Министерство сельского хозяйства РФ

Кубанский государственный
аграрный университет

Кафедра Системного анализа и обработки информации

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Методические указания к лабораторным работам по дисциплине
«Языки программирования»

для студентов второго курса направления подготовки 09.03.02
«ИФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ»
всех форм обучения

Краснодар, 2019

Работу подготовили по решению методической комиссии факультета прикладной информатики и кафедры Системного анализа и обработки информации (протокол №_____ от _____ 2019 г.)
Мурлин А.Г., Иванова Е.А.,

Языки программирования.

Методические указания к лабораторным работам по дисциплине «Языки программирования» для студентов второго курса направления подготовки 09.03.02 «ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ» всех форм обучения/ Кубан. гос. аграрн. ун-т., Сост. А.Г. Мурлин, Е.А. Иванова, 2019, 198 с.

Составлены в соответствии с рабочей программой курса “Языки программирования” для студентов второго курса направления подготовки 09.03.02

Содержат описание лабораторных работ, методические указания к их выполнению, требования к оформлению отчета, приложение с примерами программ и пр.

Ил. 46. Библиогр.: 8 назв.

Рецензенты: проф. В.И. Лойко (КубГАУ),
проф. Л.А. Максименко (КубГТУ).

СОДЕРЖАНИЕ

Лабораторная работа №1. Работа со строками

1 Цель работы

Изучить принципы разработки программ с использованием строк. Изучить стандартные свойства и методы работы со строками типа **string**.

2 Порядок выполнения работы

Получить задание на выполнение лабораторной работы (раздел 6) согласно своему варианту. Разработать и отладить программу. Составить и защитить отчет о лабораторной работе у преподавателя.

3 Содержание отчета

- наименование и цель работы;
- задание на лабораторную работу согласно варианту;
- схема алгоритма, текст программы на алгоритмическом языке;
- результаты работы программы.

4 Краткая теория

Основным типом при работе со строками является тип **string**, задающий строки переменной длины.

Переменные типа **string** объявляются как все прочие объекты простых типов – с явной или отложенной инициализацией. Чаще всего при объявлении строковой переменной инициализация задается строковой константой:

```
string s1 = "Hello, world!";
```

Но, кроме этого, строку можно сконструировать из:

- символа, повторенного заданное число раз:

```
string s2 = new string('a',5);    //строка s2 будет состоять
                                  // из 5 символов 'a'
```

- массива символов **char[]**:

```
char[] yes = "Yes".ToCharArray();
string s3 = new string(yes);      //s3 = "Yes"
```

– части массива символов. При этом задается индекс начального элемента массива и количество элементов массива, которые должны попасть в строку:

```
char[] yes = "Yes".ToCharArray();
string s4 = new string(yes,0,2); //s4 = "Ye" (из строки yes
                                 //берем 2 символа, начиная
                                 //с нулевого)
```

4.1 Операции над строками

Над строками определены следующие операции:

– присваивание (**=**);

– две операции проверки эквивалентности (**==**) и (**!=**). Другие операции сравнения (**<**, **>**, **<=**, **>=**) применительно к строкам дадут некорректные

результаты, т.к. будут сравнивать не сами строки, а ссылки на них, т.е. адреса в памяти, где эти строки расположены;

– конкатенация или сцепление строк (**+**). Бинарная операция "**+**" сцепляет две строки, приписывая вторую строку к хвосту первой;

– взятие индекса (**[]**). Возможность взятия индекса при работе со строками отражает тот факт, что строку можно рассматривать как массив и получать без труда каждый ее символ. Каждый символ строки имеет тип **char**, доступный только для чтения, но не для записи.

4.2 Свойства и методы типа `string`

В языке C# существует понятие неизменяемый (immutable) тип. Для такого типа невозможно изменить значение переменной. Динамические методы могут создавать новую переменную, но не могут изменить значение существующей.

К таким неизменяемым типам относится и тип **string**. Ни один из его методов не меняет значения существующих переменных. Конечно, некоторые из методов создают новые значения и возвращают в качестве результата новые строки. Невозможность изменять значения строк касается не только методов. Аналогично, при работе со строкой как с массивом разрешено только чтение отдельных символов, но не их замена:

```
string s1= "test";
char ch1 = s1[0];
//s1[0]='T';  //Ошибка! Нельзя менять содержимое строки!
```

Свойства и методы типа **string** (как, впрочем, и других типов) делятся на обычные и статические. Обычные свойства и методы вызываются через строковые переменные (объекты):

```
Имя_строки.Имя_свойства
```

или

```
Имя_строки.Имя_метода()
```

Вызов же статических компонентов осуществляется через название типа (**string**):

```
string.Имя_свойства
```

или

```
string.Имя_метода()
```

Начнем со статических компонентов для работы со строками. Список основных свойств и методов приведен в таблице 6.1.

Таблица 1.1 – Статические свойства и методы типа **string**

Компонент	Описание
Свойства	
Empty	Возвращает пустую строку. Свойство только для чтения
Методы	

Compare(строка1, строка2)	Сравнение двух строк. Возвращает отрицательное значение, если первая строка меньше второй, положительное – если больше, ноль – если строки равны.
Concat(строка1, строка2)	Конкатенация (сцепление) строк.
Copy(строка)	Создает копию строки и возвращает ее как результат
Format(строка, формат)	Выполняет форматирование в соответствии с заданными спецификациями формата. Более полное описание метода можно посмотреть в справочниках
Join(разделители, массив_строк)	Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители. Операция, заданная методом **Join**, является обратной к операции, заданной методом **Split.** Последний является обычным (не статическим) методом и, используя разделители, осуществляет разделение строки на элементы. Подробнее эти методы описаны ниже.

Методы **Join** и **Split** выполняют над строкой текста взаимно обратные преобразования. Метод **Split** позволяет осуществить разбор текста на элементы. Статический метод **Join** выполняет обратную операцию, собирая строку из элементов.

Заданный строкой текст зачастую представляет собой совокупность структурированных элементов - абзацев, предложений, слов, скобочных выражений и т.д. При работе с таким текстом необходимо разделить его на элементы, пользуясь специальными разделителями элементов, - это могут быть пробелы, скобки, знаки препинания. Практически подобные задачи возникают постоянно при работе со структурированными текстами. Методы **Split** и **Join** облегчают решение этих задач.

Метод **Split** имеет следующий синтаксис:

массив_строк Split(символы_разделители)

На вход методу **Split** передается один или несколько символов, интерпретируемых как разделители. Объект **string**, вызвавший метод, разделяется на подстроки, ограниченные этими разделителями. Из этих подстрок создается массив, возвращаемый в качестве результата метода. Другая реализация позволяет ограничить число элементов возвращаемого массива. С помощью дополнительного параметра метода **Split** со значением **StringSplitOptions.RemoveEmptyEntries** можно избавиться от пустых элементов массива-результата, которые получаются в случае, если между словами встречаются сразу несколько символов-разделителей.

Синтаксис статического метода **Join** таков:

строка Join(строка_разделителей, массив_строк)

В качестве результата метод возвращает строку, полученную конкатенацией элементов массива строк, между которыми вставляется строка разделителей. Как правило, строка разделителей состоит из одного символа, который и разделяет в результирующей строке элементы массива; но в отдельных случаях ограничителем может быть строка из нескольких символов.

Сводка методов, приведенная в таблице 6.2, дает достаточно полную картину широких возможностей, имеющихся при работе со строками в C#. Следует помнить, что тип **string** является неизменяемым. Поэтому **Replace, Insert** и другие методы представляют собой функции, возвращающие новую строку в качестве результата и не изменяющие строку, вызвавшую метод.

Таблица 1.2 – Динамические методы типа **string**

Метод	Описание
Insert(индекс, подстрока)	Вставляет подстроку в заданную позицию
Remove(индекс, количество_символов)	Удаляет подстроку заданной длины в заданной позиции
Replace(подстрока, новая_подстрока	Заменяет все вхождения подстроки на новую подстроку
Substring(индекс, длина)	Выделяет подстроку заданной длины с заданной позиции
IndexOf(подстрока) **IndexOfAny(массив_символов)** **LastIndexOf(подстрока)** **LastIndexOfAny(массив_символов)**	Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного массива
StartsWith(подстрока) **EndsWith(подстрока)**	Возвращается **true** или **false**, в зависимости от того, начинается или заканчивается строка заданной подстрокой
PadLeft(длина) **PadRight(длина)**	Выполняет набивку нужным числом пробелов в начале и в конце строки так, чтобы длина строка стала равна заданному значению
Trim() **TrimStart(массив_символов)** **TrimEnd(массив_символов)**	Обратные операции к методам **Pad.** Удаляются пробелы (или другие символы, заданные в массиве) в начале и в конце строки, или только с одного ее конца
ToCharArray()	Преобразование строки в массив символов

4.3 Свойства и методы типа Char

Часто при анализе текстовой информации возникает необходимость проверки вида символов: является ли символ буквенным, цифровым, знаком препинания и т.п. В языке C# для этих целей можно использовать статические методы типа **Char**, основные из которых приведены в таблице 6.3.

Таблица 1.3 – Статические методы типа **Char**

Метод	Описание
GetNumericValue(символ)	Возвращает численное значение символа, если он является цифрой, и -1.0 в противном случае
IsControl(символ)	Возвращает **true**, если символ является управляющим
IsDigit(символ)	Возвращает **true**, если символ является десятичной цифрой
IsLetter(символ)	Возвращает **true**, если символ является буквой
IsLetterOrDigit(символ)	Возвращает **true**, если символ является буквой или цифрой
IsLower(символ)	Возвращает **true**, если символ задан в нижнем регистре
IsNumber(символ)	Возвращает **true**, если символ является числом (десятичной или шестнадцатеричной цифрой)
IsPunctuation(символ)	Возвращает **true**, если символ является знаком препинания
IsSeparator(символ)	Возвращает **true**, если символ является разделителем
IsUpper(символ)	Возвращает **true**, если символ задан в верхнем регистре
IsWhiteSpace(символ)	Возвращает **true**, если символ является "белым пробелом". К белым пробелам, помимо пробела, относятся и другие символы, например, табуляция, символ конца строки и символ перевода каретки
Parse(строка)	Преобразует строку в символ. Естественно, строка должна состоять из одного символа, иначе возникнет ошибка
ToLower(символ)	Приводит символ к нижнему регистру
ToUpper(символ)	Приводит символ к верхнему регистру

5 Примеры программ

5.1 Дана строка, в которой слова разделены одним или несколькими пробелами. Выделить слова из этой строки и отсортировать их в алфавитном порядке.

Внешний вид главной формы программы показан на рисунке 6.1.

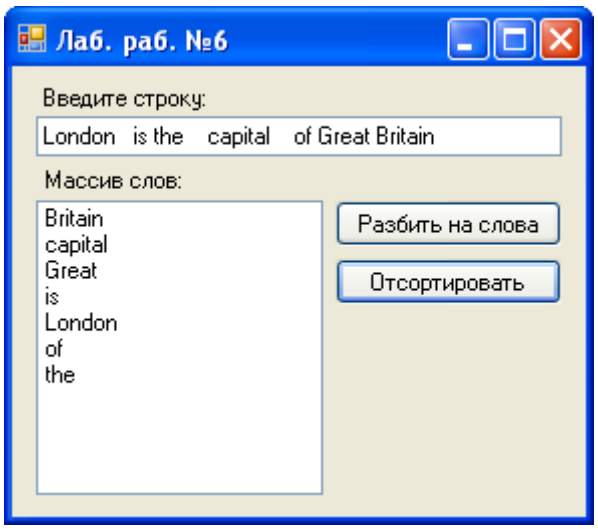


Рисунок 1.1 – Главное окно программы

```
//массив описываем вне обработчиков
string[] mas;

//Обработчик кнопки "Разбить на слова"
private void button1_Click(object sender, EventArgs e)
{
    //исходная строка
    string s = textBox1.Text;
    //массив разделителей состоит из одного элемента - пробела
    char[] sep = new char[] { ' ' };
    //разбиваем строку на слова и удаляем пустые строки
    mas = s.Split(sep, StringSplitOptions.RemoveEmptyEntries);
    //выводим слова в список на форме
    listBox1.Items.Clear();
    for (int i = 0; i < mas.Length; i++)
        listBox1.Items.Add(mas[i]);
}

//Обработчик кнопки "Отсортировать"
private void button2_Click(object sender, EventArgs e)
{
    string t = "";
    //сортировка слов методом "пузырька"
    for (int i = 0; i < mas.Length; i++)
        for (int j = mas.Length - 1; j > i; j--)
        if (mas[j].CompareTo(mas[j - 1]) < 0)
        {
            t = mas[j];
            mas[j] = mas[j - 1];
            mas[j - 1] = t;
        }
        //выводим отсортированные слова в список на форме
        listBox1.Items.Clear();
        for (int i = 0; i < mas.Length; i++)
            listBox1.Items.Add(mas[i]);
}
```

5.2 Дана текстовая строка. Сформировать новую строку из тех символов, которые стоят на нечетных позициях в исходной строке и не являются цифрами.

