",
                      name, area, kvo, area/kvo);
  }
}
```

4. В методе **Main()** класса **Program** создадим объект **dom1** класса **Build,** (т. е. построим дом по проекту **Build**), используя конструктор по умолчанию (без параметров). Зададим значения полей (параметры нашего дома). Вызовем метод **ShowInfo()**.

```
class Program
{ static void Main()
  {  Build dom1 = new Build();                    // создание объекта dom1
```

```
    dom1.name="Дача"; dom1.area=30; dom1.kvo=4;          // инициализация
    dom1.ShowInfo();                                      // вызов метода
    Console.ReadKey();
  }
}
```

5. Протестируем программу, изменяя параметры объекта **dom1**. Заметим, что при использовании конструктора по умолчанию значения полей задавать не совсем удобно.

6. Модифицируем программу. Создадим в классе **Build** собственный конструктор с тремя параметрами (**nm, ar, k**). Конструктор по умолчанию (без параметров) теперь тоже необходимо записать в явном виде.

```
public Build() { }                            // конструктор без параметров
public Build(string nm, double ar, int k)  // конструктор с параметрами
  {  this.name = nm;   this.area = ar;  this.kvo = k;  }
```

Служебное слово **this** используют для указания на конкретный экземпляр объекта, а также для устранения неоднозначности между полями и локальными переменными или параметрами методов. Например, чтобы не плодить большое количество имен, полям (т. е. переменным класса) и используемым внутри метода локальным переменным (значения которых передаются через соответствующие параметры), обычно дают одинаковые имена. При этом конструктор с параметрами будет выглядеть так:

```
public Build(string name, double area, int kvo)
   {  this.name = name;  this.area = area;  this.kvo = kvo;  }
```

Служебное слово **this** указывает на поля класса **Build** (выделены полужирным), которым при создании объекта будут присвоены значения локальных переменных, переданные через одноименные параметры (выделены курсивом). В дальнейшем мы тоже будем придерживаться такой практики задания имен.

7. В методе **Main()** класса **Program** создадим объект **dom2** класса **Build,** (т. е. построим новый дом по тому же проекту **Build**), используя теперь конструктор с параметрами. Снова вызовем метод **ShowInfo()**.

```
Build dom2 = new Build("Коттедж", 80, 6);
dom2.ShowInfo();
```

8. Протестируем программу, изменяя параметры объекта **dom2**. Заметим, что использование собственного конструктора гораздо удобнее.

Задания для самостоятельной работы

1. Модифицируйте программу примера **1** (**con211**), добавив поле **floor** (количество этажей). Создайте два объекта типа **Build** с разными параметрами и способами инициализации. Создайте приложения, в которых определяются классы, поля, конструкторы, методы, создаются и инициализируются 2–3 объекта.

2. Класс **Student**. Метод **ShowInfo** выводит фамилию, имя, курс, возраст.

3. Класс **Computer.** Метод **Info** выводит модель (IBM, Asus, Sony) и параметры компьютера (объем ОЗУ и жесткого диска).

4. Класс **Tovar**. Метод **Kupi** выводит название (тетрадь, книга, ручка), цену, наличие на складе (есть, нет), количество.

5. Класс **Pogoda**. Метод **Show** выводит город (Минск, Брест, Гомель), температуру, осадки (ясно, пасмурно, гроза), направление и скорость ветра.

6. Класс **Transport**. Метод **ShowInfo** выводит параметры транспортного средства: тип (автомобиль, мотоцикл, велосипед), цвет, скорость, масса.

7. Класс **Animal**. Метод **Golos** выводит вид (кошка, собака, попугай), имя (Мурка, Шарик, Кеша), голос (мяу, гав, ррр).

8. Класс **Figura.** Метод **ShowArea** выводит название (квадрат, прямоугольник) и параметры фигуры (основание, высоту), вычисляет и выводит площадь.

2.2. Перегрузка методов

Цель работы: формирование навыков создания перегруженных методов.

Введение

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.

Пример 1

Использование перегрузки методов. Вычисление периметров разных фигур.

1. Создадим проект **con221**.

2. В классе **Program** создадим три статических метода с одинаковым именем **Perim**, но разными сигнатурами. В методе **Main** будем вызывать эти методы.

```
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();
  }
}
```

3. Протестируем программу, вызывая метод **Perim** с разными параметрами. Обратим внимание на вариант метода с переменным числом параметров.

Пример 2

Заказ номеров в гостинице. Использование перегрузки конструкторов.

1. Создадим проект **con222**.

2. В едином пространстве имен **con222** с классом **Program** создадим класс **Zakaz** с полями **fam** (фамилия), **size** (количество мест в номере), **comfort** (комфортность), методом **Show** (показать заказ) и четырьмя конструкторами с разным числом и типами параметров.

```
class Zakaz
{
    private string fam; private int size; private string comfort;
       // создаем четыре конструктора
   public Zakaz(string fm, int sz, string cmf)          // 3 параметра
   {  fam = fm; size = sz; comfort = cmf;  }
   public Zakaz(string fm, int sz)                      // 2 параметра
   {  fam = fm; size = sz; comfort = "стандарт";  }
   public Zakaz(string fm)                              // 1-параметр
   {  fam = fm; size = 3; comfort = "стандарт";  }
   public Zakaz()                                       // без параметров
   {  fam = "неизвестный"; size = 6; comfort = "общежитие";  }
   public void Show()
   {  Console.WriteLine("{0} забронировал {1} местный номер класса {2}",
                        fam, size, comfort);    }     }
```

3. В методе **Main** класса **Program** будем создавать объекты класса **Zakaz** с разными параметрами и вызывать один и тот же метод **Show**.

```
class Program
{ static void Main()
   {  Zakaz z1 = new Zakaz("Иванов", 1, "Люкс");  z1.Show();
      Zakaz z2 = new Zakaz("Петров", 2);          z2.Show();
      Zakaz z3 = new Zakaz("Сидоров");            z3.Show();
      Zakaz z4 = new Zakaz();                     z4.Show();
      Console.ReadKey();  }
}
```

4. Протестируем программу, вызывая метод **Zakaz** с разными параметрами.

Задания для самостоятельной работы

1. Модифицируйте программу примера 1 (**con221**), добавив вариант перегрузки метода **Perim** для определения периметра квадрата по его стороне: **4*a**.

2. Модифицируйте программу примера 2 (**con222**), добавив возможность многократного ввода заказа с клавиатуры (фамилия, количество мест в номере, комфорт). Завершение ввода – символ **Q**. Создайте приложения, в которых определяются классы, поля, конструкторы, методы, создаются и инициализируются 2–3 объекта.

3. Класс **Figura.** Метод **ShowArea** перегружен. В зависимости от количества введенных параметров выводится название фигуры (один параметр – квадрат, два – прямоугольник, три – трапеция), вычисляется и выводится площадь.

4. Класс **Tour**. Метод **TourCalc** перегружен. Стоимость тура вычисляется в зависимости от количества и типа введенных параметров: без параметров – Минское море, бесплатно; один параметр (страна) – 1 день, 50 руб; два параметра (страна, количество дней **n**) – стоимость = 50***n**.

2.3. Инкапсуляция. Сокрытие полей, создание свойств

Цель работы: формирование навыков управления доступом к полям.

Введение

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 1

Сокрытие полей, создание свойств.

1. Создадим проект **con231**.

2. В едином пространстве имен **con231** с шаблоном класса **Program** создадим класс **Student**. В этом классе объявим два поля **fam** (фамилия) и **kurs** (курс). Создадим конструкторы и метод **ShowInfo()**, который выводит информацию о студенте.

Внимание! Для удобства отладки программы все поля и методы сначала объявляем **public** (общедоступные). Имена полей вводим с маленькой буквы!

```
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);
    }
 }
```

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()**.

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

3. Протестируем программу с разными параметрами. Существенный недостаток – незащищенность от ввода абсурдных данных (например, можно ввести, что студент st2 не имеет фамилии и учится на отрицательном курсе –7.

4. **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.

5. Для этого 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 окно **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**.

6. 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**). А если фамилия не введена (поле **fam** пустое), то блок **get** будет возвращать значение «неизвестный». Измененный фрагмент кода будет выглядеть так:

```
private string fam;                              // поле стало закрытым
public string Fam                                // сгененрировано свойство
{
  get  { return (fam != "") ? fam : "неизвестный"; }     // получение значения
  set { fam = value.ToUpper(); }                         // установка значения
}
```

7. 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;  }      // установка значения
}
```

8. 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.

9. Протестируем окончательный вариант с разными параметрами.

Задания для самостоятельной работы

1. Модифицируйте программу примера1 (**con231**).

2. Добавьте еще два поля: имя **name** и возраст **age**. Инкапсулируйте их, введя ограничения на возраст от 15 до 35 лет. Добавьте конструктор с четырьмя параметрами: **public Student(string fam, string name, int kurs, int age)**.

Протестируйте программу, изменяя параметры инициализации, например;

```
Student st3 = new Student("Петров", "Петр", -7, 120);   st3.ShowInfo();
```

3. Создайте подсчет количества вызовов метода **ShowInfo** для каждого студента. Для этого добавьте поле **id** и задайте метод доступа к нему только для чтения (есть только **get**):

```
private int id = 300;
public int Id  {  get { return id++; }   }
```

Протестируйте окончательный вариант, повторяя инициализацию.

4. Создайте приложения, в которых определяются классы, поля, конструкторы, создаются и инициализируются 2–3 объекта. Поля инкапсулируются. Информация выводится методом **Show.**

5. Создается класс **Avto** с полями: марка автомобиля **brand**, цвет **color**, скорость **skor**. Поля инкапсулируются с ограничениями (скорость от 20 до 120 км/ч).

6. Создается класс **Kadry** с полями: фамилия **fam**, возраст **age**, должность **dol**, стаж **staj**. Поля инкапсулируются с ограничениями (возраст от 16 до 60, стаж от 0 до 45).

7. Создается класс **Computer** с полями: модель **model,** объем ОЗУ **ram** и жесткого диска **hdd**. Поля инкапсулируются с ограничениями (объем ОЗУ от 2 до 32 Гбайт, жесткого диска от 200 до 2000 Гбайт.

8. *Создается класс **Tovar** с полями: название **name**, цена **price**, количество **kvo**. Поля инкапсулируются с ограничениями (цена от 1 до 20, количество от 0 до 10). Вычисляется стоимость заказанного товара каждого вида и всего заказа.

2.4. Визуальное проектирование классов

Цель работы: формирование навыков визуального проектирования классов.

Введение

Технологии визуального объектно-ориентированного проектирования программ основаны на использовании принципов унифицированного языка моделирования **UML** (*Unified Modeling Language*), который представляет собой язык графического описания модели проектируемой системы.

Подобно тому, как алгоритмические конструкции представляются с помощью блок-схем, для изображения классов, а также связей между ними используют **диаграммы классов** (*class diagrams*). Диаграмма классов представляет статическую модель системы, ее структуру. Она не описывает поведение системы или механизмы взаимодействия экземпляров классов. Основная цель – показать классы, их состав и отношения между ними.

Основные элементы на диаграммах классов изображаются прямоугольниками, а их отношения – линиями. Стрелка, например, изображает наследование. Прямоугольник (класс) содержит секции, отражающие его состав. Следует подчеркнуть, что UML – это абстрактный язык описания и графического представления системы. Его реализация в Microsoft Visual Studio использует привычные для программиста обозначения и термины. На рис. 2.1 приведен пример диаграммы простейших классов в среде Visual Studio и описания класса Computer на языке C#.

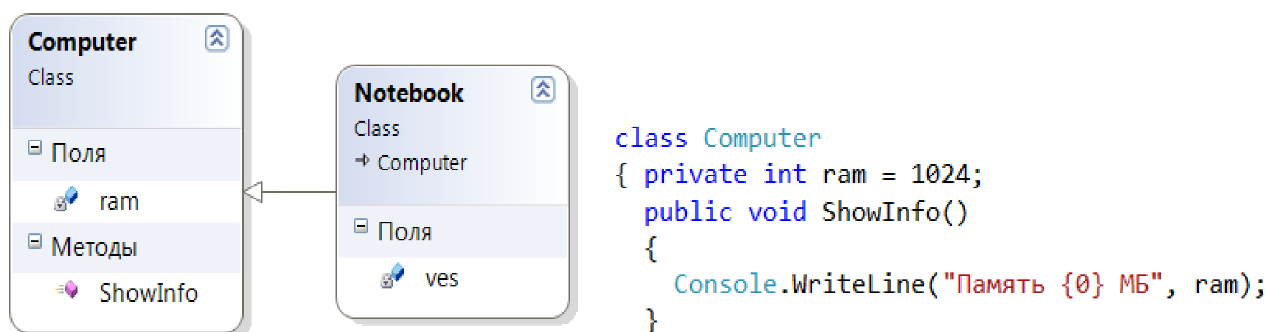


Рис. 2.1. Диаграмма классов и описание на языке C#

Система Visual Studio позволяет автоматизировать процесс проектирования классов. Она содержит набор инструментов визуального конструирования классов в интерактивном режиме. Так, для создания и отображения классов разных типов предназначены графические шаблоны – стереотипы, которые от-

личаются не только визуально, но и функционально (класс, абстрактный класс, интерфейс). Элементы класса (поля, свойства, методы) и их модификаторы помечены разными значками и отличаются по цвету.

Пример 1

Визуальное проектирование класса ***Computer****.*

1. Спроектируем класс **Computer**, который содержит поля **model** (модель) и **ram** (оперативная память), а также методы **Start** (включение) и **End** (выключение).

2. Создадим новый проект **con241**. По умолчанию он имеет класс **Program**.

3. В этом же пространстве имен в отдельном файле создадим класс **Computer** с помощью визуального **Конструктора классов** (Class Designer).

4. Для его вызова в окне **Обозреватель решений** (Solution Explorer) выделим **имя проекта** и щелкнем по значку **Перейти к схеме классов** (рис 2.2, *а*). Откроется окно **Схема классов** (ClassDiagrams) с созданным по умолчанию классом **Program**, содержащим шаблон метода **Main** (рис 2.2, *б*).