Внешний вид приложения приведен на рисунке 6.2.

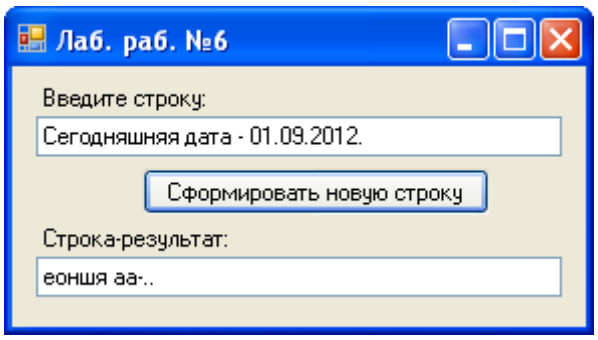


Рисунок 1.2 – Главное окно программы

```
private void button1_Click(object sender, EventArgs e)
{
    //исходная строка
    string s1 = textBox1.Text;
    string s2 = "";
    //просмотр символов исходной строки
    for (int i = 0; i < s1.Length; i++)
        //если выполняются заданные условия,
        if (i % 2 != 0 && !char.IsDigit(s1[i]))
            //то добавляем символ в новую строку
            s2 += s1[i];
    textBox2.Text = s2;
}
```

5.3 Найти сумму цифр введенного числа.

Внешний вид окна программы показан на рисунке 6.3.

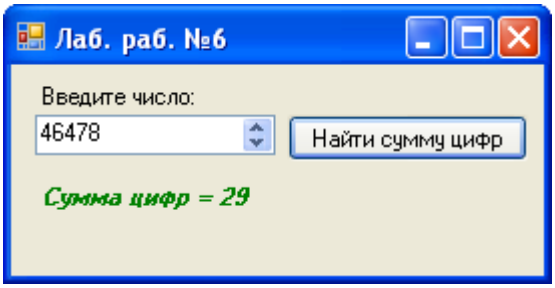


Рисунок 1.3 – Главное окно программы

```
private void button1_Click(object sender, EventArgs e)
{
    decimal n = numericUpDown1.Value;
    string s = n.ToString();
```

```
    int sum = 0;
    for (int i = 0; i < s.Length; i++)
        sum += Convert.ToInt32(s[i].ToString());
    label2.Text = "Сумма цифр = " + sum;
}
```

6 Варианты заданий для самостоятельного решения

1. Дана последовательность слов. Проверить правильность и исправить ошибки написания сочетаний "жи", "ши", "ча", "ща", "чу", "щу".
2. Дан массив фамилий. Составить новый массив, который будет содержать только женские фамилии. (Примечание: те фамилии, по которым пол трудно определить, не считать).
3. Найти произведение и минимальную четную цифру, составляющую некоторое число X.
4. Написать программу, которая запрашивает полное имя пользователя, затем пользователь вводит строку символов. Компьютер должен подсчитать, сколько раз в тексте встречается каждая буква из имени.
5. Перевести число из 10-ой системы в 16-тиричную и наоборот.
6. Дан текст, состоящий не менее чем из пяти слов. Сформировать строку, в которую попадут только те слова, где одинаковые буквы встречаются более двух раз. Например, м<u>о</u>л<u>о</u>к<u>о</u>.
7. Из имеющегося набора слов выбрать наиболее длинное и записать его прописными буквами.
8. Дана строка цифр. Сформировать строку, в которую войдут все цифры из исходной строки, кроме той, которая встречается наибольшее количество раз. Ее вывести отдельно.
9. Дана последовательность, состоящая из цифр, букв и знаков пунктуации в произвольном порядке. Подсчитать чего больше и составить строки только из цифр, букв и знаков пунктуации.
10. Дана некоторая последовательность букв русского алфавита. Написать программу, которая запрашивает Ваше имя и определяет, можно ли из букв исходной строки составить его.
11. Дан текст, записанный в виде криптограммы (шифрограммы), в которой буквы истинного текста размещаются в позициях, кратных 3. Прочитать исходный текст.
12. Дана строка цифр. Составить из них 5-значные числа. Если на последнее число не хватит цифр, дополнить его первыми цифрами исходной строки.
13. Перевести число из 2-ой системы в 10-тичную и наоборот.
14. Дан текст. Найти самое короткое слово длиной не менее 3 букв.
15. Написать программу, которая переставит слова в строке по мере увеличения их длины.
16. Дан текст, состоящий не менее чем из пяти слов. Вывести на экран слова, в которых отсутствует буква "Е".
17. Дан текст, состоящий не менее чем из пяти слов. Определить процентное

соотношение в нем коротких слов (длиной менее четырех символов) к длинным (все остальные).
18. Найти количество нечетных цифр и максимальную нечетную цифру, составляющую некоторое число X.
19. Дан текст, состоящий не менее чем из пяти слов. Определить, есть ли в нем слова, начинающиеся и заканчивающиеся с буквы "А", а также количество таких слов.
20. Дан текст, состоящий не менее чем из пяти слов. Написать программу, которая в каждом слове делает заглавной первую букву.
21. Написать программу, которая проверяет, является ли введенное слово палиндромом. Палиндромом называется слово, которое читается одинаково слева направо и справа налево, например, "КАЗАК".
22. Дана строка слов. Определить буквы, которые встречаются наибольшее и наименьшее количество раз.
23. Дан текст, состоящий не менее чем из пяти слов. Вывести на экран слова, которые имеют окончания "ИЯ","ИСТ","ИКА".
24. Найти минимальную и максимальную цифры среди четных и нечетных цифр, составляющих некоторое число X.
25. Дан текст. Преобразовать его по следующему правилу: если нет символа ‘*’, то оставить его без изменения, иначе заменить каждый символ, встречающийся после первого вхождения символа ‘*’, на символ ‘–’ .
26. Подсчитать сумму и количество всех цифр, входящих в некоторое предложение, вводимое с клавиатуры.
27. Дан текст, состоящий не менее чем из семи слов. Все слова из четырех букв записать наоборот.
28. Дана последовательность из латинских букв и цифр. Определить, чего больше.
29. Дан текст, оканчивающийся точкой. Найти количество слов, у которых первый и последний символы совпадают.
30. Найти среднее арифметическое значение между минимальной и максимальной цифрами некоторого числа X.
31. Дан текст, состоящий не менее чем из семи слов. Определить, есть ли в нем слова, начинающиеся с буквы "Ф", а также количество таких слов.
32. Дан текст. Заменить в нем символы, расположенные между круглыми скобками на символ ‘*’ (внутри каждой пары скобок нет других скобок).

Лабораторная работа №2. Составные типы данных: структуры

1 Цель работы

Получить практические навыки использования комбинированного типа данных СТРУКТУРА в разработке приложений.

2 Порядок выполнения работы

Получить задание на выполнение лабораторной работы согласно своему варианту. Разработать и отладить программу. Составить и защитить отчет о лабораторной работе у преподавателя.

3 Содержание отчета

- наименование и цель работы;
- задание на лабораторную работу согласно варианту;
- схема алгоритма, текст программы на алгоритмическом языке;
- результаты работы программы.

4 Краткая теория

Структуры – это составные типы данных, построенные с использованием других типов. Они представляют собой объединенный общим именем набор данных различных типов. Именно тем, что в них могут храниться данные разных типов, они и отличаются от массивов, хранящих данные одного типа.

Отдельные данные структуры называются *элементами или полями*. Все это напоминает запись в базе данных, только хранящуюся в оперативной памяти компьютера.

Простейший вариант объявления структуры может выглядеть следующим образом:

```
struct address
{
  public string name;
  public int number;
  public string street;
  public string town;
  public long zip;
}
```

Ключевое слово **struct** начинает определение структуры. Идентификатор **address** – обозначение, имя типа структуры. Оно используется при объявлении переменных структур данного типа. Имена, объявленные в фигурных скобках описания структуры – это элементы структуры. Элементы одной и той же структуры должны иметь уникальные имена, но две разные структуры могут содержать не конфликтующие элементы с одинаковыми именами.

Определение **address** содержит пять элементов. Предполагается, что такая структура может хранить данные о почтовом адресе. Типы данных разные: элементы **name, street** и **town** – строки, хранящие соответственно фамилию адресата, улицу и город, где он проживает. Элемент **number** целого типа хранит номер дома, элемент **zip** типа **long** хранит сведения об индексе. Служебное слово **public** перед описанием каждого поля необходимо для того, чтобы получить доступ к полям структуры из любого места программы.

Само по себе объявление структуры не резервирует никакого пространства в памяти; оно только создает новый тип данных, который может использоваться для объявления переменных. Переменные структуры объявляются так же, как переменные других типов. Строки

```
address addr;
address[] addrArray = new address[10];
```

объявляют переменную **addr** типа **address** и массив **addrArray** – с 10 элементами типа **address.**

Для доступа к элементам структуры используется операция точка (.), которая обращается к элементу структуры по имени объекта или по ссылке на объект. Например:

```
addr.name = "Иванов Иван Иванович";
addr.number = 79;
addr.street = "Красная";
addrArray[0].town = "Краснодар";
MessageBox.Show(addr.name);
```

Элементом определяемой структуры может быть структура, тип которой уже определен:

```
struct point
 {
     public int x;
     public int y;
 }
 struct rect
 {
     public point LeftTop;
     public point RightBottom;
     public string color;
}
```

Рассмотрим «взаимоотношение» структур и функций. Функция может возвращать структуру как результат:

```
struct message
{
  public string name;
  public int number;
};
message func1()
{
    ...
}
```

Через аппарат параметров в функцию может передаваться информация о структуре:

```
void func2(message str)
{
    ...
}
```

5 Пример программы

Программа находит в массиве данных о людях самого младшего (по годам) человека.

Внешний вид окна приложения приведен на рисунке 7.1.

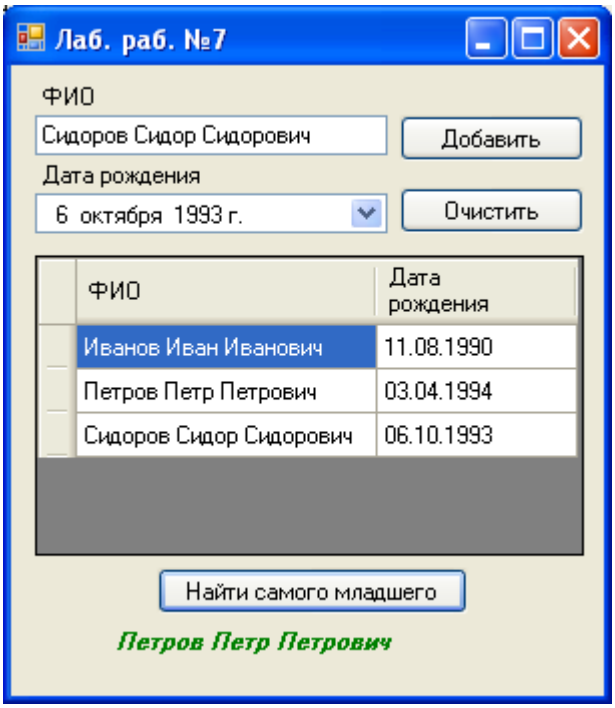


Рисунок 2.1 – Главное окно приложения

Для указания даты рождения на форме используется компонент **`DateTimePicker`**, а в программном коде – тип **`DateTime`**, представляющий значения типа «Дата/время». Для того чтобы получить из переменной типа **`DateTime`** значение даты в формате «чч.мм.гггг» (для последующего занесения его в таблицу), в программе использован метод **`ToShortDateString().`**

Обратите внимание, что описания структур и массива находятся вне обработчиков событий (например, перед ними).

```
//структура "Дата"
struct Date
{
    public int number;
    public int month;
    public int year;
}
```

```
//структура "Человек"
struct Person
{
    public string FIO;
    public Date Birthday;
}

//описание массива людей
Person[] mas;

//обработчик кнопки "Добавить"
private void button1_Click(object sender, EventArgs e)
{
    //берем данные о человеке из компонентов на форме
    string fio = textBox1.Text;
    DateTime birthday = dateTimePicker1.Value;
    //и заносим их в таблицу dataGridView1
    dataGridView1.Rows.Add(fio, birthday.ToShortDateString());
}

//обработчик кнопки "Очистить"
private void button2_Click(object sender, EventArgs e)
{
    dataGridView1.Rows.Clear();
}

//обработчик кнопки "Найти самого младшего"
private void button3_Click(object sender, EventArgs e)
{
    //определяем кол-во людей в массиве
    int n = dataGridView1.RowCount;
    mas = new Person[n];
    //записываем данные из таблицы на форме в массив mas
    for (int i = 0; i < n; i++)
    {
        mas[i].FIO =
dataGridView1.Rows[i].Cells[0].Value.ToString();
        DateTime dt = Convert.ToDateTime(dataGridView1.Rows[i].
                      Cells[1].Value.ToString());
        mas[i].Birthday.number = dt.Day;
        mas[i].Birthday.month = dt.Month;
        mas[i].Birthday.year = dt.Year;
    }
    //элемент массива, соответствующий самому младшему человеку
    Person min = mas[0];
    //его порядковый номер в массиве
    int ind = 0;
    //ищем самого младшего
    for (int i = 1; i < n; i++)
        if (mas[i].Birthday.year > min.Birthday.year)
        {
            min = mas[i];
```

```
            ind = i;
        }
        //выводим его ФИО на форму
        label3.Text = mas[ind].FIO;
}
```

6 Варианты заданий для самостоятельного решения

1. Для каждого из N студентов группы известны ФИО и оценки (в баллах) по четырем дисциплинам. Найти среднюю оценку каждого студента и выбрать человека, имеющего максимальный средний бал.
2. Дан массив студентов ВУЗа: ФИО, возраст, регион, факультет. Вывести на экран результирующую таблицу: регион, количество студентов из этого региона. Отсортировать данные по названию региона.
3. Багаж пассажира характеризуется количеством вещей и общим весом вещей. Дан массив, содержащий сведения о багаже нескольких пассажиров. Найти средний вес одной вещи в каждом багаже и по всем пассажирам.
4. Дан массив данных о клиентах пункта проката автомобилей: ФИО, адрес (улица, дом, квартира) и марка машины. Во второй массив записать данные только тех людей, кто ездит на «Audi».
5. Дан список английских глаголов: тип (правильный/неправильный), первая форма, вторая форма, третья форма. Вывести в алфавитном порядке неправильные глаголы, у которых все три формы совпадают.
6. Рациональное число можно представить записью с двумя полями: числитель и знаменатель. Дан массив из N рациональных чисел. Разработать функцию для нахождения максимального среди них.
7. Дан массив, в котором хранятся данные о расписании самолетов на неделю: пункт назначения, время вылета, количество свободных мест. В кассу аэропорта обращается пассажир, желающий купить ***n*** билетов на первую половину сегодняшнего дня до выбранного пункта назначения. Определить, есть ли возможность выполнить его заказ, предоставить ему возможные варианты на выбор.
8. Дан массив данных о работниках фирмы: ФИО и год поступления на работу. Во второй массив записать только данные тех из них, кто на сегодняшний день проработал уже не менее 5 лет.
9. Дан массив данных о работниках фирмы: фамилия, имя, отчество, адрес (улица, дом, квартира) и год поступления на работу. Определить, есть ли в списке Петровы (Петров, Петрова), если есть, то вывести их адрес (адреса).
10. Дан список иногородних студентов из ***n*** человек: ФИО, адрес (город, улица, дом-квартира), расстояние до Краснодара. Для них в общежитии выделено ***k*** мест. Вывести список студентов, которых необходимо селить в общежитие в первую очередь. Критерий отбора: расстояние до города. Подсказка: отсортировать исходный массив по убыванию расстояний.
11. Дан массив данных о студентах некоторой группы: фамилия, имя, отчество и дата рождения (день, месяц, год). Вывести на экран фамилию и имя тех студентов, у кого сегодня день рождения (сегодняшнюю дату вводить с

клавиатуры).
12. Дан массив, содержащий сведения о студентах некоторой группы: ФИО, оценки по пяти экзаменационным дисциплинам. Вывести студентов, получающих повышенную стипендию. Организовать поиск студента по фамилии с выводом информации о нем.
13. Дан массив книг: название, автор, количество страниц. Вывести все книги А.С. Пушкина в алфавитном порядке.
14. Дан массив, в котором хранятся данные о расписании поездов на сегодняшний день: номер поезда, название (т.е. откуда – куда, например, Новороссийск-Москва), время прибытия на станцию и время отправления (часы, минуты). По данному времени определить, какие из поездов стоят сейчас на станции.
15. Дан массив, содержащий сведения о студентах некоторой группы: ФИО, адрес (улица, дом, квартира) и телефон (если есть). Вывести на экран сведения о тех из них, до которых нельзя дозвониться.
16. Точка плоскости может быть представлена двумя координатами X и Y. Дан массив, содержащий N точек. Найти точку, которые максимально удалена от начала координат.
17. Багаж пассажира характеризуется количеством вещей и общим весом вещей. Дан массив, содержащий сведения о багаже нескольких пассажиров. Выяснить имеется ли пассажир, багаж которого состоит из одной вещи весом более 30 кг.
18. Комплексное число можно представить через реальную (*Re*) и мнимую (*Im*) часть. Разработать функции сложения и вычитания комплексных чисел. Пусть $Z_1=(Re_1, Im_1)$, $Z_2=(Re_2, Im_2)$, тогда: 1) сумма: $Z = (Re_1+Re_2, Im_1+Im_2)$; 2) разность: $Z = (Re_1-Re_2, Im_1-Im_2)$.
19. Дан перечень постельного белья: название (наволочка/одеяло/простыня), цвет, размер (односпальный/полуторный/двуспальный). Подсчитать, сколько комплектов двуспального белья белого цвета можно составить из данного перечня.
20. Дан массив, содержащий сведения о студентах группы: фамилия, имя, отчество, дата рождения (день, месяц, год). Найти самого младшего студента по полной дате рождения.
21. Время можно представить с помощью часов, минут и секунд. Написать функцию *проверка(t1,t2),* проверяющую, предшествует ли время *t1* времени *t2* (в рамках суток).
22. Для каждого из N студентов группы известны ФИО и оценки (в баллах) по пяти дисциплинам. Вывести на экран фамилию и имя тех из них, у которых количество положительных оценок больше, чем отрицательных (положительная оценка – 3 и больше).
23. Рациональное число можно представить записью с тремя полями: целая часть, числитель и знаменатель. Разработать функцию, позволяющую из неправильной дроби получить правильную. Неправильной называется дробь, у которой числитель больше знаменателя.
24. Рациональное число можно представить записью с тремя полями: целая

часть, числитель и знаменатель. Дан массив рациональных чисел. Найти их сумму и по возможности сократить.

25. Багаж пассажира характеризуется количеством вещей и общим весом вещей. Дан массив, содержащий сведения о багаже нескольких пассажиров. Найти число пассажиров, имеющих более двух вещей.

26. Имеется список членов коллектива с указанием принадлежности каждого к различным общественным организациям (профком, ученый совет, общество книголюбов и т.п.). Напечатать приглашение всем членам на очередное заседание указанной организации. Задается только вид организации, место и время сбора.

27. Дан массив, содержащий информацию об учениках некоторой школы. Вывести на экран сведения об учениках только десятых классов. На сколько человек в девятых классах больше, чем в десятых.

28. Дан массив, содержащий сведения о книгах: название, жанр, автор. Вывести книги только классического жанра, отсортировав их по фамилии автора.

29. Время можно представить с помощью часов, минут и секунд. Написать функцию *перевод(t1,t2),* присваивающую параметру *t2* время на 1 секунду большее времени *t1* (учесть смену суток).

30. Дан массив книг: название, тип, автор, количество страниц, страна-родина автора. Организовать поиск по автору, по типу. Вывести все книги зарубежных авторов.

31. Рациональное число можно представить записью с двумя полями: числитель и знаменатель. Разработать функцию *сократить(m)* приведения рационального числа *m* к несократимому виду.

32. Дан массив данных о работниках фирмы: ФИО и адрес (улица, дом, квартира). Во второй массив записать только тех из них, которые живут на улице Красной. Вывести их на экран в алфавитном порядке.

Лабораторная работа №3. Работа с файлами

1 Цель работы

Получить практические навыки разработки программ с использованием файлового ввода/вывода. Изучить приемы работы с текстовыми и двоичными файлами.

2 Порядок выполнения работы

Получить задание для выполнения лабораторной работы согласно своему варианту. Разработать и отладить программу. Составить отчет о лабораторной работе и защитить его у преподавателя.

3 Содержание отчета

- наименование и цель работы;
- задание на лабораторную работу согласно варианту;
- схема алгоритма, текст программы на алгоритмическом языке;
- результаты работы программы.

4 Краткая теория

При решении многих задач возникает проблема хранения и обработки достаточно большого объема данных. В таких случаях широко используют внешние устройства для записи и чтения информации. Связь с внешними источниками, приемниками и носителями информации в C# осуществляется с помощью файлов.

Традиционно под файлом понимается поименованная совокупность данных на внешнем носителе, однако в C# этот термин трактуется более широко. ***Файлом*** здесь считается также **любое внешнее устройство**, по своему назначению являющееся источником или приемником информации, например, клавиатура, принтер, диск и т.д. Такое устройство принято называть логическим, поскольку учитывается только его главная функция, а не физические характеристики.

До начала операции ввода-вывода конкретному внешнему файлу должна быть поставлена в соответствие специальная переменная в программе.

4.1 Файловый ввод/вывод с помощью потоков

В языке C# файл рассматривается как поток (**`stream`**), представляющий собой последовательность считываемых или записываемых байтов. При этом поток «не знает», что и в какой последовательности в него записано. Расшифровка смысла записанных последовательностей байтов лежит на программе.

Для работы с файлами необходимо подключить пространство имен **`System.IO`**.

Чтобы создать байтовый поток, связанный с файлом, нужно описать переменную типа **FileStream**. При этом необходимо указать имя открываемого файла (оно может включать и полный путь к файлу) и режим его открытия. Этот режим может принимать одно из значений, заданных перечислением **FileMode**. Значения этого перечисления приведены в таблице 8.1.

Таблица 3.1 – Значения перечисления **FileMode**

Значение	Описание
FileMode.Append	Добавляет выходные данные в конец файла
FileMode.Create	Создает новый выходной файл. Существующий файл с таким же именем будет удален
FileMode.CreateNew	Создает новый выходной файл. Файл с таким же именем не должен существовать
FileMode.Open	Открывает существующий файл
FileMode.OpenOrCreate	Открывает файл, если он существует. В противном случае создает новый
FileMode.Truncate	Открывает существует файл, но усекает его длину до нуля

Например, если необходимо открыть существующий файл «test.dat» из текущей папки, то это можно сделать следующей строчкой:

```
FileStream in = new FileStream("test.dat", FileMode.Open);
```

По умолчанию файл открывается с доступом для чтения и записи. Если необходимо ограничить доступ только чтением или только записью, нужно использовать дополнительный параметр типа **FileAccess**. Он определяет способ доступа к файлу и может принимать одно из значений, определенных перечислением **FileAccess**: **FileAccess.Read**, **FileAccess.Write**, **FileAccess.ReadWrite**.

Например, при выполнении следующей инструкции файл test.dat будет открыт только для чтения:

```
FileStream fin = new FileStream("test.dat", FileMode.Open,
                 FileAccess.Read);
```

По завершении работы с файлом его необходимо закрыть. При закрытии файла освобождаются системные ресурсы, ранее выделенные для этого файла, что дает возможность использовать их для других файлов. Для этого достаточно вызвать метод **Close()**:

```
fin.Close();
```

В типе **FileStream** определены два метода, которые считывают байты из файла: **ReadByte()** и **Read()**. При каждом вызове метода **ReadByte()** из файла считывается один байт, и метод возвращает его как целочисленное значение. При обнаружении конца файла метод возвращает –1. Следующий пример считывает байты из файла до тех пор, пока не встретится конец файла (**EOF**):

```
int x;
do
{
     x = fin.ReadByte();
     if (x != -1)
          Console.WriteLine((char)x);
}
while (x != -1);
```

Чтобы считать блок байтов, используется метод **Read()**, общая форма вызова которого такова:

```
int Read(byte[] buf, int offset, int numBytes)
```

Метод **Read()** пытается считать **numBytes** байтов в массив **buf**, начиная с элемента **buf[offset]**. Он возвращает количество успешно считанных байтов.

Чтобы записать байт в файл, используется метод **WriteByte()**. Байт, который нужно записать, передается методу в качестве параметра. В следующем примере в текущей папке создается файл test.txt и в него записывается алфавит английского языка:

```
FileStream fout = new FileStream("test.txt", FileMode.Create);
for (char c = 'A'; c <= 'Z'; c++)
    fout.WriteByte((byte)c);
fout.Close();
```

С помощью метода **Write()** можно записать в файл массив байтов. Это делается следующим образом:

```
void Write(byte[] buf, int offset, int numBytes)
```

Метод записывает в файл **numBytes** байтов из массива **buf**, начиная с элемента **buf[offset].**

При выполнении операции вывода в файл выводимые данные зачастую не записываются немедленно на реальное физическое устройство, а буферизуются операционной системой до тех пор, пока не накопится порция данных достаточного размера, чтобы ее можно было всю сразу переписать на диск. Такой способ выполнения записи на диск повышает эффективность системы. Но, если нужно записать данные на физическое устройство вне зависимости от того, полон буфер или нет, следует использовать метод **Flush()**. При завершении работы с выходным файлом вызывается метод **Close()**. Это гарантирует, что любые данные, оставшиеся в дисковом буфере, будут переписаны на диск. Поэтому перед закрытием файла нет необходимости специально вызывать метод **Flush().**

4.2 Ввод/вывод текстовых файлов

Для работы с текстовыми файлами необходимо поместить переменную типа **FileStream** внутрь переменной типа **StreamReader** или **StreamWriter.** Эти типы автоматически преобразуют байтовый поток в символьный и наоборот.