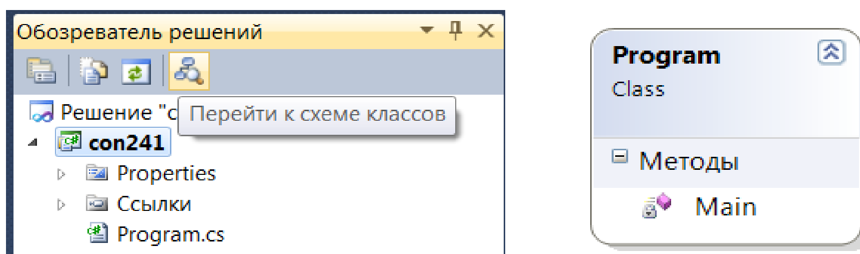


Рис 2.2. Окно **Обозреватель решений** (*а*) и изображение класса Program (*б*)

5. С помощью контекстного меню **Добавить** или **Панели элементов** добавим на схему класс **Computer**. Для этого на панели инструментов выберем пункт **Класс** (рис 2.3, *а*) и перетащим мышью в окно **Схема классов**. В открывшемся диалоговом окне **Новый класс** введем имя **Computer** (рис. 2.3, *б*). По умолчанию в нашем проекте будет создан новый файл **Computer.cs** с автоматически сгенерированным шаблоном кода класса **Computer**.

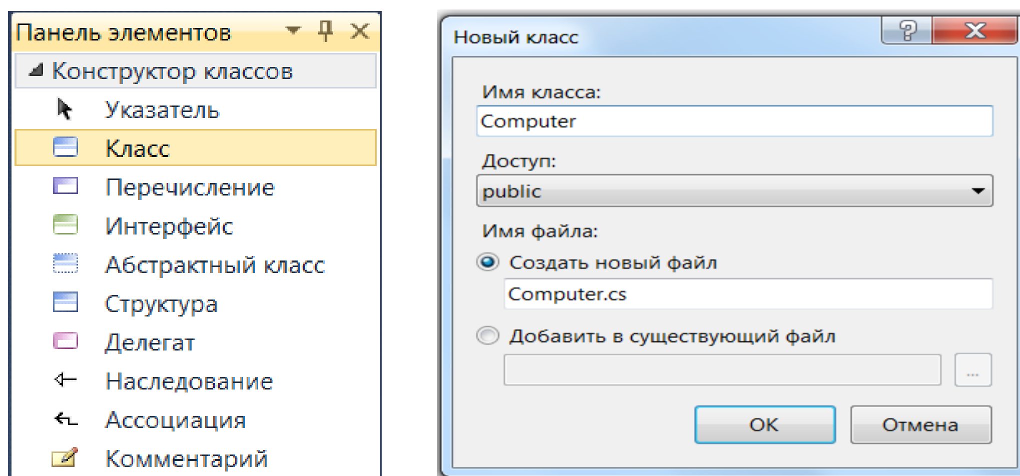


Рис. 2.3. Панель элементов (*а*) и диалоговое окно Новый класс (*б*)

6. Спроектируем теперь его элементы (поля, конструкторы, методы, свойства). Выделим на диаграмме изображение класса. Правой кнопкой мыши вызовем контекстное меню **Добавить** (рис. 2.4, *а*) и выберем требуемый элемент, например, **Поле**. Введем его имя **model** на изображении класса (рис. 2.4, *б*). Можно также воспользоваться панелью Свойства (рис. 2.4, *в*).

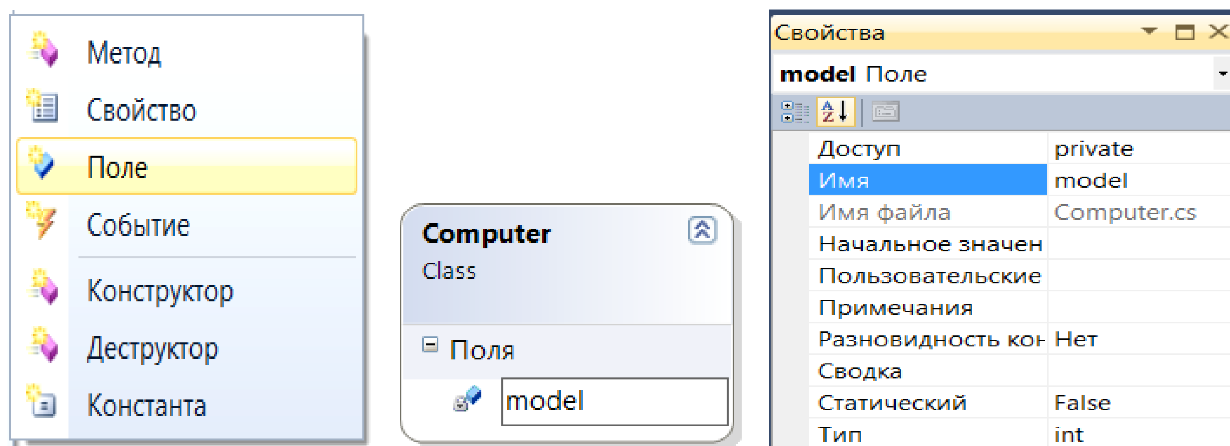


Рис. 2.4. Панель элементов (*а*), изображение класса (*б*) и панель Свойства (*в*)

7. На панели **Свойства** можно задавать характеристики каждого элемента. Однако удобнее использовать окно **Сведения о классах**, в котором в форме таблицы задаются имя, тип, модификатор и другие параметры сразу всех элементов класса (рис. 2.5, *а*).

Будем создавать необходимые элементы класса **Computer** в соответствии с рис. 2.5, *б* и задавать их параметры в соответствии с рис. 2.5, *а*. Рекомендуем сначала задавать все поля, затем методы и конструкторы. Напомним, что в языке C# конструктор является особым методом с тем же именем, что и класс (в нашем примере **Computer**), но не имеющим никакого типа (даже **void**)!

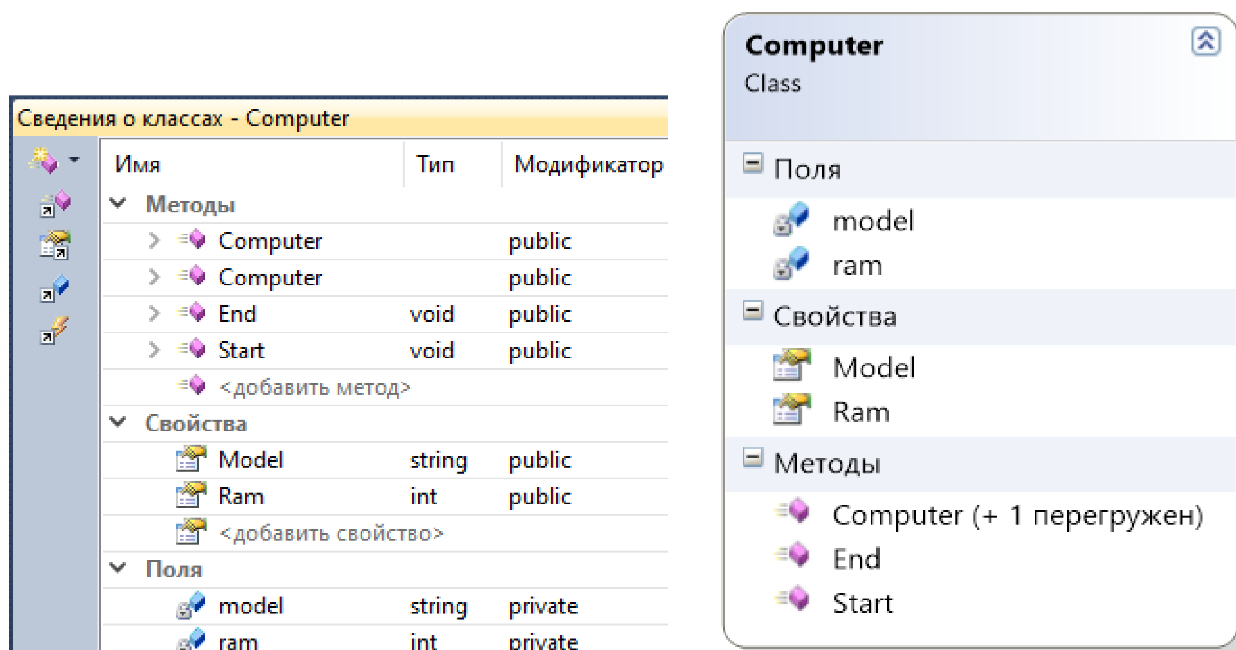


Рис. 2.5. Конечный вид окна **Сведения о классах** (*а*) и класса **Computer** (*б*)

В результате визуального проектирования в окне программы автоматически генерируются шаблоны элементов, в которые будем вводить программный код. На первом этапе целесообразно все поля задавать общедоступными (*public*), и лишь убедившись в отсутствии ошибок, инкапсулировать поля, настроив ограничения доступа в методах set – get. Окончательный вид программы:

```
public class Computer
{
  private string model; private int ram;         // скрытые поля
  public  string Model                           // общедоступные свойства
   {  get { return model ; }
      set { model = (value!="") ? value : "noName"; }
   }
  public int Ram
   {  get { return ram; }
      set { ram = (value < 500) ? 640 : value;  }
   }
  public Computer() {  }                    // конструктор без параметров
  public Computer(string model, int ram)              // с параметрами
   {  this.Model = model;  this.Ram = ram;  }
  public void Start()                       // общедоступный метод Start
   {  Console.WriteLine("{0} работает, память = {1}", Model, Ram); }
  public void End()                         // общедоступный метод End
   {  Console.WriteLine("{0} выключается", Model); }
}
class Program
{  static void Main()
   { Computer comp = new Computer("IBM", 2048);
     comp.Start();  comp.End();
     Console.ReadKey();
   }
}
```

9. Протестируем программу, изменяя параметры.

Задания для самостоятельной работы

Используя инструменты и методы визуального проектирования, создайте приложения, в которых определяются классы, поля, конструкторы, свойства. Информация выводится методом **Show.** В методе **Main** класса Program создаются и инициализируются 2–3 объекта. Демонстрируется ограничение недопустимых значений.

1. Создается класс **Avto** с полями: модель автомобиля **model**, цвет **color**, скорость **skor**. Поля инкапсулируются с ограничениями (скорость от 60 до 150 км/ч).

2. Создается класс **Student** с полями: фамилия **fam**, возраст **age**, курс **kurs**. Поля инкапсулируются с ограничениями (возраст от 16 до 24, курс 2–5).

3. Создается класс **Sotrudnik** с полями: фамилия **fam**, стаж **staj**, зарплата **zar**. Поля инкапсулируются с ограничениями (стаж от 3 лет, зарплата от 500 руб).

4. Создается класс **Build** с полями: название **name**, площадь **area** (от 10 до 200), количество жильцов **kvo** (от 2 до 9). Поля инкапсулируются с ограничениями.

5. *Создается класс **Tovar** с полями: название **name**, цена **price**, количество **kvo**. Поля инкапсулируются с ограничениями (цена от 1 до 10 руб, количество от 0 до 10). Вычисляется стоимость заказанного товара.

2.5. Наследование

Цель работы: формирование навыков реализации наследования.

Введение

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 представляется в виде деревьев, в которых более общие объекты (предки) располагаются ближе к корню, а более специализированные (потомки) – на ветвях и листьях (рис. 2.6).

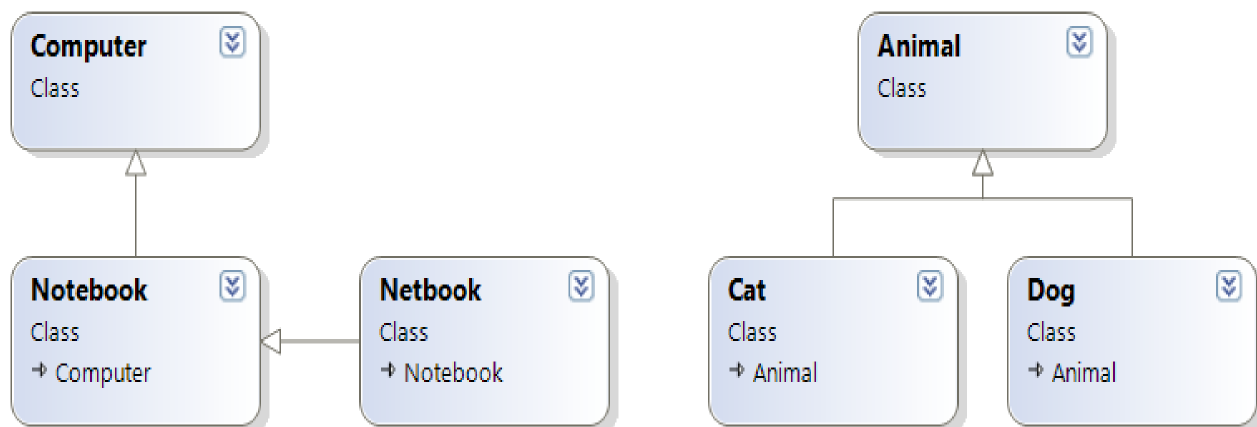


Рис. 2.6. Иерархии классов в окне Схема классов

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 класса (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.

Отметим, что элементы базового класса с модификатором **private** в классе-наследнике недоступны, для них следует использовать модификатор **protected**, а для самого базового класса **public** или **internal.**

Спроектируем класс **Notebook**, который имеет два поля **model** (модель) и **ram** (оперативная память), а также методы **Start** (включение) и **End** (выключение). Ранее мы уже создали класс **Computer** с такими полями и методами. Имеет смысл считать класс **Notebook** потомком класса **Computer**, который наследует эти поля и метод **Start** без изменения (при включении они работают одинаково). Особенность ноутбука – наличие батареи, которая может заряжаться и после выключения ноутбука. Поэтому в класс **Notebook** надо добавить поле **time** (требуемое время зарядки батареи) и переопределить метод **End**, чтобы он показывал это время.

Пример 1

Реализация наследования.

1. Откроем ранее созданный проект **con241** с классами **Program** и **Computer**.