Тип **StreamWriter** используется для записи данных в текстовый файл:

```
FileStream fout = new FileStream("test.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fout);
```

Метод **Write()** данного типа позволяет записать в файл переменную практически любого типа. При этом переменная будет автоматически преобразована в текстовый формат:

```
bool b = true;
double f = 3.15;
sw.Write(b);
sw.Write(f);
```

Метод **WriteLine()** записывает в файл текстовые данные и переходит на следующую строку (аналогично **Console.WriteLine()**):

```
sw.WriteLine("This is a test!");
```

Чтобы создать входной поток с ориентацией на обработку символов, байтовый поток необходимо поместить в объект **StreamReader**:

```
FileStream fin = new FileStream("test.txt", FileMode.Open);
StreamReader sr = new StreamReader(fin);
```

Метод **Read()** данного объекта по сути своей похож на аналогичный метод объекта **FileStream**, описанный выше. Однако при работе со строками гораздо удобнее использовать метод **ReadLine()**, читающий из файла целую строку. Если в результате вызова возвращается значение **null**, значит, достигнут конец файла. Далее показано, как организовать построчное чтение данных из текстового файла:

```
string s;
do
{
    s = sr.ReadLine();
    Console.WriteLine(s);
}
while (s != null);
```

Если нужно считать все символы от текущей позиции до конца файла, можно воспользоваться методом **ReadToEnd()**.

Иногда удобнее открывать файл напрямую с помощью типов **StreamWriter** или **StreamReader**, без использования **FileStream**. Например:

```
StreamWriter sw = new StreamWriter("test.txt", true);
StreamReader sr = new StreamReader("test.txt");
```

Для типа **StreamWriter** второй параметр, равный **true**, указывает, что выводимые данные должны добавляться в конец файла. В противном случае файл перезаписывается.

При завершении работы с файлами объекты **StreamWriter** и **StreamReader** также необходимо закрыть. Это делается с помощью метода **Close()**:

```
sw.Close();
sr.Close();
```

4.3 Двоичные файлы

Когда данные сохраняются в файле, их можно сохранить в текстовой форме или двоичном формате. Текстовая форма означает, что все данные сохраняются как текст, даже числа. Например, сохранение значения -2.324216e+07 в текстовой форме означает сохранение 13 символов, из которых состоит данное число. Двоичный формат означает, что число сохраняется во внутреннем представлении, т.е. вместо символов сохраняется 64-разрядное представление числа типа **`double`**. Для символа двоичное представление совпадает с его текстовым – двоичным представлением ASCII-кода (или его эквивалента) символа. Однако для чисел двоичное представление очень сильно отличается от их текстового представления (рис.8.1).

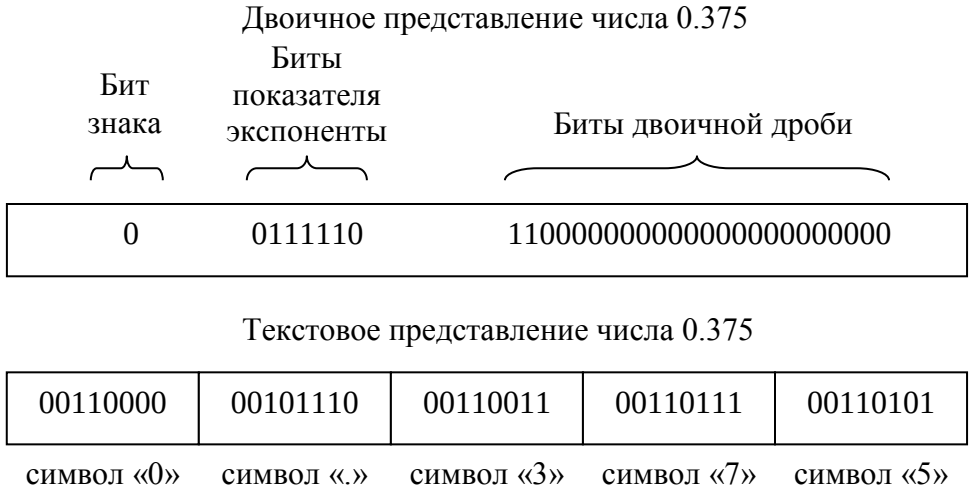


Рисунок 3.1 – Двоичное и текстовое представление вещественного числа

В двоичном формате числа сохраняются более точно, поскольку он позволяет сохранить точное внутреннее представление числа. Не происходит ошибок преобразования и округления. Сохранение данных в двоичном формате может происходить быстрее, поскольку при этом не происходит преобразования и данные можно сохранять большими блоками. Кроме того, двоичный формат обычно занимает меньше места.

Для записи в файл двоичных данных используется тип **`BinaryWriter`**, который привязывается к файловому потоку:

```
FileStream fout = new FileStream("test.dat", FileMode. Cre-
                  ate);
BinaryWriter bw = new BinaryWriter(fout);
```

В этом типе определены методы, способные записывать значения всех встроенных C#-типов. Основные из них перечислены в таблице 8.2. Также для типа **`BinaryWriter`** определены стандартные методы **`Close()`** и **`Flush()`**, работа с которыми описана выше.

Таблица 3.2 – Методы записи информации в двоичные файлы

Метод	Описание
`void Write(sbyte val)`	Записывает **byte**-значение (со знаком)
`void Write(byte val)`	Записывает **byte**-значение (без знака)
`void Write(byte[] buf)`	Записывает массив **byte**-значений
`void Write(short val)`	Записывает целочисленное значение типа **short** (короткое целое)
`void Write(ushort val)`	Записывает целочисленное **ushort**-значение (короткое целое без знака)
`void Write(int val)`	Записывает целочисленное значение типа **int**
`void Write(uint val)`	Записывает целочисленное **uint**-значение (целое без знака)
`void Write(long val)`	Записывает целочисленное значение типа **long** (длинное целое)
`void Write(ulong val)`	Записывает целочисленное **ulong**-значение (длинное целое без знака)
`void Write(float val)`	Записывает **float**-значение
`void Write(double val)`	Записывает **double**-значение
`void Write(char val)`	Записывает символ
`void Write(char[] val)`	Записывает массив символов
`void Write(string val)`	Записывает **string**-значение с использованием его внутреннего представления, которое включает спецификатор длины

Для чтения из двоичного файла используется тип **BinaryReader**:

```
FileStream fin = new FileStream("test.dat", FileMode.Read);
BinaryReader br = new BinaryReader(fin);
```

В этом типе предусмотрены методы для считывания всех простых C#-типов. Наиболее часто используемые из них показаны в таблице 8.3.

Таблица 3.3 – Методы чтения информации из двоичных файлов

Метод	Описание
`int Read()`	Возвращает целочисленное представление следующего доступного символа из вызывающего входного потока. При обнаружении конца файла возвращает значение -1
`int Read(byte[] buf, int offset, int num)`	Делает попытку прочитать **num** байтов в массив **buf**, начиная с элемента **buf[offset]**, и возвращает количество успешно прочитанных байтов
`bool ReadBoolean()`	Считывает **bool**-значение
`byte ReadByte()`	Считывает **byte**-значение
`sbyte ReadSByte()`	Считывает **sbyte**-значение
`byte[] ReadBytes(int num)`	Считывает **num** байтов и возвращает их в виде массива

char ReadChar()	Считывает **char**-значение
char[] ReadChars(int num)	Считывает **num** символов и возвращает их в виде массива
double ReadDouble()	Считывает **double**-значение
float ReadSingle()	Считывает **float**-значение
short ReadInt16()	Считывает **short**-значение
int ReadInt32()	Считывает **int**-значение
long ReadInt64()	Считывает **long**-значение
ushort ReadUInt16()	Считывает **ushort**-значение
uint ReadUInt32()	Считывает **uint**-значение
ulong ReadUInt64()	Считывает **ulong**-значение
string ReadString()	Считывает **string**-значение, представленное во внутреннем двоичном формате, который включает спецификатор длины

Также для типа **BinaryReader** определен стандартный метод **Close().**

4.5 Произвольный доступ к файлам

Произвольный доступ к файлам предоставляет возможность переместиться в любое место файла сразу, вместо последовательного передвижения по нему. Подход с произвольным доступом часто используется при обработке файлов баз данных. Этот подход проще реализовать, если файл состоит из набора записей одинакового типа (или хотя бы размера).

Для реализации «передвижения» по файлу используется метод **Seek(),** определенный для типа **FileStream**. Этот метод позволяет установить индикатор позиции (или указатель позиции) в любое место файла:

```
long Seek(long newPos, SeekOrigin origin)
```

Здесь элемент **newPos** означает новую позицию, выраженную в байтах, файлового указателя относительно позиции, заданной элементов **origin.** Элемент **origin** может принимать одно из значений, определенных перечислением **SeekOrigin**: **SeekOrigin.Begin** (поиск от начала файла), **SeekOrigin.Current** (от текущей позиции), **SeekOrigin.End** (от конца файла).

Примеры вызова функции:

```
//30 байтов от начала файла
fin.Seek(30, SeekOrigin.Begin);
//один байт назад от текущей позиции
fin.Seek(-1, SeekOrigin.Current);
//переход к концу файла
fout.Seek(0, SeekOrigin.End);
```

Получить текущую позицию файлового потока можно с помощью свойства **Position.**

5 Примеры программ

5.1 Пример работы с текстовым файлом. Дан текстовый файл, нужно вычислить его длину в байтах и сохранить ее в конце файла. Кроме того, каждый раз при встрече символа новой строки требуется сохранить текущее смещение в конце файла. Например, если исходный файл имеет вид

```
abcd
efg
hi
j
```

то программа должна модифицировать этот файл следующим образом:

```
abcd
efg
hi
j
6 11 15 18 18
```

Пояснение. В конце каждой строки содержатся по два символа: '\r' (возврат каретки) и '\n' (переход строки). Мы считаем только «Переход строки».

Внешний вид программного приложения показан на рисунке 8.2.

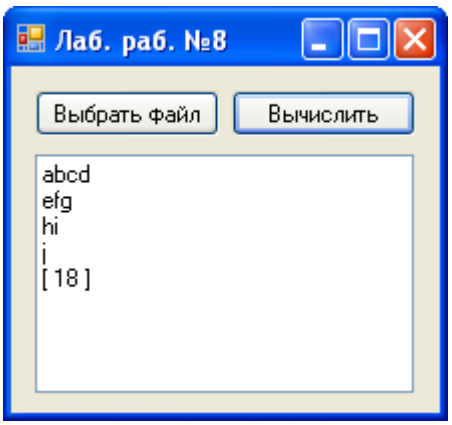


Рисунок 3.2 – Главное окно программы

Вывод результатов на форме осуществляется в компонент **`TextBox`**. Чтобы можно было выводить в него несколько строк, нужно установить его свойство **`Multiline`** в значение **`True`**.

Кроме этого, чтобы пользователь мог выбрать нужный ему файл для работы, в программе используется компонент **`OpenFileDialog`** из раздела **`Dialogs`** панели **`Toolbox`**, представляющий собой стандартное диалоговое окно выбора файла для открытия (рис. 8.3). Этот компонент невизуальный, т.е. при проектировании он не будет виден на форме.

Чтобы отобразить такое диалоговое окно на экране во время работы программы, нужно вызвать метод **`ShowDialog()`**. Если пользователь выберет файл и нажмет кнопку «Открыть», этот метод возвратит значение **`DialogResult.OK`**, и тогда можно будет выполнять какие-то действия с файлом. Если же пользователь нажимает «Отмена», то возвращается значение **`Di-`**

alogResult.Cancel. В этом случае ничего делать не нужно. Получить имя выбранного пользователем файла можно с помощью свойства **FileName** компонента **OpenFileDialog**.

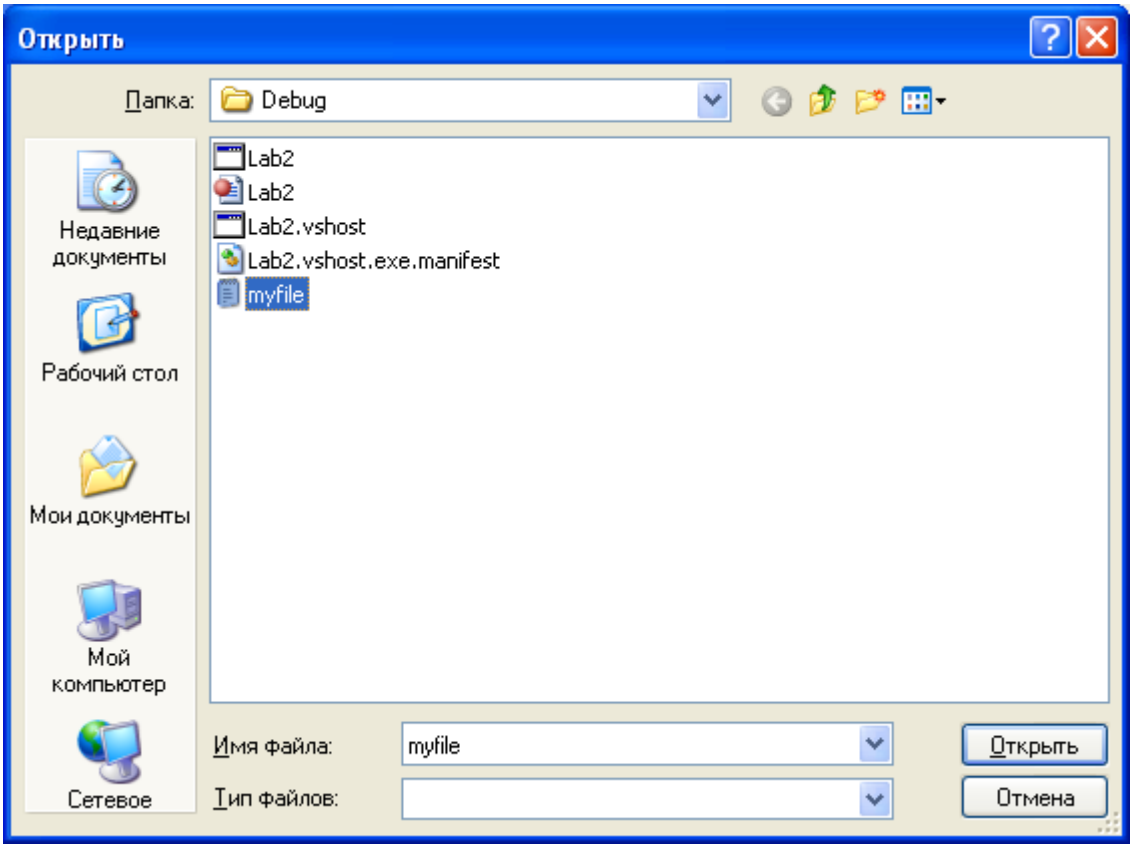


Рисунок 3.3 – Стандартное диалоговое окно выбора файла

В коде программы прежде всего следует подключить пространство имен **System.IO**:

```
using System.IO;
```

Переменная, содержащаая имя выбранного пользователем файла для работы, должна быть описана вне обработчиков событий (например, перед ними):

```
string fName;
```

Коды обработчиков событий:

```
//Обработчик кнопки "Выбрать файл"
private void button1_Click(object sender, EventArgs e)
{
    //показать стандартное диалоговое окно выбора файла
    //и, если пользователь нажал в нем кнопку "Открыть",
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
        //то получить имя выбранного файла
        fName = openFileDialog1.FileName;
}
```

```
//Обработчик кнопки "Вычислить"
private void button2_Click(object sender, EventArgs e)
{
    textBox1.Text = "";
    //открыть файл для ввода и дозаписи
    FileStream fs = new FileStream(fName, FileMode.Open);
    //создать объект для записи в текстовый файл
    StreamWriter sw = new StreamWriter(fs);
    int k = 0; // счетчик байтов
    char ch;
    //ставим указатель на начало файла
    fs.Seek(0, SeekOrigin.Begin);
    //считываем данные из файла посимвольно
    int x;
    do
    {
        //читаем очередной байт
        x = fs.ReadByte();
        //если не конец файла
        if (x != -1)
        {
            //переводим считанный байт в символ
            ch = (char)x;
            textBox1.Text += ch;  //скопировать на форму
            k++;
            if (ch == '\n')   //если конец строки
            {
                // запомнить текущую позицию
                long pos = fs.Position;
                //переместить указатель вывода в конец файла
                fs.Seek(0, SeekOrigin.End);
                sw.Write(k + " ");
                // восстановить позицию
                fs.Seek(pos, SeekOrigin.Begin);
            }
        }
    }
    while (x != -1);

    //вывести окончательное значение счетчика байтов
    sw.Write(k + " ");
    textBox1.Text += ("[ " + k + " ]");
    sw.Close();
    fs.Close();
}
```

5.2 Пример работы с двоичным файлом. Компонентами файла являются массивы из 10 элементов. Программа последовательно считывает из файла эти массивы, и определяет для массивов с четными номерами максимальное значение, с нечетными – минимальное.

Внешний вид работающей программы показан на рисунке 3.4.

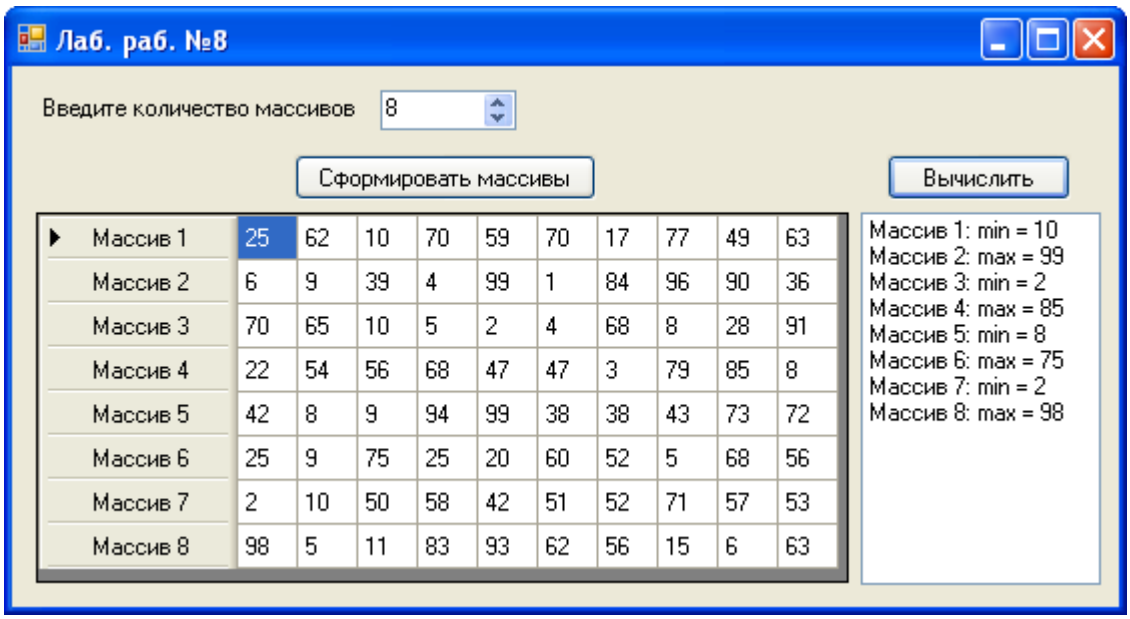


Рисунок 3.4 – Главное окно приложения

Выбор двоичного файла для работы осуществляется с помощью компонента **SaveFileDialog** (стандартное диалоговое окно выбора файла для записи), который нужно поместить на форму. Работа с ним аналогична работе с компонентом **OpenFileDialog**, которая была описана выше.

Для проверки конца файла при чтении из двоичного файла используется метод **PeekChar()**, считывающий очередное значение из файла без передвижения файлового указателя. Если он возвращает положительное значение, значит, конец файла еще не достигнут.

При проектировании формы свойство **Value** компонента **NumericUpDown** следует установить в 1. В таблицу **DataGridView** добавить 10 столбцов, установив им нужную ширину (свойство **Width**). Заголовки столбцов можно скрыть: свойство **ColumnHeadersVisible** равно **False**.

Коды обработчиков событий программы:

```
string fName;

//Обработчик кнопки "Сформировать массивы"
private void button1_Click(object sender, EventArgs e)
{
    //Объект для работы с генератором случайных чисел
    Random rnd = new Random();
    //байтовый массив
    byte[] mas = new byte[10];
    //количество массивов берем из счетчика на форме
    int kol = (int)numericUpDown1.Value;
    //установить соответствующее кол-во строк в таблице
    dataGridView1.RowCount = kol;

    //запросить у пользователя имя двоичного файла для записи
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
```

```
    {
        //получить имя выбранного файла
        fName = saveFileDialog1.FileName;
        FileStream fs = new FileStream(fName, FileMode.Create);
        BinaryWriter bw = new BinaryWriter(fs);
        //цикл по количеству массивов
        for (int i = 0; i < kol; i++)
        {
            //задать заголовок очередной строки в таблице
            dataGridView1.Rows[i].HeaderCell.Value = "Массив " +
                          (i + 1).ToString();
            //цикл по элементам массива
            for (int j = 0; j < 10; j++)
            {
                //очередной элемент массива - случайное число
                mas[j] = (byte)(rnd.Next(1, 100));
                //отобразить его в таблице
                dataGridView1.Rows[i].Cells[j].Value = mas[j];
            }
            //записать массив в файл
            bw.Write(mas);
        }
        bw.Close();
        fs.Close();
    }
}

//функция нахождения максимума или минимума в массиве
//(в зависимости от параметра choice)
byte MaxMin(byte[] a, string choice)
{
    byte M = a[0];
    if (choice == "Max")
        for (int i = 1; i < 10; i++)
            if (a[i] > M)
                M = a[i];
    if (choice == "Min")
        for (int i = 1; i < 10; i++)
            if (a[i] < M)
                M = a[i];
    return M;
}

//Обработчик кнопки "Вычислить"
private void button2_Click(object sender, EventArgs e)
{
    //очистить список на форме
    listBox1.Items.Clear();
    FileStream fs = new FileStream(fName, FileMode.Open);
    BinaryReader br = new BinaryReader(fs);
    byte[] mas = new byte[10];
    //номер очередного прочитанного из файла массива
    int n = 0;
```

```
    //строка-результат для вывода
    string res = "";
    //пока не дошли до конца файла
    while (br.PeekChar() >= 0)
    {
        //считать из файла очередной массив
        mas = br.ReadBytes(10);
        n++;
        res = "Массив " + n.ToString() + ": ";
        //если его номер четный - найти Max, иначе - Min
        if (n % 2 == 0)
            res += ("max = " + MaxMin(mas, "Max").ToString());
        else
            res += ("min = " + MaxMin(mas, "Min").ToString());
        //вывод результата в ListBox
        listBox1.Items.Add(res);
    }
    br.Close();
    fs.Close();
}

//Обработчик события загрузки формы
private void Form1_Load(object sender, EventArgs e)
{
    //По умолчанию - один массив, т.е. одна строка в таблице
    dataGridView1.RowCount = (int)numericUpDown1.Value;
    dataGridView1.Rows[0].HeaderCell.Value = "Массив 1";
}
```

6 Задания для самостоятельной работы

6.1 Текстовые файлы

Примечание. Исходные текстовые файлы можно создавать с использованием редактора «Блокнот».

1. В файле data.txt записан некоторый текст. Написать программу, которая подсчитывает количество букв латинского и русского алфавита и вычисляет процентное отношение между ними.

2. Дано два текстовых файла. Написать функцию, которая поменяет самую длинную строку первого файла с самой короткой строкой второго файла и наоборот.

3. Описать функцию triangle, формирующую текстовый файл из 9 строк, в первой из которых – один символ '1', во второй – два символа '2', …, в девятой – девять символов '9'.

4. В файле data.txt построчно записан некоторый текст. Найти строку максимальной длины и ее позицию.

5. Дан текстовый файл. Написать функцию, которая закодирует его, т.е. каждую строку текста перепишет в обратном порядке.

6. В файле data.txt записан некоторый текст построчно. В каждой строке отсортировать слова по возрастанию их длины.

7. Описать функцию line40, которая считывает из входного файла сим-

волы до первой точки и записывает их (без точки) в текстовый файл, формируя в нем строки по 40 символов (в последней строке символов может быть меньше).

8. Дан текстовый файл. Подсчитать количество знаков препинания (X) и количество слов (У). Если выполняется условие $\frac{Y}{X} \le 6$, то вывести сообщение о том, что текст в достаточной мере обогащен знаками препинания, иначе – недостаточно.

9. Описать функцию, которая подсчитывает количество пустых строк в текстовом файле.

10. Описать функцию для подсчета числа строк, начинающихся и заканчивающихся одним и тем же символом.

11. Описать функцию для подсчета числа строк, состоящих из одинаковых символов.

12. Дан текстовый файл. Написать функцию PersName(<имя_файла>), которая удалит из текста все имена собственные.

13. Дан текстовый файл. Написать программу, которая, игнорируя исходное деление этого файла на строки, переформатирует его, разбивая на строки так, чтобы каждая строка оканчивалась точкой либо содержала ровно 60 символов, если среди них нет точки.

14. Описать функцию, которая преобразовывает исходный файл с выравниванием текста по левому краю в файл с выравниванием текста по правому краю. Длину строки считать равной 80 символам.