2. С помощью визуального конструктора классов в том же пространстве имен в отдельном файле создадим класс **Notebook.**

3. Для задания отношения наследник-предок на **Панели элементов** выберем пункт **Наследование** и мышью с нажатой левой кнопкой протащим стрелку от прямоугольника класса **Notebook** до класса **Computer** (рис 2.7).

4. С помощью окна **Сведения о классах** зададим поля и методы класса **Notebook** в соответствии с рис. 2.7.

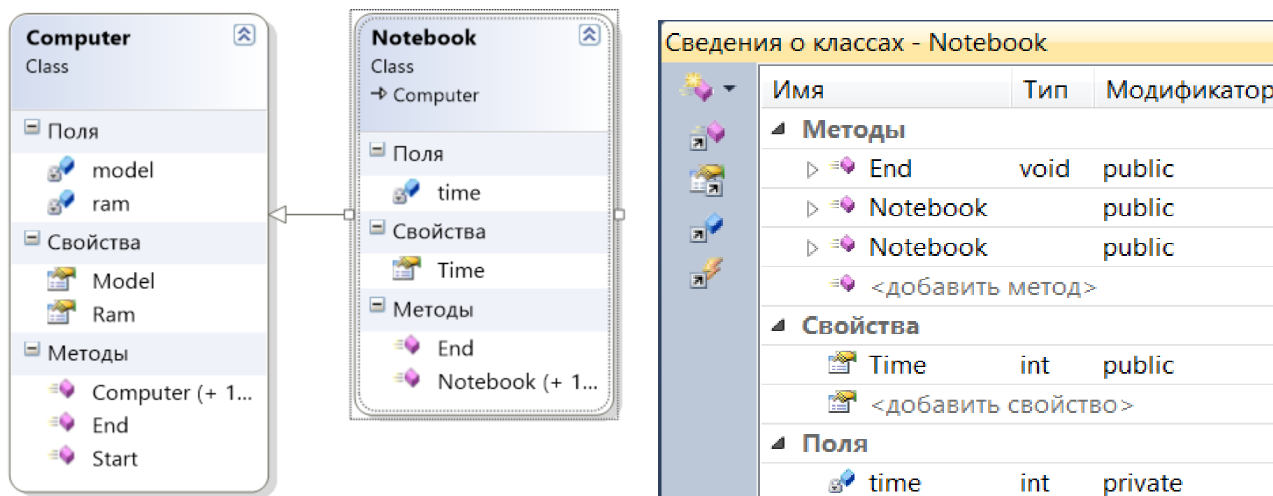


Рис. 2.7. Проектирование класса Notebook

5. Введем в автоматически сгенерированные шаблоны элементов класса **Notebook** программный код. Инкапсулируем поле **time.**

```
public class Notebook : Computer
{
  private int time;
  public int Time
   {  get { return time;  }
      set { time = (value < 10) ? 15 : value;  }    }
   // Конструкторы не наследуются, а вызываются! Поэтому их нужно создавать
   // в каждом классе-потомке. При этом можно ссылаться на базовый класс
   // и лишь добавлять новые параметры.
  public Notebook() { }
  public Notebook(string model, int ram,  int time) :  base(model,ram)
     {  this.Time = time;  }
  // Создадим новый метод End класса – наследника с модификатором override.
  public override void End()
  {  Console.WriteLine("{0} выключается, заряд {1} мин", Model, Time);
  }
}
```

// Чтобы он смог переопределить одноименный метод базового класса,
// зададим тому в классе – предке **Computer** модификатор **virtual.**

6. Протестируем программу, добавив в класс **Program** создание экземпляров класса **Notebook** и вызовы методов:

```
Notebook nb = new Notebook("Asus", 1024, 120);
   nb.Start();     nb.End();
Computer comp2 = new Notebook("Dell", 4096, 30);
   comp2.Start();  comp2.End();
```

7. Обратим внимание на важное проявление принципа полиморфизма: объект **comp2** имеет тип **Computer**, но ведет себя как ноутбук, поскольку инициализируется с помощью конструктора **Notebook.**

Задания для самостоятельной работы

1. Модифицируйте пример **1** (**con241):** добавив **public class Netbook** – потомок **Notebook**. В нем определите поле **mas** (масса). Переопределите метод **Start**, добавив информацию о массе. Протестируйте, изменяя параметры.

2. Используя инструменты и методы визуального проектирования, создайте классы с наследниками, содержащие указанные поля, конструкторы и методы. В классе **Program** создаются и инициализируются 2–3 объекта и указанными методами выводится информация. Продемонстрируйте результаты, изменяя параметры.

3. Спроектируйте класс **Transport** с полями **model** (модель)**, speed** (скорость), **mas** (масса) и методами **Start**, **Stop**, **ShowInfo.** Наследуйте от него классы **Avto** (автомобиль), **Moto** (мотоцикл), **Velo** (велосипед).

4. Спроектируйте класс **Animal** с полями **name** (имя)**, ves** (вес)**, col** (цвет) и методами **Run** (бегать), **Sleep**(спать), **Golos** (голос). Наследуйте от него классы **Cat** (Кот) и **Dog** (Собака). Переопределите метод **Golos** для каждого животного (**мяу-мяу** и **гав-гав**).

5. Спроектируйте класс **Tovar** с полями **name** (название), **price** (цена) и методом **Calc** (расчет и печать стоимости). Наследуйте от него классы **Book** (книги) с полем **kvo** (количество), **Pen** (ручки) с полем **kvo** и **Candy** (конфеты) с полем **ves** (вес). Переопределите метод **Calc** для расчета стоимости каждого товара.

2.6. Абстрактные классы. Интерфейсы

Цель работы: формирование навыков создания абстрактных классов и интерфейсов.

Введение

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. Абстрактный метод 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. Например, имеет смысл объявить класс **Tovar** абстрактным. В нем задать общие для всех товаров поля: **name** (название), **price** (цена), **summa** (стоимость). А вот конкретные характеристики для каждого вида товаров и соответствующих им классов-потомков будут разные: для класса **Book** (книги) это количество экземпляров (поле **kvo**), для класса **Candy** (конфеты) – это граммы (поле **ves**), для класса **Tkany** (ткани) – это метры (поле **dlina**). Метод **Calc** (расчет) будет для каждого товара свой. Поэтому в базовом классе его надо объявить абстрактным (из-за чего и класс **Tovar** приходится делать абстрактным), а у каждого потомка реализовать по-своему. В то же время базовый абстрактный класс может содержать и обычные методы, например **PrintSum** (печать суммы), который наследуются всеми потомками без изменений.

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. Для реализуемых элементов интерфейса в классе следует указывать спецификатор public. К этим элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса.

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. Заметим, что в библиотеке классов **.NET** определено множество стандартных интерфейсов, декларирующих желаемое поведение объектов.

Пример 1

Реализация абстрактных классов и интерфейсов.

1. Создадим новый проект **con241**.
2. С помощью визуального конструктора классов спроектируем в отдельном файле абстрактный класс **Avto**, и в этом же файле классы-потомки **Vaz** и **Maz**, а также интерфейсы **ITurbo** и **IEco** в соответствии с рис. 2.8.

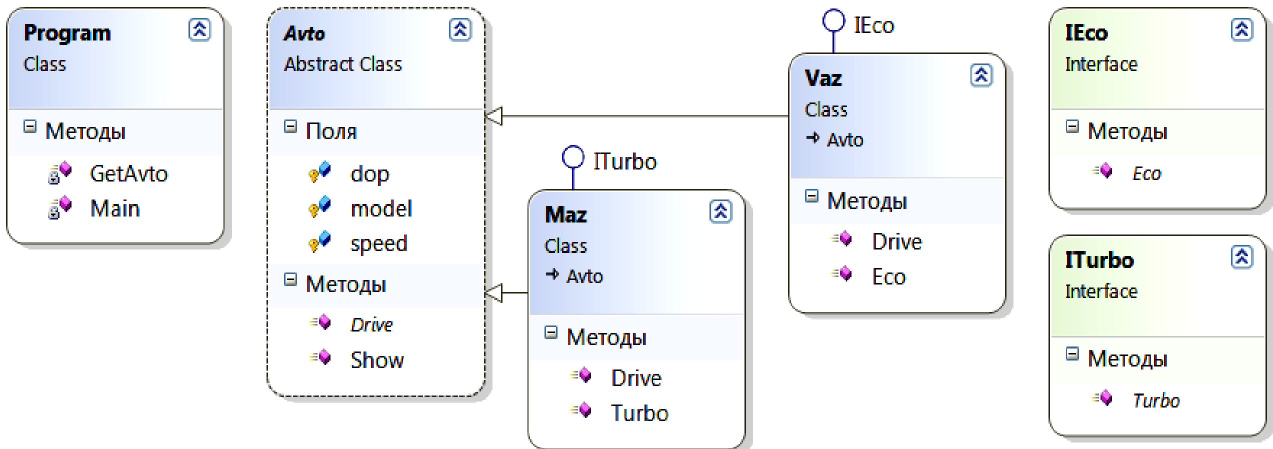


Рис. 2.8. Диаграмма классов

3. Введем в сгенерированные автоматически шаблоны программный код:

```
public abstract class Avto
{
 // объявляем поля с модификатором protected, доступные в классах-потомках
  protected string model; protected int speed;
  protected string dop;
  public abstract void Drive();          // объявляем абстрактный метод Drive
  public void Show()                     // объявляем обычный метод Show
   {  Console.WriteLine("модель {0}, скорость {1}, двигатель {2}",
                           model, speed, dop);     }
}
interface ITurbo  {  void Turbo();  }             // объявляем интерфейсы
interface IEco    {  void Eco();    }
           // класс Maz  наследует класс Avto и реализует интерфейс ITurbo
public class Maz : Avto, ITurbo
{ public override void Drive()      // переопределяем метод Drive
     { model = "Maz200"; speed = 90; Turbo(); }
  public void Turbo()  {  dop="турбо";  }      // реализуем интерфейс Turbo
}
           // класс Vaz  наследует класс Avto и реализует интерфейс IEco
public class Vaz : Avto, IEco
{ public override void Drive()// переопределяем  метод Drive
     { model = "vaz2107";  speed = 70; Eco(); }
```

```
  public void Eco()  {  dop = "экологичный"; }      // реализуем интерфейс Eco
}
class Program
{
  static void Main()
  {  Avto myAvto = GetAvto();                     // создание объекта myAvto
    myAvto.Drive(); myAvto.Show();                // вызов методов
    Console.ReadKey();
  }
  static Avto GetAvto()                           // метод выбора автомобиля
  {  Console.Write("Введите марку автомобиля (Vaz, Maz):   ");
     string mod = Console.ReadLine();
     switch (mod)
      {  case "Vaz": return new Vaz();
         case "Maz": return new Maz();
         default: return new Maz();
      }
  }
}
```

4. Протестируем программу, изменяя параметры.

Задания для самостоятельной работы

1. Модифицируйте проект **con241**, добавив еще один класс-потомок **BMV** (интерфейсы **Turbo, Eco**) и параметры автомобилей (мощность двигателя, расход топлива). Используя инструменты и методы визуального проектирования, создайте классы с потомками, содержащие указанные поля, конструкторы и методы. В классе **Program** создаются и инициализируются 2–3 объекта и указанными методами выводится информация. Продемонстрируйте результаты, изменяя параметры.

2. Спроектируйте абстрактный класс **Animal** с полями (**name, rost, ves**) и классы-потомки **Cats** и **Dogs**. Абстрактный метод **Golos** переопределяется для каждого вида (мяу-мяу и гав-гав).

3. Спроектируйте абстрактный класс **Figura** и классы–потомки: **Rectangle** (прямоугольник) и **Circle** (круг). Абстрактный метод вычисления площади **Area** переопределяется для каждой фигуры.

4. Спроектируйте абстрактный класс **Tovar** и классы-потомки: **Obuv** (обувь), **Odejda** (одежда), **Posuda** (посуда). Задайте общие (название, цена) и особенные (количество, размер, масса, цвет) поля и свойства. В классе **Tovar** задайте абстрактный метод **CalcSum** (расчет стоимости) и обычный **Print** (вывод всей информации). Абстрактный метод **CalcSum** переопределяется для каждого товара.

3. СОЗДАНИЕ WINDOWS-ПРИЛОЖЕНИЙ

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:

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

– **Программирование логики работы** приложения путем написания методов обработки событий.

Рассмотрим подробнее этапы разработки и структуру Windows-приложения.

– 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.**

– В результате открывается окно с формой в режиме конструктора (Design) (рис. 3.1). Слева по умолчанию располагается **Панель элементов** (Toolbox), а справа **Обозреватель решений** (Solution Explorer). При отсутствии их можно вызвать с помощью меню **Вид** (View).

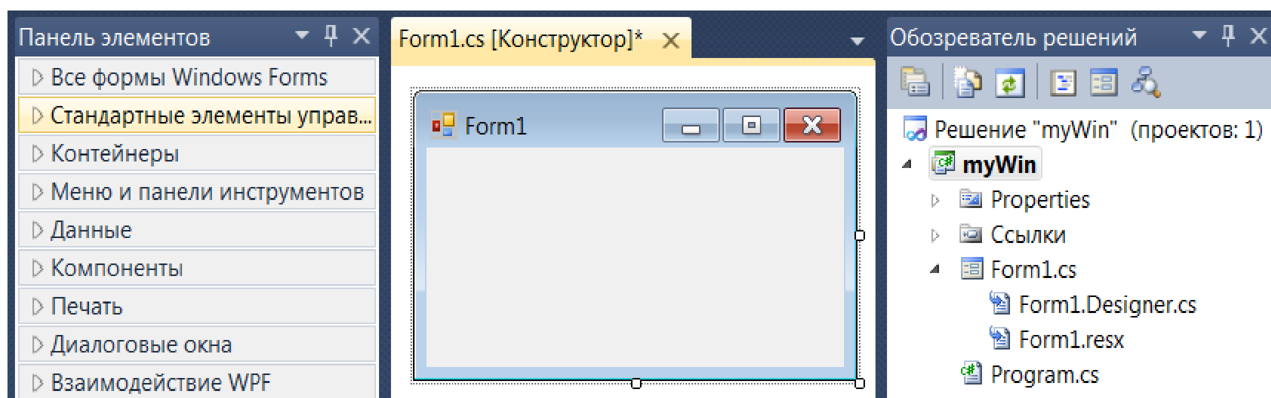


Рис. 3.1. Окно с формой в режиме конструктора

– Требуемые элементы (например, кнопка) перетаскиваются мышью с **Панели элементов** на форму (рис. 3.2). 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.