15. Описать функцию fromto, переписывающую содержимое одного текстового файла в другой, но без пустых строк.

16. Дан текстовый файл. Написать функцию, которая строки, содержащие больше десяти символов, заменит на цифры (1,2,3,4,…) по порядку.

17. В текстовом файле записана непустая последовательность целых чисел, разделенных пробелами. Описать функцию positive, записывающую в другой текстовый файл все положительные числа из исходного файла.

18. Описать функцию lines, которая построчно печатает содержимое непустого текстового файла, вставляя в начало каждой строки ее порядковый номер (он должен занимать 4 позиции) и пробел.

19. Дан текстовый файл, состоящий из N строк. Написать функцию Trans(k1, k2), которая меняет местами две строки файла с номерами k1 и k2.

20. Дан текстовый файл. Написать функцию DoubleString(n), удваивающую в тексте каждый символ в строке с номером n.

21. Дан текстовый файл без разбиения на абзацы. Написать функцию, которая ищет строки, заканчивающиеся точками, и делает после них новый абзац. Абзацем считать отступ от края на 6 пробелов.

22. Дан текстовый файл. Написать функцию replace(c1, c2), заменяющую в тексте все символы c1 на символы c2.

23. Дан текстовый файл. Написать процедуру firsts, оставляющую в каждой строке исходного текста только первые буквы каждого слова.

24. Дано два текстовых файла. Написать функции, которая запишет в

третий файл те строки, которые встречаются в этих двух файлах.

25. Дан текстовый файл. Поменять местами первую и N-ную строку (N вводится с клавиатуры).

26. Дан текстовый файл, строки которого состоят из заглавных и строчных букв. Записать текст в новый файл таким образом, чтобы заглавные буквы стали строчными и наоборот.

27. Дан текстовый файл. Написать функцию, которая проверяет, есть ли в нем слова, состоящие только из цифр, если есть – удалить их.

28. Файл содержит текст, записанный в виде криптограммы, в которой буквы истинного текста размещаются в позициях, кратных 3. Напечатать исходный текст.

29. Дан текст. Заменить в каждой строке символы, расположенные между скобками ‘(‘, ‘)’ на ‘–’. Предполагается, что внутри каждой пары скобок нет других скобок.

30. Дан текстовый файл из латинских букв. Написать функцию, которая подсчитывает общее количество гласных букв и количество каждой гласной в отдельности. Результаты записать в новый файл.

31. Дан текстовый файл. Вывести на экран среднюю строку (две строки, если четное количество строк).

32. Описать функцию, которая построчно переписывает в другой файл содержимое текстового файла, вставляя в начало и в конец каждой строки через пробел знак восклицания.

6.2 Двоичные файлы

Указание к работе. Двоичные файлы создавать программно с использованием отдельной функции.

1. Дан некоторый файл, компоненты которого являются целыми числами. Найти сумму компонент первой половины файла и произведение второй половины.

2. Файл содержит список лотерейных билетов с четырехзначными номерами. Выигрышными считаются те билеты, сумма первых 3 цифр которых равна 8. Найти все выигрышные билеты и записать их в новый файл.

3. Дан некоторый файл, компонентами которого являются структуры типа: день, месяц, год. Описать функцию, проверяющую по сегодняшней дате (введенной с клавиатуры), какая из дат в файле является ближайшей к сегодняшнему дню.

4. Дан символьный файл. Удалить из него все записи, расположенные между заглавными буквами латинского алфавита и сами заглавные буквы. Например: jsdgfj**A**d12jfhgjf8**K**6775jjjsdhjh6365. В файле останется последовательность jsdgfj6775jjjsdhjh6365.

5. Дан файл, компонентами которого являются целые числа. Отсортировать их по возрастанию и записать в результирующий файл, при этом между каждой их парой вставить число, большее предыдущего, но меньшее последующего. Например, если после сортировки имеем последовательность 1,4,9,20, тогда в результате получим 1,***3***,4,***5***,9,***15***,20. (Примечание: исходный

файл заполнять так, чтобы в нем не оказалось одинаковых и рядом стоящих чисел).

6. Записать в новый файл все символы из некоторого символьного файла chrs.dat, не являющиеся буквами. Определить, сколько таких символов.

7. Дан некоторый файл, компоненты которого являются целыми числами. Подсчитать количество компонент, у которых значение совпадает с номером позиции. Например: 0 1 4 6 7 5. Таких компонент три: 0 (номер позиции=0), 1 (номер позиции=1) и 5 (номер позиции=5).

8. Дан некоторый файл, компоненты которого являются вещественными числами. Найти разность сумм первых трех и последних трех компонент файла.

9. В файле содержатся номера автомобилей, стоящих на стоянке (123, 467, 129 и др.). Вывести сначала список автомобилей с тремя одинаковыми цифрами, потом с двумя и в конце все остальные.

10. Дан файл, компонентами которого являются структуры типа: день, месяц, год. Удалить из файла все записи, следующие за некоторой датой, введенной с клавиатуры (если такой даты нет - то дописать ее в конец файла).

11. Некоторая упорядоченная последовательность целых чисел записана в файл. Разбить последовательность на две части и вывести на экран сначала вторую часть, затем первую. Обе части выводить в обратном порядке.

12. Дан файл, представляющий собой телефонную книжку некоторого молодого человека, т.е. только семизначные номера сотовых телефонов. У своей новой подруги он взял номер телефона и, естественно, пока дошел до дома, забыл его. Однако он помнит, что номер начинается на 497****. Написать функцию, которая выведет на экран только те номера из телефонной книги, среди которых может быть номер новой подруги.

13. Дан файл, содержащий данные о студентах: ФИО студента и оценки по предметам: программирование, высшая математика, политология, дискретная математика. Написать функцию, которая запишет в новый файл только тех студентов, средний бал сессии которых не меньше 4.

14. Даны два файла, компонентами которых являются вещественные числа. Описать функцию, проверяющую, равны ли файлы между собой.

15. Дан файл, компонентами которого являются структуры типа: день, месяц, год. Записать в другой файл даты, которые меньше некоторой даты, введенной с клавиатуры.

16. Дан некоторый файл, компоненты которого являются вещественными числами. Найти сумму максимального среди четных и минимального среди нечетных компонентов.

17. Дан символьный файл. Удалить из него все пробелы и специальные символы.

18. Файл содержит шестизначные номера проездных билетов. Написать функцию, которая запишет в новый файл только номера «счастливых» билетов. («Счастливыми» считать те билеты, сумма цифр первой половины которых равна сумме цифр второй половины).

19. Дан некоторый файл, компонентами которого являются структуры

типа: день, месяц, год. Вывести на экран сначала все зимние даты, затем весенние и т.д.

20. Дан некоторый файл, компоненты которого являются целыми числами. Подсчитать количество нулей, стоящих на четных местах.

21. Записать в новый файл все символы из некоторого символьного файла chrs.dat, являющиеся цифрами. Проверить, все ли цифры из диапазона 0…9 попали в новый файл, вывести те, которые не попали.

22. Дан файл, содержащий данные о погоде: время года, дату, облачность и температуру. Студенты со своими преподавателями решили идти в поход. Проверить, удовлетворяют ли погодные условия данному мероприятию путем ввода интересующей даты. (Примечание: хорошей погодой считается: безоблачно, для зимы +5 и более градусов, для весны и осени +15 и более, для лета - +20 и более).

23. Дан символьный файл. Вывести на экран максимальную последовательность цифр. Например: jsdgfjd**12**jfhgjf**86775**jjjsdhjh**6365**. На экран будет выведено: 86775, так как длина этого фрагмента с цифрами больше длины (12)=2 и больше длины (6365)=4.

24. Дан некоторый файл, компонентами которого являются вещественные числа. Найти сумму компонент первой половины файла и произведение второй половины. Найти среднее арифметическое компонентов, стоящих на местах, кратных трем.

25. Дан файл file1.dat, компонентами которого являются структуры типа: фамилия, имя, отчество. Поля "имя" и "отчество" записаны в полном формате (например: Петров Петр Петрович). Переписать в файл file2.dat все записи по следующему образцу: Петров П.П.

26. Даны два символьных файла. Записать в новый файл все совпадающие компоненты исходных файлов.

27. Дан файл, содержащий данные о студентах: ФИО студента, оценки по пяти дисциплинам и номер телефона родителей. Написать функцию, которая запишет в новый файл те записи о студентах, для которых стоит позвонить родителям по поводу сдачи сессии, т.е. средний бал < 20.

28. Даны два файла с числами, компоненты которых упорядочены по возрастанию. Объединить эти файлы в третий, не нарушая упорядоченности.

29. Дан файл, содержащий данные о преподавателях двух кафедр: ФИО, предмет, который он ведет, и название кафедры. Написать функцию, которая выведет сначала сотрудников одной кафедры, затем другой. Список сотрудников каждой кафедры должен идти в алфавитном порядке.

30. Дан символьный файл. Вставить после каждой цифры пробел.

31. Дан символьный файл, содержащий пробелы. Удалить из файла все «однобуквенные» слова и лишние пробелы. Например: jkdfh h kjdfh j jf. В результате: jkdfh kjdfh jf.

32. Дан файл, содержащий данные о студентах, проживающих в общежитии: ФИО, номер курса, смену(1-3), в которую он должен дежурить на проходной общежития. Создать три файла: в первый записать дежурящих в первую смену, во второй – во вторую, в третий – в третью.

Лабораторная работа № 4 – Основы построения классов

1 Цель работы

Изучить основные принципы разработки классов в C# и получить представление о взаимосвязи классов с формами.

2 Порядок выполнения работы

- прочесть краткие теоретические сведения;
- выполнить задания раздела;
- составить отчет по лабораторной работе и защитить его у преподавателя.

3 Теоретическая часть

3.1 Основные понятия класса

Весь активный процесс C#-программы происходит в пределах класса. Класс является основой для создания объектов. В классе определяют данные и код, который работает с этими данными. Объекты являются экземплярами класса. Непосредственно инициализация переменных в объекте (переменных экземпляра) происходит в конструкторе. В классе могут быть определены несколько конструкторов, то есть класс является набором проектов, которые определяют, как строить объект.

3.2 Общий синтаксис класса

Класс – некоторый набор данных и методов. Классы играют основополагающую роль в языке программирования C#. Они нужны для объединения однотипных функций (или функций для работы с определёнными абстракциями) в совокупности. Это удобно для быстрого и эффективного наращивания функционала программы.

Общий синтаксис класса:

[атрибуты] [спецификаторы] class имя класса [: предки] тело класса

3.2 Разработка приложения Windows Forms

1) Нажать Файл -> Создать -> Проект, в появившемся окне выбрать “Приложение Windows Forms”, ввести имя проекта.

2) В форме Form1 разместить четыре элемента TextBox, один элемент GroupBox, Один элемент Button, три элемента Label.

3) Сгруппировать элементы так, как показано на рисунке 2, изменить значение поля Text элемента GroupBox1 на «Информация», сменить значение поля Name элементов TextBox1, TextBox2, TextBox3 на NameBox, AgeBox, ProfessionBox и ResultBox соответственно. Сменить значение поля Name элемента Button1 на StartButton

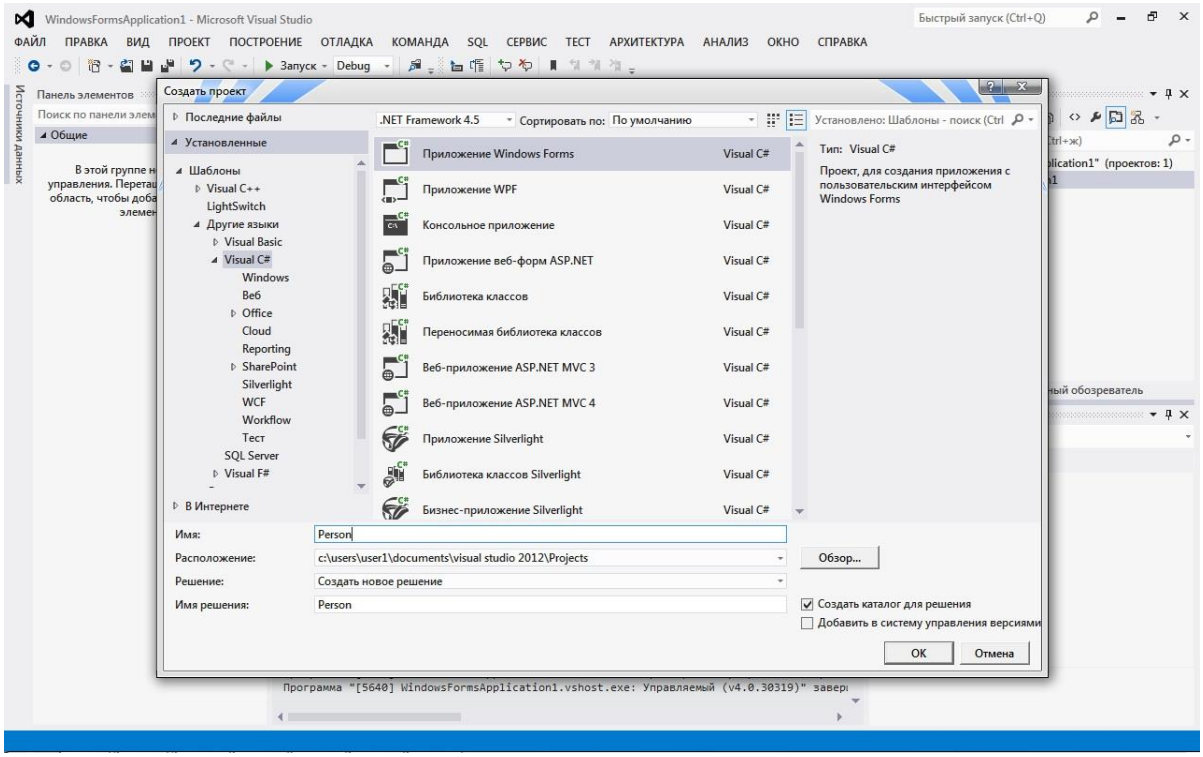


Рисунок 1 – Создание проекта

4) Поместить Элементы NameBox, AgeBox, ProfessionBox в GroupBox1, подписать их с помощью элементов Label1, Label2 и Label3(сменив значение поля Text элементов Label). Элемент ResultBox перевести в режим MultiLine (рисунок 2).

5) Значение поля Text элемента StartButton сменить на «Добавить».

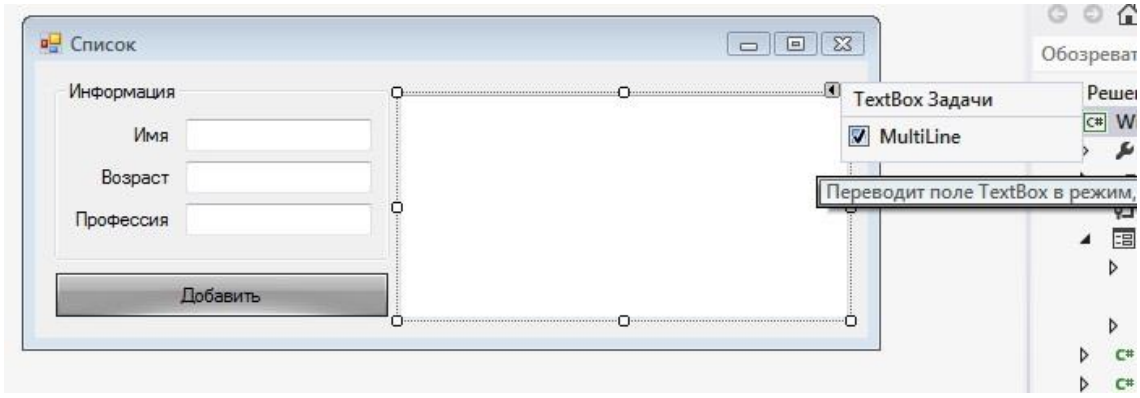


Рисунок 2 – Форма списка

3.3 Разработка класса Person

В окне Обозревателя Решений щелкнуть правой кнопкой по имени проекта, затем Добавить -> Класс. В появившемся окне ввести имя класса (Person) и нажать кнопку Добавить (см. рисунок 3).

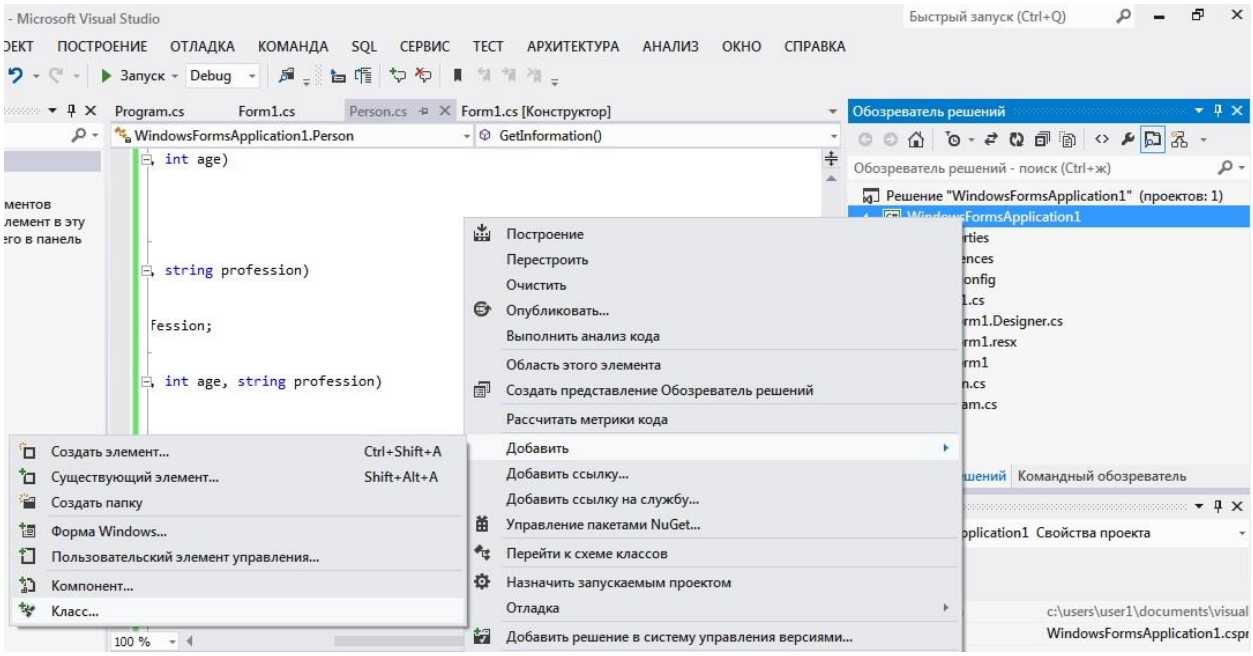


Рисунок 3 – Добавление класса

2) Объявить переменные name и profession типа string и переменную age типа integer:

```
string name;
int age;
string profession;
```

3) Объявить конструктор класса Person, который будет принимать значение поля name:

```
public Person(string name)
{
        this.name = name;
}
```

4) Перегрузить этот конструктор для различных параметров, которые могут передаваться при создании экземпляра класса.

Пример конструктора, принимающего значения полей name и age:

```
 public Person(string name, int age)
 {
        this.name = name;
        this.age = age;
 }
```

Пример конструктора, принимающего значения полей name и profession:

```
public Person(string name, string profession)
```

```
{
    this.name = name;
    this.profession = profession;
}
```

Пример конструктора, принимающего значения полей name, age и profession:

```
public Person(string name, int age, string profession)
{
    this.name = name;
    this.age = age;
    this.profession = profession;
}
```

3.4 Взаимосвязь класса с формой

1) Дважды щелкнуть по элементу StartButton, откроется код обработчика события нажатия кнопки

```
private void button1_Click(object sender, EventArgs e)
```

2) Объявить экземпляр класса Person, не вызывая при этом конструктор (необходимо, чтобы работать с объектом впоследствии, вызывая различные перегрузки конструктора для конкретной ситуации):

```
Person new_person;
```

3) Реализовать проверку на наличие текста в NameBox, если текста нет, то выводим сообщение пользователю путем создания диалогового окна MessageBox .

```
if (NameBox.Text == "")
{
    MessageBox.Show("Введите имя");
}
```

4) Реализовать конструкцию из условий, проверяющих либо наличие, либо отсутствие текста в элементах управления TextBox, в результате выполнения условий, явно вызывать одну из перегрузок конструктора класса Person:

```
if (AgeBox.Text != "")
{
    if (ProfessionBox.Text != "")
    {
        new_person = new Person(NameBox.Text, Convert.ToInt32(AgeBox.Text), ProfessionBox.Text);
    }
    else
    {
        new_person = new Person(NameBox.Text, Convert.ToInt32(AgeBox.Text));
    }
```

```
        }
        else
        {
        if (ProfessionBox.Text != "")
        {
                new_person = new Person(NameBox.Text, 0,ProfessionBox.Text);
        }
        else
        {
                new_person = new Person(NameBox.Text);
        }
}
```

3.5 Вывод результатов

1) Для вывода результатов работы программы, необходимо использовать метод GetInformation() класса Person, задача которого будет выводить строку, в которой будет содержаться текущая информация об экземпляре класса: поля name, age, profession (см. рисунок 4).

```
public string GetInformation()
{
        string information;
        information = "Имя: " + this.name + "; Возраст: " + this.age.ToString() + ";
        Профессия: " + this.profession;
        return information;
}
```

Синтаксис в коде:

```
 ResultBox += new_person.GetInformation();
```

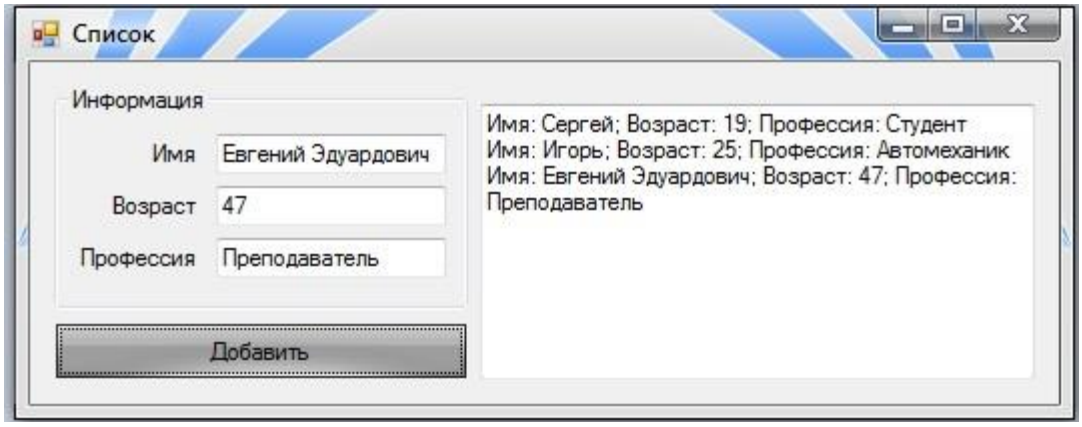


Рисунок 4 – Результат в форме

4 Задания для выполнения работы

Каждый разрабатываемый класс должен, как правило, содержать следующие элементы: скрытые поля, конструкторы с параметрами и без параметров, методы, свойства. Методы и свойства должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. При возникновении ошибок должны выбрасываться исключения.
В программе должна выполняться проверка всех разработанных элементов класса.

1.Описать класс, реализующий десятичный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и свойство, позволяющее получить его текущее состояние. При выходе за границы диапазона выбрасываются исключения.
Написать программу, демонстрирующую все разработанные элементы класса.
2.Описать класс, реализующий шестнадцатеричный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и свойство,
позволяющее получить его текущее состояние. При выходе за границы диапазона выбрасываются исключения.
Написать программу, демонстрирующую все разработанные элементы класса.
3.Описать класс, представляющий треугольник. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и вращения на заданный угол. Описать свойства для получения состояния объекта. При невозможности построения треугольника выбрасывается исключение.
Написать программу, демонстрирующую все разработанные элементы класса.
4.Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса и проверки допустимости вводимых значений. В случае недопустимых значений полей выбрасываются исключения.
Написать программу, демонстрирующую все разработанные элементы класса.
5.Составить описание класса для представления комплексных чисел. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.
Написать программу, демонстрирующую все разработанные элементы класса.

6.Составить описание класса для вектора, заданного координатами его концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.
Написать программу, демонстрирующую все разработанные элементы класса.
7.Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменение размеров, построение наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.
Написать программу, демонстрирующую все разработанные элементы класса.
8.Составить описание класса для представления даты. Предусмотреть возможности установки даты и изменения ее отдельных полей (год, месяц, день) с проверкой допустимости вводимых значений. В случае недопустимых значений полей выбрасываются исключения. Создать методы изменения даты на заданное количество дней, месяцев и лет.
Написать программу, демонстрирующую все разработанные элементы класса.
9.Составить описание класса для представления времени. Предусмотреть возможности установки времени и изменения его отдельных полей (час, минута, секунда) с проверкой допустимости вводимых значений. В случае недопустимых значений полей выбрасываются исключения. Создать методы изменения времени на заданное количество часов, минут и секунд.
Написать программу, демонстрирующую все разработанные элементы класса.
10.Составить описание класса многочлена вида $ax^2 + bx + c$. Предусмотреть методы, реализующие:
- вычисление значения многочлена для заданного аргумента;
- операцию сложения, вычитания и умножения многочленов с получением нового объекта-многочлена;
- вывод на экран описания многочлена.