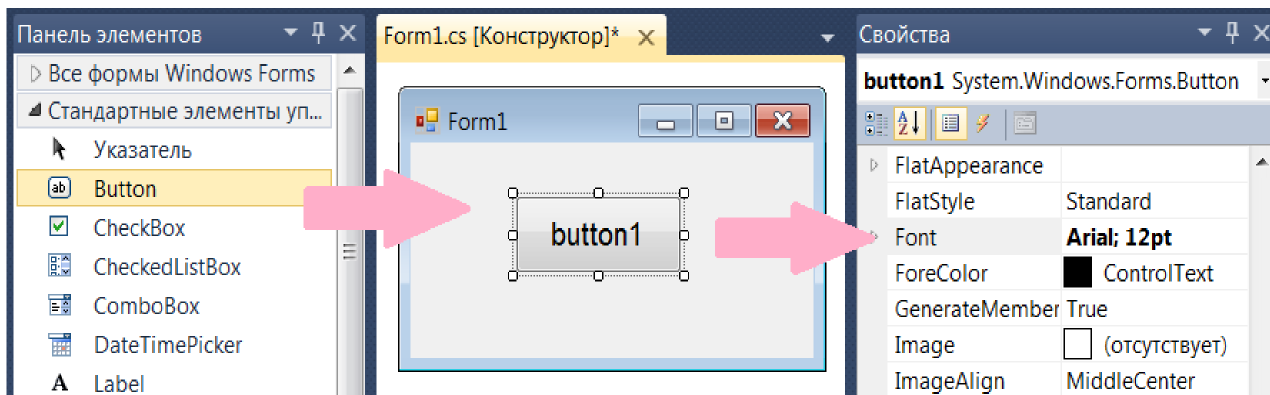


Рис. 3.2. Установка элемента управления и задание свойств

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

– Файл **Program.cs** содержит класс **Program**. Его метод **Main** является точкой входа, обеспечивает запуск приложения **Application.Run(new Form1())** и задает визуальный стиль (рис. 3.3). Для других целей при визуальном проектировании Windows-приложения его обычно **не** используют.

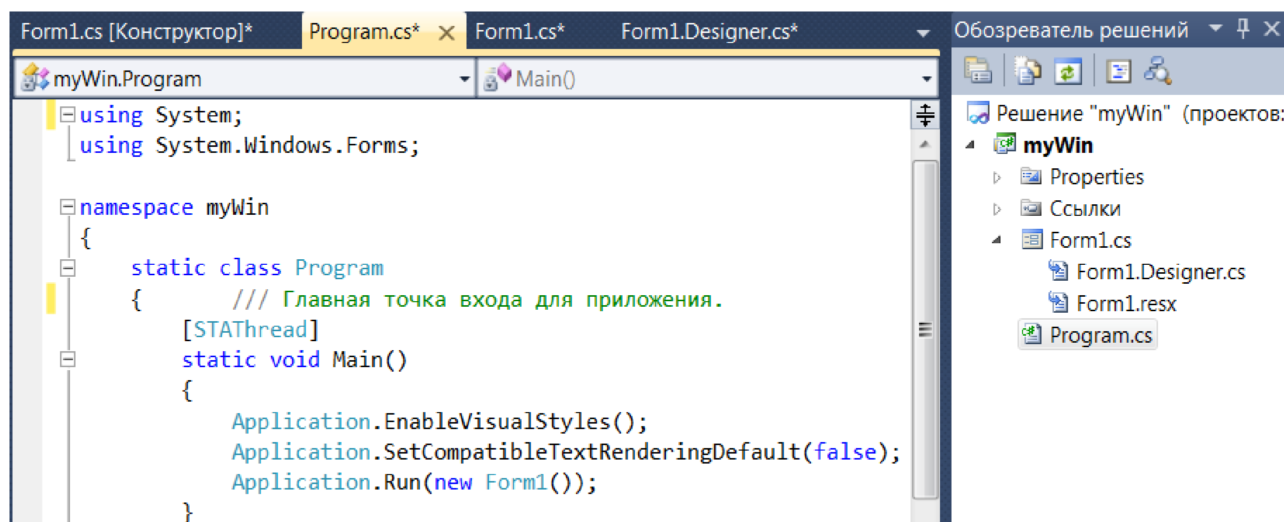


Рис. 3.3. Окна с кодом класса **Program**

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).

– Файл **Form1.cs** содержит часть класса **Form1** – конструктор с вызовом метода инициализации компонентов **InitializeComponent()** и обработчики событий (рис. 3.4). Именно в обработчиках событий программируется логика работы приложения.

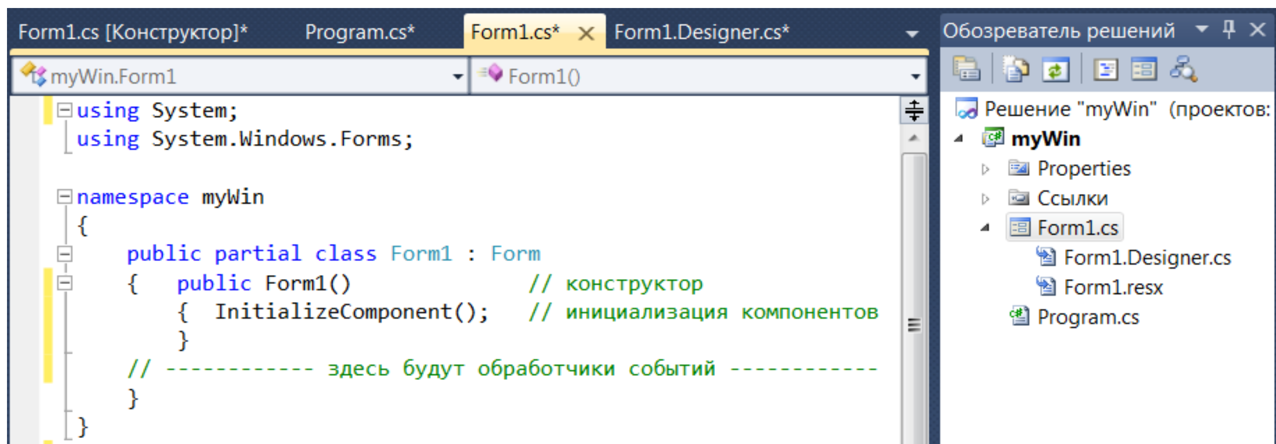


Рис. 3.4. Окно с программируемой частью кода класса **Form1**

– Файл **Form1.Designer.cs** в области #region … #endregion содержит код метода **InitializeComponent()**, автоматически создаваемый конструктором форм при установке элементов и регистрации событий (рис. 3.5).

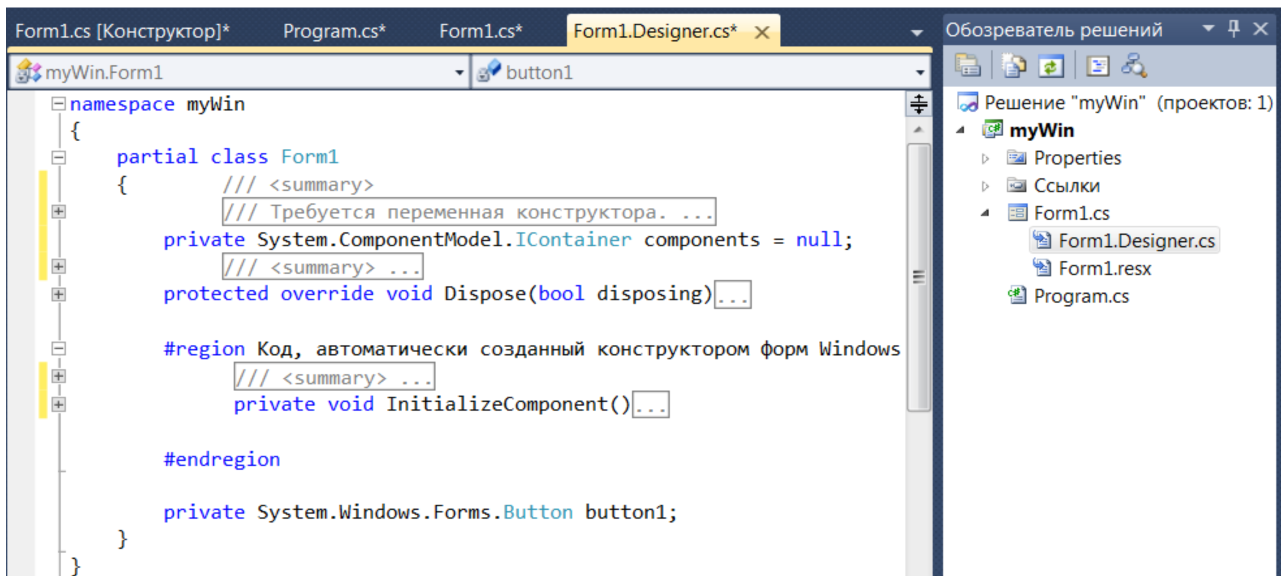


Рис. 3.5. Окно с автоматически создаваемым кодом класса **Form1**

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 **Свойства** двойным щелчком мыши на поле, расположенном справа от имени соответствующего события (рис. 3.6).

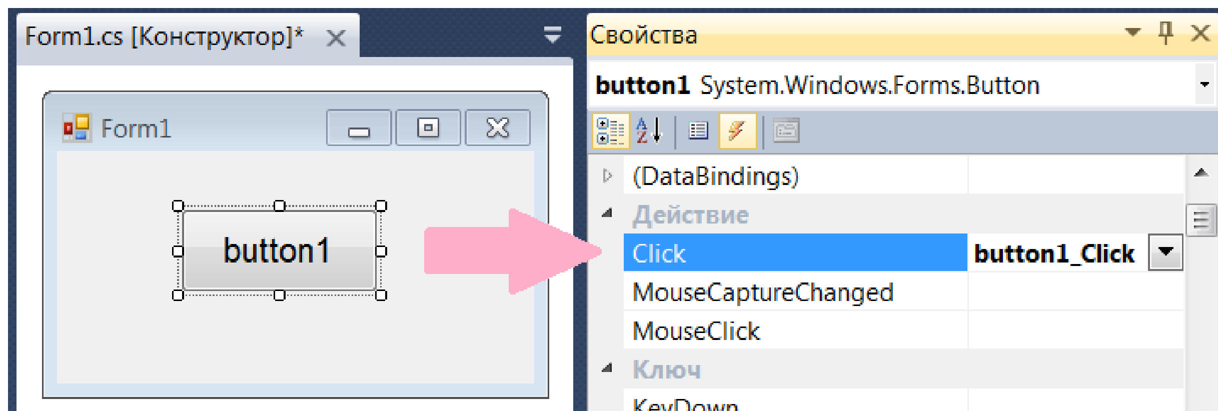


Рис. 3.6. Регистрация события нажатия кнопки Click

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

```
this.button1.Click += new System.EventHandler(this.button1_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 программирование логики работы приложения путем написания методов обработки событий.

3.1. Разработка приложений Windows Forms

Цель работы: формирование навыков создания приложений Windows Forms.

Введение

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**), задается 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 (рис. 3.7, *а*). По умолчанию это свойство равно Top, Left.

– **Dock** – прикрепление элементf к определенной стороне контейнера (рис. 3.7, *б*). По умолчанию имеет значение None.

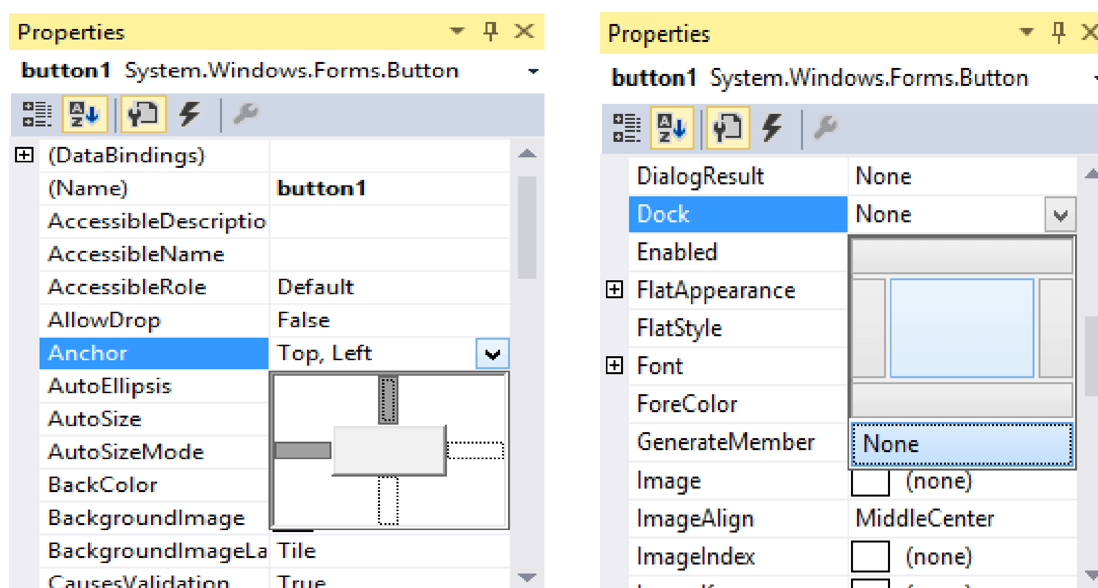


Рис. 3.7. Пример задания свойств Anchor (*а*) и Dock (*б*) для кнопки Button

Пример 1

Создать Windows-приложение, которое приглашает ввести в элемент TextBox имя и по нажатию кнопки выводит в элемент Label приветствие.

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: **wf311**.
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**» (рис. 3.8). Подберем размеры шрифта (свойства **Font** 10–12 пт.).

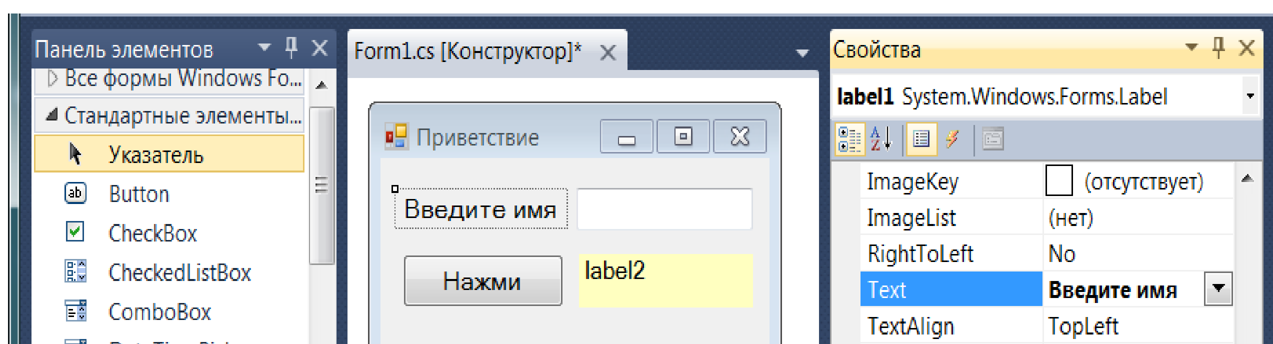


Рис. 3.8. Задание свойств элементов формы

7. Зарегистрируем событие нажатия кнопки. Для этого выделим кнопку и на вкладке **События** (**Events**) панели свойств выберем **Click**. Заметим, 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. Результат может выглядеть так (рис. 3.9, *а*). Откорректируем программный код и свойства элементов (например, зададим желтый фон **BackColor** надписи_2).
10. Модифицируем наш проект. Разместим на форме элемент **pictureBox1** (рис. 3.9, *б*). Импортируем изображение (свойство **Image**) из файла **hacker.jpg**. Подберем размеры и установим значения свойств **Visible = false** (в исходном

состоянии изображение невидимо) и **SizeMode** = **StretchImage** (растягивается по размеру контейнера).

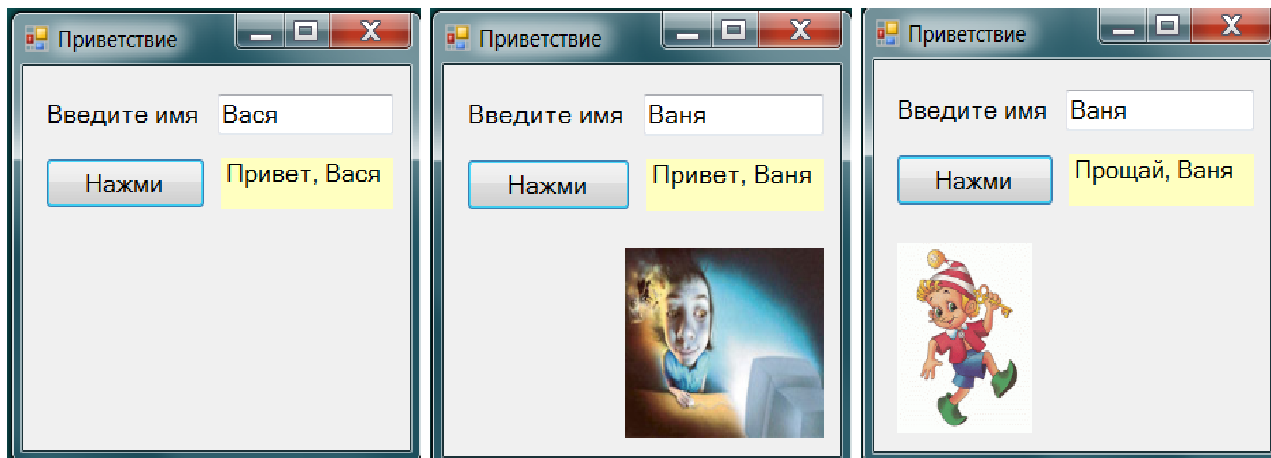


Рис. 3.9. Модификация интерфейса приложения

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 появлялась новое из файла **buratino.gif**, и надпись «**Привет ,**» заменялась на «**Прощай ,**» (рис. 3.9, *в*).

14. Разместим на форме элемент **pictureBox2** и импортируем в него изображение **buratino.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. Откорректируем код и свойства элементов.

Пример 2

*Простейший тест с использованием элементов **radioButton.** Выбор единственного верного ответа.*

1. Создадим новый проект **wf312** типа Windows Forms.

2. Разместим на форме надпись **label1** с вопросом: «Что такое компьютерный вирус?», три элемента **radioButton** с ответами «Вредный программист», «Вредоносная программа», «Живущий в компьютере микроб», а также кнопку **button1** «ответ» и поле **textBox1** для вывода результата (рис. 3.10).

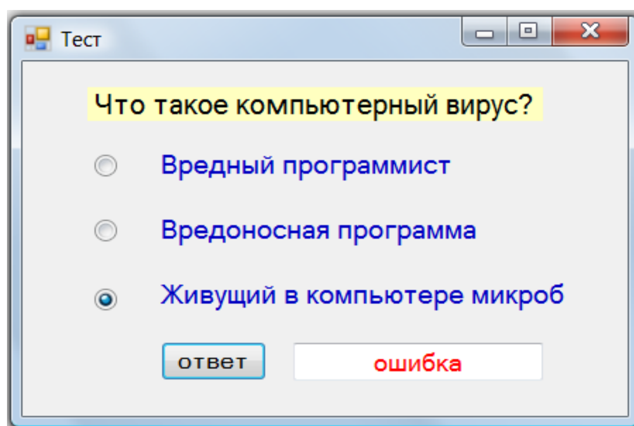


Рис. 3.10. Интерфейс теста с элементами **radioButton**

3. Зарегистрируем событие нажатия кнопки. В шаблон обработчика введем код:

```
private void button1_Click(object sender, EventArgs e)
 {  textBox1.Text = (radioButton2.Checked) ? "верно" : "ошибка";
 }
```

Пример 3

*Тест с использованием элементов **checkBox.** Выбор нескольких верных ответов.*

1. Создадим новый проект **wf313** типа Windows Forms.
2. Разместим на форме надпись **label1** «Устройства ввода:», четыре элемента **checkBox** с текстами ответов «клавиатура», «рука», «мышь», «кошка», кнопку «ответ» и надпись **label12** для вывода результата (рис. 3.11).

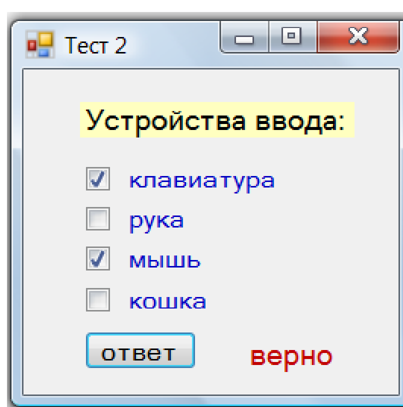


Рис. 3.11. Интерфейс теста с элементами **checkBox**

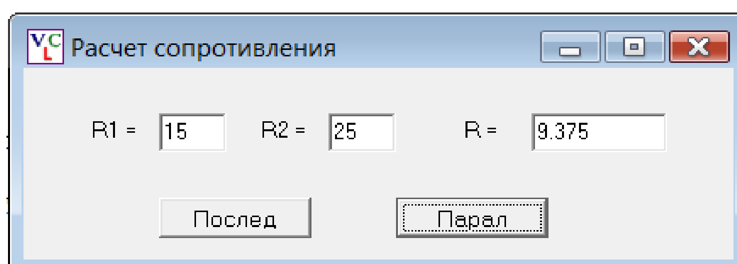
3. Зарегистрируем событие нажатия кнопки. В шаблон обработчика введем код:

```
private void button1_Click(object sender, EventArgs e)
 {  label2.Text = (checkBox1.Checked && checkBox3.Checked &&
    !checkBox2.Checked && !checkBox4.Checked) ? "верно" : "ошибка";
 }
```

Задания для самостоятельной работы

Создайте Windows-приложения, в которых выполняются следующие действия.

1. При щелчке мыши по форме показывается изображение автомобиля avto.gif. Щелчок мышью по изображению скрывает его.
2. При наведении указателя мыши на форму цвет ее фона становится желтым. При уводе указателя мыши с формы ее цвет становится серым.
3. При нажатии и удержании левой кнопки мыши на форме ее цвет становится розовым. Отпускание кнопки мыши изменяет цвет фона на голубой.
4. По нажатию кнопки проверяется правильность введенного в текстовое поле пароля. При совпадении с заданным ключом выводится «Добро пожаловать» и появляется изображение Буратино (файл buratino.gif), при несовпадении выводится «Вход запрещен» и изображение попугая (файл popugai.gif).
5. Вычисляется среднее арифметическое **sred** двух введенных чисел **a** и **b**. Ввод в текстовые поля, вывод по нажатию кнопки в надпись.
6. Вычисляется площадь **s** и периметр **p** прямоугольника по сторонам **a** и **b**. Ввод в текстовые поля, вывод по нажатию кнопки в надписи.
7. Вычисляется сопротивление при последовательном или параллельном соединении резисторов. Ввод и вывод в текстовые поля по нажатию кнопок.



8. *Создайте калькулятор, выполняющий 5 действий. Ввод и вывод в текстовые поля. Счет по нажатию кнопок.

3.2. Интерактивное управление параметрами приложений

Цель работы: формирование навыков создания Windows-приложений с интерактивным управлением параметрами.

Введение

Библиотека классов .NET содержит элементы, которые можно использовать для управления параметрами Windows-приложений в интерактивном режиме без текстового ввода. Приведем примеры таких элементов и событий, позволяющих отслеживать изменения их свойств.

Элемент **TrackBar** позволяет с помощью ползунка изменять числовые значения свойства **Value** типа **int** из диапазона Minimum, Maximum. Событие **Scroll.**

NumerickUpDown позволяет изменять значение **Value** числового типа **decimal** (десятичная дробь с разделителем, заданным локализацией операционной системы, например, запятая в русскоязычных ОС). Событие **ValueChanged.** Приращение **Increment** может как целым, так и дробным. Отображаемое количество десятичных знаков задается **DecimalPlaces** (по умолчанию **0**).

Следующие три элемента предназначены для выбора из списков.

– **DomainUpDown** возвращает строку (свойства **Text** типа **string)**. Событие **TextChanged**.

– **ListBox** (список) и **ComboBox** (поле с выпадающим списком) возвращают индекс (свойство **SelectedIndex**). Событие **SelectedIndexChanged**.

Пример 1

*Вычисление площади круга. Использование элемента **trackBar**.*

1. Создадим проект **wf321** типа Windows Forms.
2. Разместим на форме две надписи с текстами **“R = ”** и **“S = ”**, два текстовых поля и элемент **trackBar** (рис. 3.12)**.** Установим его свойства Minimum = 10, Maximum = 80.

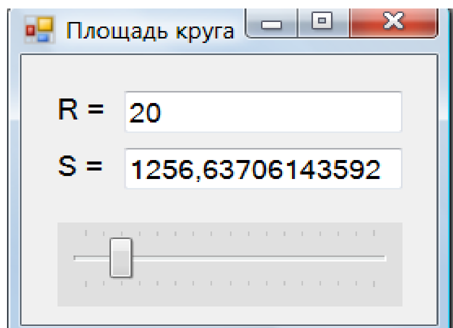


Рис. 3.12. Интерфейс приложения с элементами **label, textBox** и **trackBar**

3. Выделим элемент **trackBar.** Зарегистрируем событие перемещения ползунка **Scroll.** В шаблон обработчика введем код:

```
private void trackBar1_Scroll(object sender, EventArgs e)
  {  int r = trackBar1.Value;
     textBox1.Text = Convert.ToString(r);
     textBox2.Text = Convert.ToString(Math.PI*r*r);
  }
```

4. Протестируем программу. Откорректируем код и свойства элементов.

Пример 2

*Расчет силы тока. Использование элемента **numericUpDown**.*

*Напряжение и сопротивление будем задавать элементами **numericUpDown**, силу тока вычислять по формуле **I = U/R** и выводить в текстовое поле по нажатию кнопки.*

1. Создадим проект **wf322** типа Windows Forms.

2. Разместим на форме две надписи с текстами **“U = ”** и **“R = ”,** кнопку **“I = ”**, текстовое поле и два элемента **numericUpDown** (рис. 3.13). Установим их свойства Minimum = 1, Maximum = 20, Increment = 1.

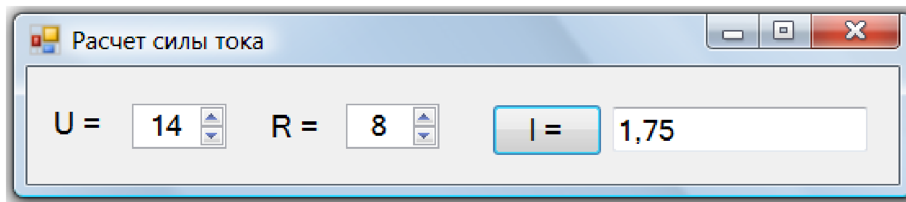


Рис. 3.13. Интерфейс приложения с кнопкой и элементами **numericUpDown**

3. Зарегистрируем событие нажатия кнопки. В шаблон обработчика введем код:

```
private void button1_Click(object sender, EventArgs e)
 {  decimal U = numericUpDown1.Value;  decimal R = numericUpDown2.Value;
    textBox1.Text = Convert.ToString(U/R);
 }
```

4. Протестируем программу, изменяя напряжение U и сопротивление R. Заметим, что хотя параметры U и R задаются элементом **numericUpDown,** расчет и вывод результата выполняется по нажатию кнопки.

Во многих практических задачах вычисления и вывод результатов полезно выполнять непосредственно при изменении входных параметров. Для этого используют события, отслеживающие изменения свойств элементов. Так, в примере 1 использовано событие **Scroll** элемента **trackBar.** Рассмотрим использование двух событий **ValueChanged** элементов **numericUpDown.**

Пример 3

*Расчет сопротивления при параллельном соединении резисторов. Использование событий двух элементов **numericUpDown.***

*При параллельном соединении резисторов складываются величины, обратные их сопротивлениям **1/R = 1/R1 + 1/R2**.*

1. Создадим проект **wf323** типа Windows Forms.

2. Разместим на форме три надписи **“R1 = ”, “R2 = ”, “R = ”,** текстовое поле и два элемента **numericUpDown** (рис. 3.14). Установим свойства: Minimum = 1, Maximum = 80, Increment = 0,1, DecimalPlaces = 1.

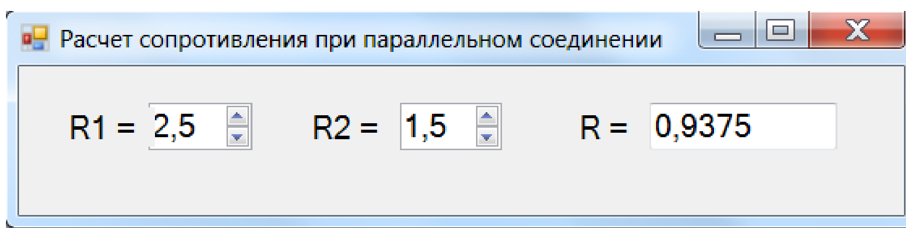


Рис. 3.14. Интерфейс приложения с элементами **numericUpDown**

3. Зарегистрируем события **ValueChanged** двух элементов **numericUpDown**.

4. В шаблоны обработчиков введем коды. Для их упрощения вне обработчиков объявлены и инициализированы поля r1 и r2, а вычисление и вывод вынесены в метод **ShowR()**, который вызывается в обработчиках.

```
public decimal r1 = 1;  public decimal r2 = 1;
public void ShowR()
  {  textBox1.Text = Convert.ToString(1/(1/r1 + 1/r2));  }
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
  {  r1 = numericUpDown1.Value;  ShowR();  }
private void numericUpDown2_ValueChanged(object sender, EventArgs e)
  {  r2 = numericUpDown2.Value;  ShowR();  }
```

5. Протестируем программу. Теперь при любом изменении сопротивлений резисторов результат сразу же пересчитывается.

Задания для самостоятельной работы

Создайте приложения Windows Forms, которые вычисляют и выводят:

1. Сопротивление при последовательном (параллельном) соединении резисторов. Вывод в надписи по событиям **Scroll** элементов **trackBar**.

2. Площадь поверхности **s** и объем шара **v** по радиусу **r**. Ввод и вывод в текстовые поля по события **Scroll** элемента **trackBar**.

3. Высоту $\mathbf{h = V^2/2g}$ подъема мяча, брошенного вертикально вверх с начальной скоростью **V** (задается элементом **trackBar**). Вывод в надпись по нажатию кнопки.

4. Потенциальную энергию **E = mgh** камня массой **m** (ввод в текстовое поле) на высоте **h** (задается элементом **trackBar**). Вывод в надпись по нажатию кнопки.

5. Путь **s = v*t**, пройденный автомобилем за время **t**. Скорость **v** вводится в текстовое поле, время **t** (10 – 40) задается элементом **trackBar.** Вывод в надпись по событию **Scroll** элемента **trackBar**.

6. Оплату за электроэнергию = тариф * расход. Вывод в надпись по событию **Scroll.** Тариф (в руб за 1 кВт ч) задается элементом **numericUpDown.** Расход в кВт ч (от 0 до 400) задается элементом **trackBar**,

7. Стоимость поездки на автомобиле (*ввод*: **s** – расстояние, **b** – расход бензина на 100 км, **c** – цена бензина за 1 литр). Вывод в надпись по события **Scroll** элемента **trackBar.**

8. Стоимость товара в трех валютах по его стоимости в бел. рублях. Ввод в **textBox** (бел. руб.), вывод по нажатию кнопки в надписи. Курсы валют задаются элементами **numericUpDown.**

3.3. Использование таймера. Анимация

Цель работы: формирование навыков использования таймера и создания простейшей анимации движения.

Введение

Неотображаемый на форме компонент **Timer** предназначен для запуска периодически повторяющихся действий. Свойство **Interval** задает период (в миллисекундах), с которым будет повторяться событие **Tick**. При установке свойства **Enabled = true** таймер включается вместе с запуском приложения. Метод **Start()** запускает, а **Stop()** останавливает таймер.

Типичные примеры использования таймера – часы и секундомер (вывод времени и даты), а также анимация (имитация плавного изменения положения, размеров и формы объектов).

Пример 1

Простые часы. Вывод времени и даты по таймеру.

1. Создадим новый проект **wf331** типа Windows Forms.

2. Разместим на форме размером 540 × 230 две надписи и зададим их свойства:

– **label1** для вывода времени (свойства: **BackColor = Green**, **ForeColor = = Yellow**, **Text = 00:00:00**, размер шрифта **Font = 72)**;

– **label2** для вывода даты (**ForeColor = Olive**, **Text = дата**, **Font = 16**).

3. Из категории **Компоненты** (Components) панели элементов перетащим на форму **Timer**. Его значок отобразится в нижней части окна **Конструктор** (рис. 3.15**).** Зададим его свойства: **Enabled = true, Interval = 100.**

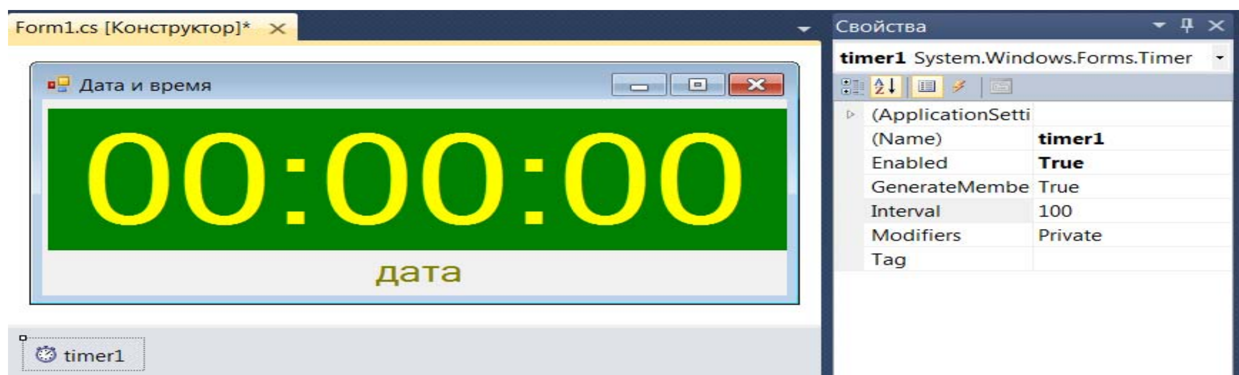


Рис. 3.15. Настройка элементов формы и таймера

4. Зарегистрируем событие **Tick** таймера. В шаблон обработчика введем код:

```
private void timer1_Tick(object sender, EventArgs e)
 {  label1.Text = DateTime.Now.ToLongTimeString();      // вывод времени
    label2.Text = DateTime.Now.ToLongDateString();      // вывод даты
 }
```

5. Протестируем программу (рис. 3.16, *а*). При необходимости откорректируем свойства компонентов и программный код.

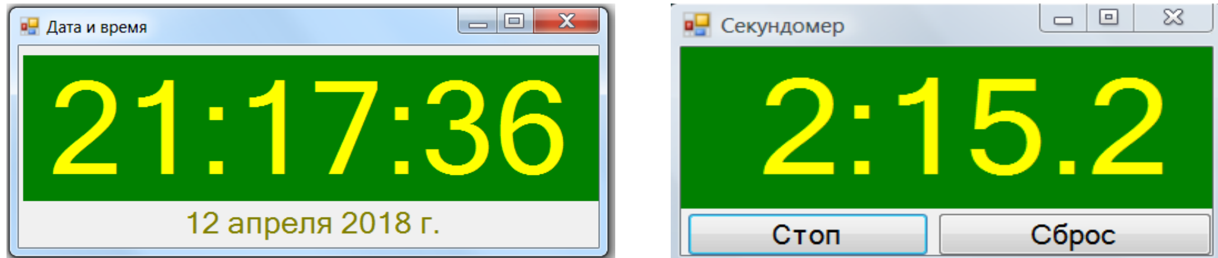


Рис. 3.16. Простые часы (*а*) и секундомер (*б*)

Пример 2

Секундомер. Вывод минут и секунд.

1. Создадим новый проект **wf332** типа Windows Forms**.**

2. Разместим на форме размером 380×200 надпись **label1** (свойства **BackColor = Green, ForeColor = Yellow, Text = 0:0.0, Font = 60**) и две кнопки **Старт** и **Сброс** (рис. 3.16, *б*).

3. Перетащим на форму **Timer** (свойства **Enabled = false, Interval = 100)**.

4. Зарегистрируем события **Tick** таймера, а также нажатий кнопок **Click**. В шаблоны обработчиков введем коды:

```
private int m, s, ms;                              // объявление полей
      // нажатие кнопки Старт/Стоп
private void button1_Click(object sender, EventArgs e)
 { if (timer1.Enabled)
    {  timer1.Stop();  button1.Text = "Старт";  }    // остановка таймера
   else {  timer1.Start();  button1.Text = "Стоп";  } // запуск таймера
 }
      // нажатие кнопки Сброс, обнуление значений
private void button2_Click(object sender, EventArgs e)
 {  m = 0; s = 0; ms = 0;  label1.Text = "0:0.0"; }
      // счет и вывод по событию Tick таймера
private void timer1_Tick(object sender, EventArgs e)
 {  ms++;  s = ms/10;  m = s/60;
    label1.Text = m + ":" + s%60 + "." + ms%10;
 }
```

5. Протестируем программу. Откорректируем код и свойства компонентов.

Пример 3

Простейшая анимация движения.

Создадим новый проект **wf333** типа Windows Forms.

Разместим на форме размером 580 × 200 две кнопки **Старт** и **Стоп**, а также элемент **PictureBox** размером 140 × 100. Импортируем в него изображение из файла **beg.gif** (рис. 3.17, *а*).

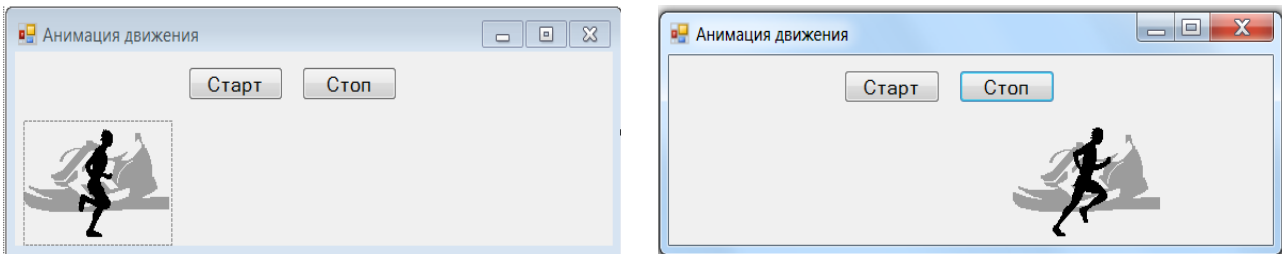


Рис. 3.17. Начальная (*а*) и промежуточная фазы анимации (*б*)

3. Перетащим на форму **Timer** (свойства **Enabled = false, Interval = 20)**.

4. Зарегистрируем события **Tick** таймера, а также нажатий кнопок **Click**. В шаблоны обработчиков введем коды:

```
// перемещение по тикам таймера вправо на 4px, если Left <520, иначе в начало
private void timer1_Tick(object sender, EventArgs e)
 {  if (pictureBox1.Left < 520) pictureBox1.Left += 4;
    else pictureBox1.Left = 8; }
private void button1_Click(object sender, EventArgs e)
 {  timer1.Enabled = true;  }                          // старт
private void button2_Click(object sender, EventArgs e)
 {  timer1.Enabled = false;  }                         // стоп
```

5. Протестируем программу.

Задания для самостоятельной работы

1. Создайте секундомер с одной кнопкой **Сброс**, который запускается и останавливается щелчком мыши по форме.

2. Создайте приложение, которое по введенной дате (год, месяц и число) показывает день недели.

3. Создайте часы, которые показывают, сколько дней (часов, минут) осталось наступления нового года.

4. Создайте приложение, в котором анимируется прямолинейное движение спутника **sputnik.jpg** на фоне звездного неба **sky.gif**.

5. Создайте приложение, в котором анимируется падение яблока **apple.gif** с башни **tower.jpg**.

6. Создайте приложение, в котором по щелчку мышью по изображению совы **sova.jpg** оно начинает увеличиваться до достижения двукратного размера. Щелчок мыши по увеличенному изображению вызывает его уменьшение до первоначальных размеров.

7. Создайте приложение, в котором анимируется движение мяча **football.gif**, брошенного под углом к горизонту на фоне деревьев **trees.gif**. Траектория движения парабола.

8. Создайте приложение, в котором анимируется движение Луны **luna.gif** вокруг Земли **zem.gif** по эллиптической траектории.

3.4. Использование меню и диалоговых окон

Цель работы: формирование навыков создания меню и диалоговых окон.

Введение

Библиотека .NET содержит компоненты позволяющие проектировать приложения с развитым графическим интерфейсом, включающим меню, панели инструментов, разнообразные диалоговые окна. Некоторые из них сразу отображаются на форме, другие могут вызываться.

Основные компоненты для построения меню:

ToolStrip – панель инструментов (базовый класс – контейнер).

На ней размещают элементы – объекты **ToolStripItem**.

Наследник **ToolStrip** – полоска меню **MenuStrip**.

StatusStrip – строка (полоска) состояния.

Основные свойства элементов меню: вид, размеры, позиционирование (Dock, LayoutStyle), видимость (ShowItemToolTips). Основные события связаны с выбором конкретного пункта меню щелчком мыши и имеют синтаксис:

пунктToolStripMenuItem_Click.

Важнейшей особенностью Windows-приложений является **оконный интерфейс**, включающий окна разного типа, имеющие свое назначение, функционал и внешний вид. Каждое приложение имеет одно **главное окно**. Напомним, что класс главного окна приложения содержит точку входа в приложение (статический метод **Main**). При закрытии главного окна приложение завершается. **Модальное окно** не позволяет пользователю переключаться на другие окна того же приложения, пока не будет завершена работа с текущим окном. **Немодальное окно** позволяет переключаться на другие окна. В виде модальных обычно оформляют **диалоговые окна**, требующие от пользователя ввода какой-либо информации или подтверждения.

Наиболее часто используют **Окно сообщений** (класс **MessageBox**), которое вызывается методом **Show()** и может принимать ряд параметров (рис 3.18): **text** – текст выводимого сообщения, **caption** – текст заголовка окна сообщения, **icon** – значок окна сообщения, **buttons** – используемые кнопки.

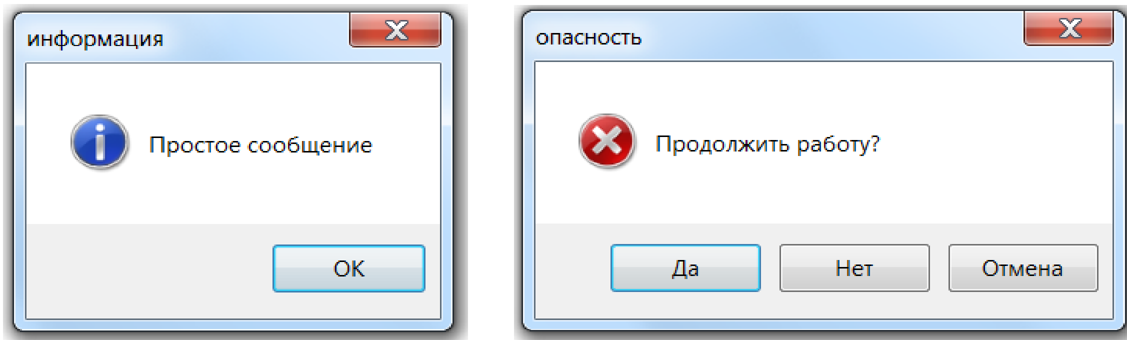


Рис. 3.18. Вид диалогового окна сообщений **MessageBox**

Это окно может иметь от 1 до 3 кнопок (рис. 3.18). Количество и назначение задается значением свойства MessageBoxButtons: **OK** – одна кнопка OK; **OKCancel** – две кнопки (OK и Отмена); **YesNo** – две кнопки (Да и Нет); **YesNoCancel** – три кнопки (Да, Нет, Отмена). Метод **Show()** возвращает объект DialogResult, позволяющий узнать, какая кнопка в окне была нажата. Вид значков в соответствии с содержанием сообщения можно задавать значением свойства MessageBoxIcon: **Information** – буква i в кружке; **Error** – белый знак «X» в красном круге; **Warning** – восклицательный знак в желтом треугольнике; **Question** – вопросительный знак в круге.

Выполнение многошаговых операций значительно упрощается благодаря использованию **специализированных** диалоговых окон (или просто диалогов), имеющих набор требуемых свойств, методов и событий. Эти окна отображаются методом **ShowDialog().** Приведем примеры таких диалогов.

– **ColorDialog** – выбор цвета (рис. 3.19, *а*). Возвращает выбранный цвет:
colorDialog1.ShowDialog(); this.BackColor = colorDialog1.Color;

– **FontDialog** – выбор шрифта (рис. 3.19, *б*). Возвращает выбранный шрифт:
fontDialog1.ShowDialog();   textBox1.Font = fontDialog1.Font;

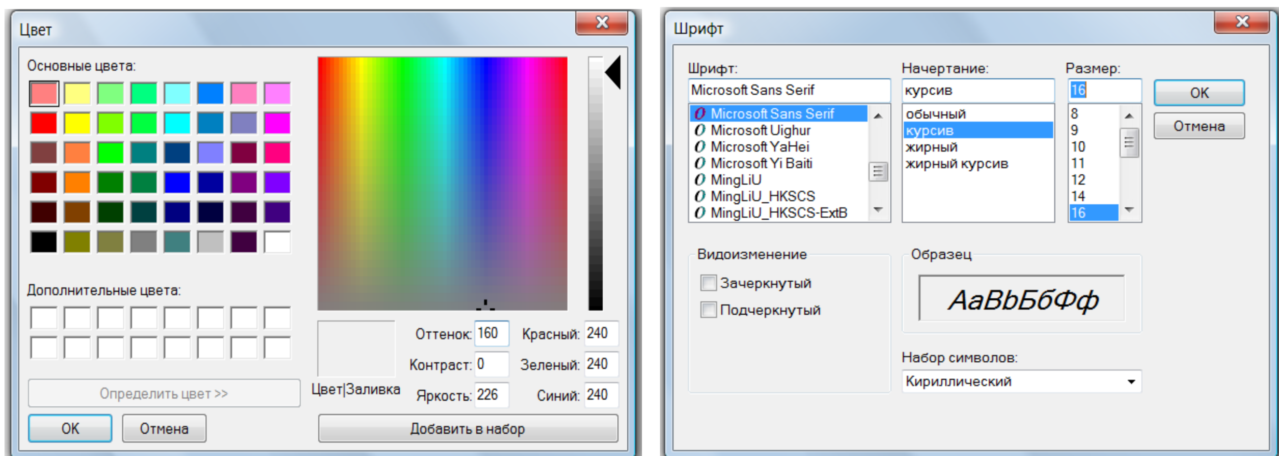


Рис. 3.19. Вид диалоговых окон **ColorDialog** (*а*) и **FontDialog** (*б*)

Диалоги **OpenFileDialog** (открытие файла) и **SaveFileDialog** (сохранение файла) возвращают дескрипторы выбранного файла, например:
string fn = openFileDialog1.FileName; или string fn = saveFileDialog1.FileName;

Пример 1

Создание простейшего текстового редактора. Шрифт и цвет текста изменяется с помощью меню.

1. Создадим новый проект **wf341** типа Windows Forms.
2. Разместим на форме размером 480×360 текстовое поле **textBox1** (свойства: **Multiline = true, ScrollBars = Vertical, Anchor** = Top, Bottom, Left, Right).
3. Перетащим на форму компоненты **ColorDialog, FontDialog** и **MenuStrip**.
4. Выделим компонент **MenuStrip.** Пользуясь подсказками, создадим два пункта меню: **вид** (с подпунктами **шрифт, цвет**) и **справка** (рис 3.20).

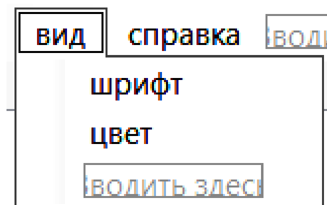


Рис. 3.20. Создание меню

5. Поочередно выделяем созданные пункты и регистрируем события **Click.**
6. В шаблоны обработчиков выбора пунктов меню **шрифт** и **цвет** введем коды, вызывающие диалоги задания шрифта и цвета текста:

```
private void шрифтToolStripMenuItem_Click(object sender, EventArgs e)
 {  fontDialog1.ShowDialog();          // вызов диалога задания шрифта
    textBox1.Font = fontDialog1.Font;
 }
private void цветToolStripMenuItem_Click(object sender, EventArgs e)
 {  colorDialog1.ShowDialog();         // вызов диалога задания цвета
     textBox1.ForeColor = colorDialog1.Color;
 }
```

7. В шаблон обработчика выбора пункта меню **Справка** введем код вызова окна сообщений **MessageBox**:

```
private void справкаToolStripMenuItem_Click(object sender, EventArgs e)
 {  MessageBox.Show("текстовый редактор \n разработал: студент");
 }
```

8. Протестируем программу, вводя текст и изменяя цвет и шрифт. Результат может выглядеть так (рис. 3.21, *а*, *б*).

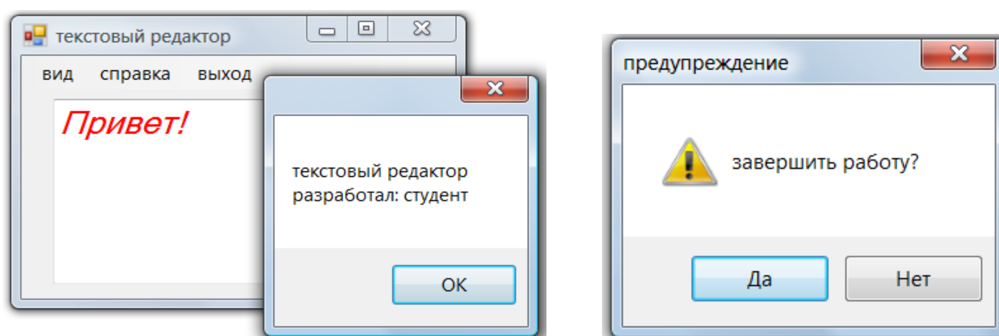


Рис. 3.21. Вид редактора (*а*) и окон сообщений Справка (*б*) и Выход (*в*)

9. Добавим пункт меню **выход**. Зарегистрируем событие **Click.** В шаблон обработчика введем код вызова окна сообщений с двумя кнопками (Да – Нет) и иконкой предупреждения (рис. 3.21, *в*):

```
private void выходToolStripMenuItem_Click(object sender, EventArgs e)
 {  DialogResult res = MessageBox.Show("завершить работу?", "предупреждение",
                      MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
    if (res == DialogResult.Yes)  Application.Exit();
 }
```

10. Протестируем окончательный вариант.

Пример 2

Открытие и сохранение текстовых файлов.

1. Продолжим модифицировать простейший текстовый редактор (проект **wf341**). Добавим возможность открывать и сохранять текстовые файлы.

2. Для этого в начале файла с программным кодом **Form1.cs** подключим пространство имен **System.IO.**

3. Перетащим на форму компоненты: **openFileDialog** (открытие) и **SaveFileDialog** (сохранение файла).

4. Добавим в меню пункт **файл** с подпунктами **открыть** и **сохранить** и зарегистрируем для них события **Click.**

5. В шаблоны обработчиков введем коды вызова диалогов открытия и сохранения файлов:

```
private void открытьToolStripMenuItem_Click(object sender, EventArgs e)
 {  openFileDialog1.FileName = string.Empty;
     if (openFileDialog1.ShowDialog() == DialogResult.OK)
      {  string fn = openFileDialog1.FileName;
         this.Text = "открыт файл " + fn;
         try {  StreamReader sr = new StreamReader(fn);
                textBox1.Text = sr.ReadToEnd();  sr.Close();
             }
        catch (Exception ex)
            {  MessageBox.Show("Ошибка чтения \n" + ex.ToString());  }
      }
 }
private void сохранитьToolStripMenuItem_Click(object sender, EventArgs e)
 {  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
     {  string fn = saveFileDialog1.FileName;
        this.Text = "сохранен файл " + fn;
        if (fn != string.Empty)
         {  FileInfo fi = new FileInfo(fn);
            try {  StreamWriter sw = fi.CreateText();
                   sw.Write(textBox1.Text); sw.Close();
                }
            catch (Exception ex)
                { MessageBox.Show("Ошибка записи \n" + ex.ToString());  }
         }
     }
 }
```

6. Протестируем программу. Откроем текст из файла **stroki.txt** (рис. 3.22, *а*).

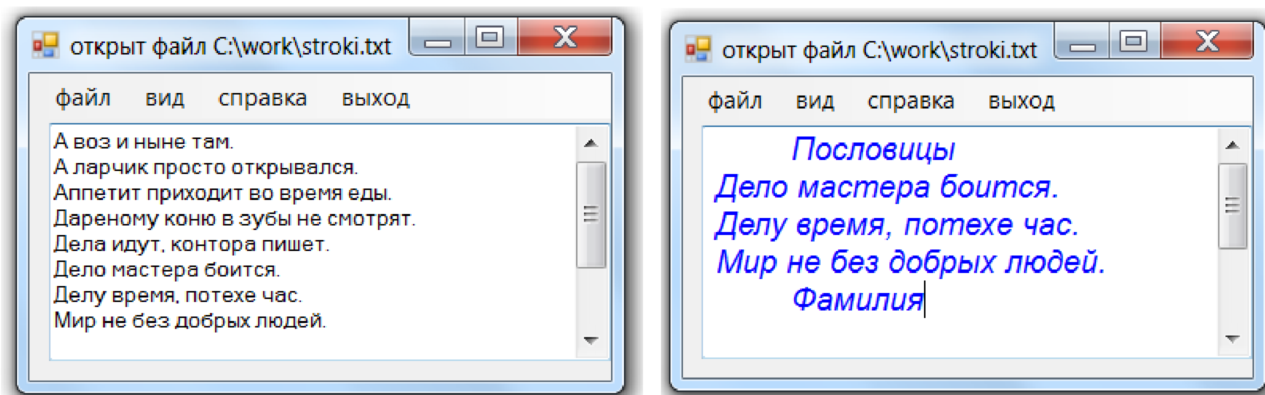


Рис. 3.22. Фрагмент текста до (*а*) и после редактирования (*б*)

7. Отредактируем и оформим текст: удалим первые 5 строк, вставим заголовок «Пословицы», а в конце свою фамилию; шрифт Arial, 12 пт, курсив, цвет синий (рис. 3.22, *б*). Сохраним файл под именем **stroki2.txt.**

Задания для самостоятельной работы

1. Создайте приложение, в котором двойным щелчком мыши на форме вызывается компонент **colorDialog**, позволяющий изменять цвет формы.

2. Создайте приложение, в котором по нажатию кнопки вызывается компонент **openFileDialog**, позволяющий выбирать и загружать в **pictureBox** изображение из файла.

3. Создайте регистрационную форму, в которую вводится имя и город. Данные можно записывать в файл **sam3.txt** и читать по нажатию кнопок.

Запись в файл и чтение

Введите имя Вася

Введите город Минск

Записать Отмена Прочитать

имя: Иван

класс: 9 10 11

предмет: физика

Записать Отмена Прочитать

4. *Создайте регистрационную форму, в которой вводится имя, с помощью **radioButton** выбирается класс, из списка **comboBox** выбирается предмет (физика, математика, информатика). Данные можно записывать в файл **reg.txt** и читать из файла по нажатию кнопок.

5. *Создайте регистрационную форму, в которой в текстовые поля вводят **имя, логин, пароль** и **e-mail**. Корректность ввода проверяется с помощью регулярных выражений. Если введены верные данные, они сохраняются в файле **log.txt**, иначе выводится сообщение «Повторите ввод».

3.5. Работа с графикой GDI+

Цель работы: формирование навыков работы с графикой в среде MS Visual Studio.

Введение

Для работы с графикой в среде .NET предназначен класс **Graphics** пространства имен **System.Drawing.** Для вывода графических примитивов (линий, геометрических фигур), текста, растровых изображений необходимо создать объект класса **Graphics**, например, методом **CreateGraphics**:

Graphics g = this.CreateGraphics();

После создания объекта типа **Graphics** можно применять его свойства и методы. Наиболее часто используются объекты и методы классов **Pen** (рисование линий и геометрических фигур), **Brush** (заполнение областей), **Font** (работа с текстом), **Color** (работа с цветом).

Для интерактивного управления свойствами графических объектов удобно использовать манипулятор мышь. События мыши **MouseDown, MouseUp, MouseMove** и другие работают в сочетании с делегатом **MouseEventHandler**.

Например, при регистрации события движения мыши по форме в методе **InitializeComponent()** (в файле Form1.Designer.cs) появляется строка:

```
MouseMove += new MouseEventHandler(Form1_MouseMove);
```

При этом в файле кода Form1.cs создается шаблон метода-обработчика, которому передаются два параметра: объект-источник события и объект класса **MouseEventArgs**, который содержит информацию о событии, например: **X** и **Y** – координаты указателя мыши; **Button** – нажатая кнопка (левая, правая); **Clicks** – количество нажатий и отпусканий кнопки мыши; **Delta** – счетчик (со знаком) щелчков поворота колесика.

Эту информацию можно использовать в обработчике, например, для вывода координат в заголовок формы:

```
public void Form1_MouseMove(object sender, MouseEventArgs e)
  {  Text = string.Format("координаты: x={0}, y={1}", e.X, e.Y);
  }
```

Для управления графическими объектами нередко используют и клавиатуру. События клавиатуры **KeyUp, KeyDown** работают в сочетании с делегатом **KeyEventHandler.**

Например, при регистрации события нажатия клавиши записывается строка

```
KeyDown += new KeyEventHandler(Form1_KeyUp);
```

Создается шаблон метода-обработчика, которому передаются: объект-источник события и объект класса **KeyEventArgs**, который содержит информацию о событии, например: **KeyCode** – код клавиши для событий KeyDown или KeyUp; **Modifiers** – какие модифицирующие клавиши (Shift, Alt, Control) были нажаты; **Handled** – было ли событие полностью обработано.

Эту информацию можно использовать в обработчике, например:

```
private void Form1_KeyDown (object sender, KeyEventArgs e)
  {  MessageBox.Show(e.KeyCode.ToString(), "клавиша нажата!");
  }
```

Пример 1

Вывод графических примитивов на форму.

1. Создадим новый проект **wf351** типа Windows Forms.
2. Подключим пространство имен **System.Drawing**.
3. Разместим на форме поле выбора со списком **listBox1**. В пункте **Item** окна свойств введем список графических примитивов: **Line, Rectangle, FillRectangle, Ellipse, FillEllipse, Pie, FillPie** (рис. 3.23).

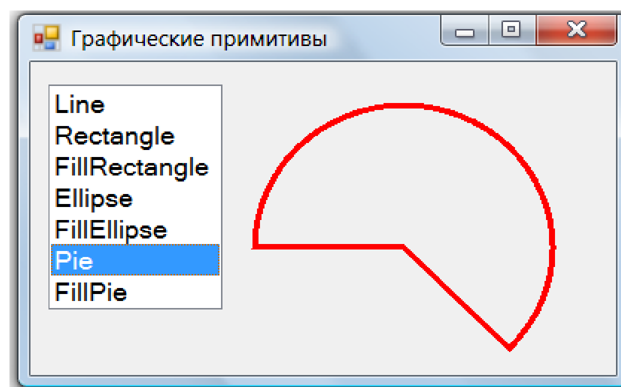


Рис. 3.23. Выбор и рисование графических примитивов на форме

4. Зарегистрируем событие выбора из списка **SelectedIndexChanged.** В шаблон обработчика введем код выбора и вывода примитивов на форму:

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
 {  Graphics g = this.CreateGraphics();    // создание графического объекта
    Pen pn = new Pen(Color.Red,4);         // создание пера
    Brush br = new SolidBrush(Color.Green);  // создание кисти
    g.Clear(SystemColors.Control);         // очистка области цветом формы
    switch (listBox1.SelectedIndex)        // выбор и рисование примитива
     {  case 0: g.DrawLine(pn, 150,40, 350,180); break;
        case 1: g.DrawRectangle(pn, 150,30, 250,150); break;
        case 2: g.FillRectangle(br, 150, 30, 250, 150); break;
        case 3: g.DrawEllipse(pn, 150,30, 250,150); break;
        case 4: g.FillEllipse(br, 150, 30, 250, 150); break;
        case 5: g.DrawPie(pn, 150,30, 200,200, 180,225); break;
        case 6: g.FillPie(br, 150,30, 150,150, 0,45); break;
     }
 }
```

5. Протестируем программу. Откорректируем свойства компонентов и код.

Пример 2

Простейший графический редактор. Рисование мышью.

1. Создадим новый проект **wf352** типа Windows Forms.
2. Разместим на форме кнопку для очистки (рис. 3.24, *а*).
3. Зарегистрируем три события мыши для формы (**MouseDown, MouseUp, MouseMove**) и событие нажатия кнопки **Click**. Введем коды в шаблоны обработчиков событий:

```
        // инициализация: перо поднято, цвет черный, толщина 4 px
bool ris = false; Color clr = Color.Black;  int w = 4;
        // обработчики событий мыши
private void Form1_MouseDown(object sender, MouseEventArgs e)
 {  ris = true;  }                              // перо опущено
private void Form1_MouseUp(object sender, MouseEventArgs e)
 {  ris = false;  }                             // перо поднято
        // движение мыши, вывод ее координат в заголовок формы
private void Form1_MouseMove(object sender, MouseEventArgs e)
 {  this.Text = "x = " + e.X + "  y = " + e.Y;
    if (ris)                                    // если нажата кнопка мыши
     {  Graphics g = CreateGraphics();   // рисуем закрашенным квадратом
        g.FillRectangle(new SolidBrush(clr), e.X, e.Y, w, w);
     }
 }
private void button1_Click(object sender, EventArgs e)
 {  Graphics g = CreateGraphics();
    g.Clear(SystemColors.Control);
 }
```

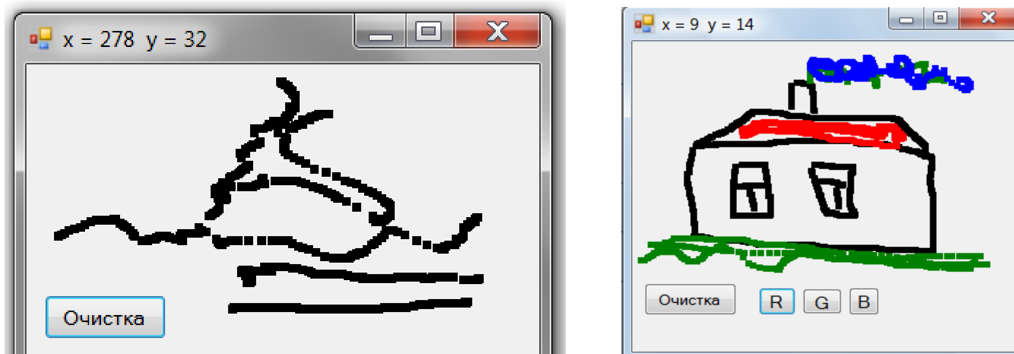


Рис. 3.24. Варианты интерфейса графического редактора

4. Протестируем программу. При необходимости откорректируем свойства элементов и код.

Пример 3

Вывод текста в графике.

1. Создадим новый проект **wf353** типа Windows Forms.
2. Разместим на форме надпись, поле ввода текста и три кнопки для выбора шрифта, цвета и вывода текста (рис. 3.25).

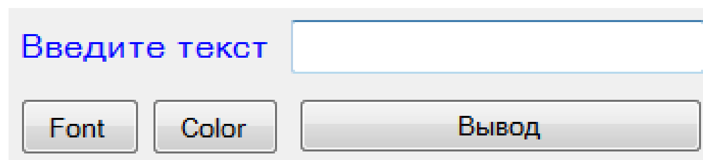


Рис. 3.25. Интерфейс приложения **wf353**

3. Зарегистрируем события нажатия кнопок. Введем код в шаблоны обработки:

```
        // инициализируем поля, задаем цвет текста и шрифт по умолчанию
Color clr = Color.Black;
Font fnt = new Font("Times New Roman", 14);
        // вызов диалога выбора шрифта
private void button1_Click(object sender, EventArgs e)
 {  FontDialog fntDia = new FontDialog();
    fntDia.ShowDialog();   fnt = fntDia.Font;
 }
        // вызов диалога выбора цвета текста
private void button2_Click(object sender, EventArgs e)
 {  ColorDialog colDia = new ColorDialog();
    colDia.ShowDialog();   clr = colDia.Color;
 }
        // ввод и рисование текста
private void button3_Click(object sender, EventArgs e)
 {  string s = textBox1.Text;                      // ввод текста
    Graphics g = CreateGraphics();
    g.Clear(SystemColors.Control);
    Brush br = new SolidBrush(clr);
    g.DrawString(s, fnt, br, 20, 100);             // вывод текста
 }
```

4. Протестируем программу. При необходимости откорректируем свойства элементов и код.

Задания для самостоятельной работы

1. Модифицируйте пример 1, добавив выбор цвета пера и кисти (Red, Green, Blue) с помощью **radioButton** и толщины пера с помощью **numericUpDown**.

2. Модифицируйте пример 2, добавив выбор цвета пера (Red, Green, Blue) с помощью кнопок и толщины пера с помощью клавиш **Up** и **Down**.

3. Создайте приложение, выводящее в указанное щелчком мыши место формы закрашенный кружок.

4. Модифицируйте пример 3, добавив вывод текста в указанное щелчком мыши место (событие MouseDown).

5. Создайте приложение, выводящее на форму **N** закрашенных квадратиков со случайными координатами

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Основная

1. Павловская, Т. А. C#. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2014. – 432 с.
2. Подбельский, В. В. Язык C#. Базовый курс / В. В. Подбельский. – М. : Финансы и статистика, 2015. – 408 с.
3. Фленов, М. Библия C# / М. Фленов. – СПб. : БХВ, 2016. – 544 с.

Дополнительная

4. Албахари, Д. C# 6.0. Справочник. Полное описание языка / Д. Албахари, Б. Албахари. – М. : «И. Д. Вильямс», 2016. – 1040 с.
5. Вайсфельд, М. Объектно-ориентированное мышление / М. Вайсфельд. – СПб. : Питер, 2014. – 304 с.
6. Скит Дж., C#: программирование для профессионалов / Дж. Скит. – М. : «И. Д. Вильямс», 2018. – 608 с.
7. Троелсен, Э. Язык программирования C# 6.0 и платформа. NET 4.6 / Э. Троелсен, Ф. Джепикс. – М. : «И. Д. Вильямс», 2017. – 1440 с.
8. Хейлсберг, А. Язык программирования C# / А. Хейлсберг, М. Торгерсен, С. Вилтамут, П. Голд. – СПб. : Питер, 2012. – 784 с.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ ... 3
1. Использование стандартных объектов и методов C# ... 4
1.1. Основы работы в среде Microsoft Visual Studio ... 4
1.2. Алгоритмическая конструкция “ветвление” ... 8
1.3. Алгоритмическая конструкция “цикл” ... 11
1.4. Работа с массивами ... 15
1.5. Работа с двумерными массивами ... 19
1.6. Обработка исключений ... 22
1.7. Работа с символами ... 25
1.8. Работа со строками ... 28
1.9. Использование регулярных выражений ... 31
1.10. Работа с файлами ... 34
2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ СОБСТВЕННЫХ КЛАССОВ ... 38
2.1. Создание класса и объекта. Методы. Конструкторы ... 38
2.2. Перегрузка методов ... 43
2.3. Инкапсуляция. Сокрытие полей, создание свойств ... 45
2.4. Визуальное проектирование классов ... 48
2.5. Наследование ... 52
2.6. Абстрактные классы. Интерфейсы ... 55
3. СОЗДАНИЕ WINDOWS-ПРИЛОЖЕНИЙ ... 59
3.1. Разработка приложений Windows Forms ... 62
3.2. Интерактивное управление параметрами приложений ... 67
3.3. Использование таймера. Анимация ... 71
3.4. Использование меню и диалоговых окон ... 74
3.5. Работа с графикой GDI+ ... 79
БИБЛИОГРАФИЧЕСКИЙ СПИСОК ... 83

Учебное издание

ЗАБОРОВСКИЙ Георгий Александрович
СИДОРИК Валерий Владимироваич

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C#

Учебно-методическое пособие
для студентов и слушателей системы повышения квалификации
и переподготовки, преподавателей

Редактор *А. С. Кириллова*
Компьютерная верстка *Н. А. Школьниковой*

Подписано в печать 16.11.2020. Формат 60×84 $^{1}/_{8}$. Бумага офсетная. Цифровая печать.
Усл. печ. л. 9,65. Уч.-изд. л. 3,77. Тираж 100. Заказ 572.

Издатель и полиграфическое исполнение: Белорусский национальный технический университет.
Свидетельство о государственной регистрации издателя, изготовителя, распространителя
печатных изданий № 1/173 от 12.02.2014. Пр. Независимости, 65. 220013, г. Минск.