Написать программу, демонстрирующую все разработанные элементы класса.
11.Описать класс, представляющий треугольник. Предусмотреть методы для создания объектов, вычисления площади, периметра и точки пересечения медиан. Описать свойства для получения состояния объекта. При невозможности построения треугольника выбрасывается исключение.
Написать программу, демонстрирующую все разработанные элементы класса.
12.Описать класс, представляющий круг. Предусмотреть методы для создания объектов, вычисления площади круга, длины окружности и проверки по-

падания заданной точки внутрь круга. Описать свойства для получения состояния объекта.
Написать программу, демонстрирующую все разработанные элементы класса.
13.Описать класс для работы со строкой, позволяющей хранить только двоичное число и выполнять с ним арифметические операции. Предусмотреть инициализацию с проверкой допустимости значений. В случае недопустимых значений выбрасываются исключения.
Написать программу, демонстрирующую все разработанные элементы класса.
14.Описать класс дробей — рациональных чисел, являющихся отношением двух целых чисел. Предусмотреть методы сложения, вычитания, умножения и деления дробей.
Написать программу, демонстрирующую все разработанные элементы класса.
15 .Описать класс «файл», содержащий сведения об имени, дате создания и длине файла. Предусмотреть инициализацию с проверкой допустимости значений полей. В случае недопустимых значений полей выбрасываются исключения. Описать метод добавления информации в конец файла и свойства для получения состояния файла.
Написать программу, демонстрирующую все разработанные элементы класса.
16.Описать класс «комната», содержащий сведения о метраже, высоте потолков и количестве окон. Предусмотреть инициализацию с проверкой допустимости значений полей. В случае недопустимых значений полей выбрасываются исключения. Описать методы вычисления площади и объема комнаты и свойства для получения состояния объекта.
Написать программу, демонстрирующую все разработанные элементы класса.
17.Описать класс, представляющий нелинейное уравнение вида $ax - cos(x) = 0$. Описать метод, вычисляющий решение этого уравнения на заданном интервале методом деления пополам (см. раздел «Цикл с параметром for») и выбрасывающий исключение в случае отсутствия корня. Описать свойства для получения состояния объекта.
Написать программу, демонстрирующую все разработанные элементы класса.
18.Описать класс, представляющий квадратное уравнение вида $ax^2 + bx + c = 0$. Описать метод, вычисляющий решение этого уравнения и выбрасывающий исключение в случае отсутствия корней. Описать свойства для получения состояния объекта.
Написать программу, демонстрирующую все разработанные элементы класса.
19.Описать класс «процессор», содержащий сведения о марке, тактовой частоте, объеме кэша и стоимости. Предусмотреть инициализацию с проверкой допустимости значений полей. В случае недопустимых значений полей вы-

брасываются исключения. Описать свойства для получения состояния объекта. Описать класс «материнская плата», включающий класс «процессор» и объем установленной оперативной памяти. Предусмотреть инициализацию с проверкой допустимости значений поля объема памяти. В случае недопустимых значений поля выбрасывается исключение. Описать свойства для получения состояния объекта.
Написать программу, демонстрирующую все разработанные элементы классов.
20.Описать класс «цветная точка». Для точки задаются координаты и цвет. Цвет описывается с помощью трех составляющих (красный, зеленый, синий). Предусмотреть различные методы инициализации объекта с проверкой допустимости значений. Допустимым диапазоном для каждой составляющей является [0, 255]. В случае недопустимых значений полей выбрасываются исключения. Описать свойства для получения состояния объекта и метод изменения цвета.
Написать программу, демонстрирующую все разработанные элементы класса.

Лабораторная работа №5. Разработка приложений с использованием коллекций

1. Цель работы

Изучить интерфейсы и классы коллекций библиотеки .NET Framework, основные свойства и методы этих классов, применяемые при работе с коллекциями.

2. Сведения из теории

2.1. Интерфейсы коллекций

В C# под *коллекцией* понимается группа объектов. Пространство имен **`System.Collections`** содержит множество интерфейсов и классов, которые определяют и реализуют коллекции различных типов (динамические массивы, стеки, очереди, словари, хеш-таблицы и т.п.).

Интерфейсы, которые поддерживают коллекции и их иерархия приведены на рисунке 1.1.

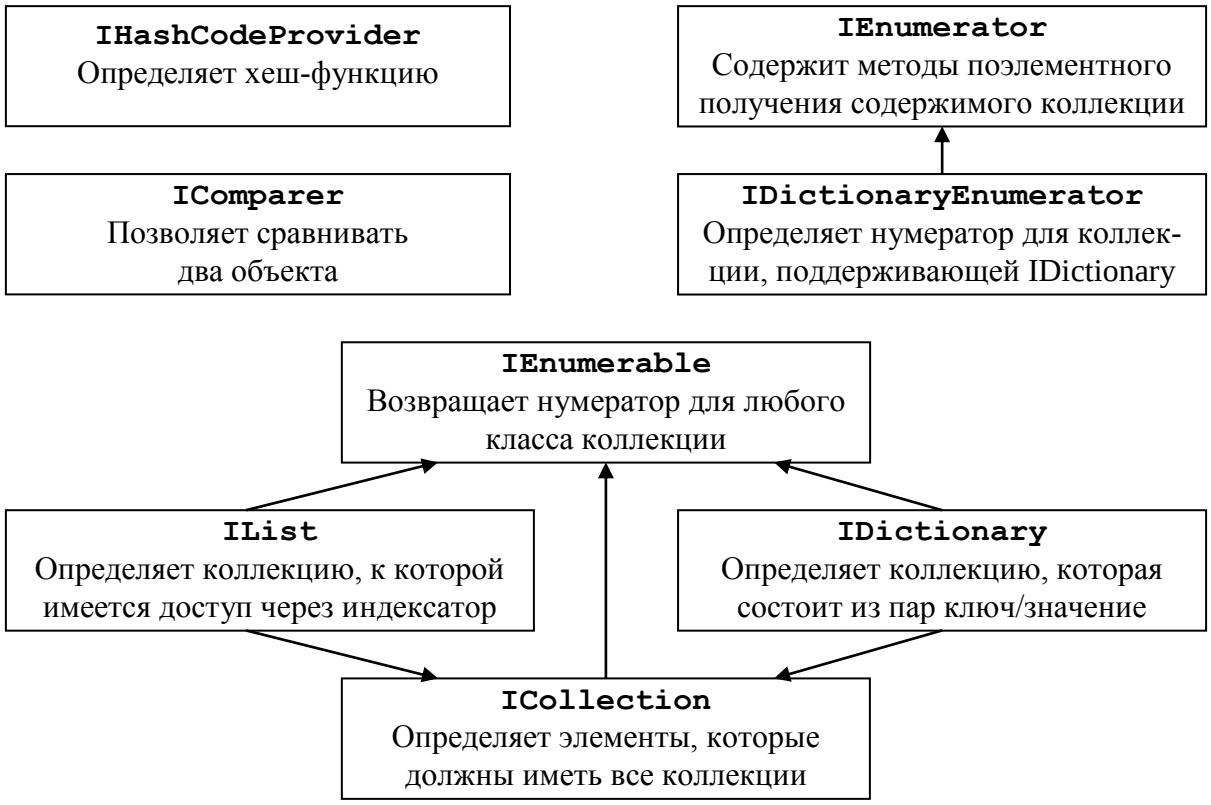


Рис. 1.1. Иерархия интерфейсов System.Collections

Интерфейс **`ICollection`** можно назвать фундаментом, на котором построены все коллекции. В нем объявлены основные методы и свойства, без которых не может обойтись ни одна коллекция. Самое востребованное свойство этого интерфейса – **`Count`**, содержит количество элементов, хранимых в коллекции в данный момент. Если свойство **`Count`** равно нулю, значит, коллекция пуста. В интерфейсе **`ICollection`** определен следующий метод:

void CopyTo(Array target, int startIdx)

Этот метод копирует содержимое коллекции в массив, заданный параметром **target**, начиная с индекса, заданного параметром **startIdx**. Можно сказать, что метод **CopyTo()** обеспечивает переход от коллекции к стандартному C#-массиву.

Интерфейс **IList** наследует интерфейс **ICollection** и определяет поведение коллекции, доступ к элементам которой разрешен посредством индекса с отсчетом от нуля. Помимо методов, определенных в интерфейсе **ICollection**, интерфейс **IList** определяет и собственные методы:

Метод	Описание
int Add(object obj)	Добавляет объект **obj** в вызывающую коллекцию. Возвращает индекс, по которому этот объект сохранен
void Clear()	Удаляет все элементы из вызывающей коллекции
bool Contains (object obj)	Возвращает **true**, если вызывающая коллекция содержит объект **obj**, иначе **false**
int IndexOf (object obj)	Возвращает индекс объекта **obj**, если он (объект) содержится в вызывающей коллекции, иначе возвращает –1
void Insert(int idx, object obj)	Вставляет в вызывающую коллекцию объект **obj** по индексу **idx**. Последующие элементы смещаются вперед, чтобы освободить место для вставляемого объекта
void Remove(object obj)	Удаляет из вызывающей коллекции первое вхождение объекта **obj**. Последующие элементы смещаются назад
void RemoveAt(int idx)	Удаляет из вызывающей коллекции объект, расположенный по индексу **idx**. Последующие элементы смещаются назад

В интерфейсе **IList** определены следующие свойства:

bool IsFixedSize { get; } – равно **true**, если коллекция имеет фиксированный размер, т.е. в нее нельзя вставлять элементы и удалять их из нее.

bool IsReadOnly { get; } – равно **true**, если коллекция предназначена только для чтения, т.е. содержимое коллекции не подлежит изменению.

В интерфейсе **IList** определен следующий индексатор:

object this[int idx] { get; set; }

Этот индексатор можно использовать для считывания или записи значения нужного элемента (но не для добавления).

Интерфейс **IDictionary** определяет поведение коллекции, которая служит для хранения пар ключ/значение. Сохраненную однажды пару можно

извлечь по заданному ключу. Интерфейс **IDictionary** наследует интерфейс **ICollection**. Методы, объявленные в интерфейсе **IDictionary**, сведены в таблице:

Метод	Описание
`void Add(object k, object v)`	Добавляет в вызывающую коллекцию пару ключ/значение, заданную параметрами **k** и **v**. Ключ **k** не должен быть нулевым. Если ключ **k** уже хранится в коллекции, генерируется исключение типа **ArgumentException**
`void Clear()`	Удаляет все пары ключ/значение из вызывающей коллекции
`bool Contains(object k)`	Возвращает **true**, если вызывающая коллекция содержит объект **k** в качестве ключа, иначе **false**
`IDictionaryEnumerator GetEnumerator()`	Возвращает нумератор для вызывающей коллекции
`void Remove(object k)`	Удаляет элемент с ключом **k**

В интерфейсе **IDictionary** определены следующие свойства:

Свойство	Описание
`bool IsFixedSize {get; }`	Равно **true**, если словарь имеет фиксированный размер
`bool IsReadOnly { get; }`	Равно **true**, если словарь предназначен только для чтения
`ICollection Keys { get; }`	Получает коллекцию ключей
`ICollection Values { get; }`	Получает коллекцию значений

В интерфейсе **IDictionary** определен следующий индексатор:

```
object this[object key] { get; set; }
```

Этот индексатор можно использовать для получения или установки значения элемента, а также для добавления в коллекцию нового элемента. Следует обратить внимание на то, что «индекс» в данном случае является не обычным индексом, а ключом элемента.

Интерфейс **IEnumerable** должен быть реализован в любом классе, если в нем предполагается поддержка нумераторов. В этом интерфейсе определен единственный метод:

```
IEnumerator GetEnumerator()
```

Он возвращает нумератор для коллекции. Кроме того, реализация интерфейса **IEnumerable** позволяет получить доступ к содержимому коллекции с помощью цикла **foreach**.

Интерфейс **IEnumerator**, собственно, и определяет действие любого нумератора. Используя его методы, можно циклически опросить элементы кол-

лекции. Для коллекций, в которых хранятся пары ключ/значение (т.е. словари), метод **GetEnumerator()** возвращает объект типа **IDictionaryEnumerator**, а не **IEnumerator**. Интерфейс **IDictionaryEnumerator** является производным от интерфейса **IEnumerator** и распространяет свои функциональные возможности нумератора на область словарей.

В интерфейсе **IEnumerator** определено единственное свойство:

```
object Current { get; }
```

Это свойство позволяет получить (но не установить) элемент, соответствующий текущему значению нумератора.

В интерфейсе **IEnumerator** определены два метода:

Метод	Описание
bool MoveNext()	Перемещает текущую позицию нумератора к следующему элементу коллекции. Возвращает **false**, если достигнут конец коллекции, иначе **true**. До первого обращения к этому методу значение свойства **Current** не определено
void Reset()	Устанавливает нумератор в начало коллекции

Интерфейс **IDictionaryEnumerator** дополнительно определяет «свои» свойства:

Свойство	Описание
DictionaryEntry Entry { get; }	Получает пару ключ/значение в форме структуры типа **DictionaryEntry**. В этой структуре определены два свойства, **Key** и **Value**, которые можно использовать для доступа к ключу или значению, относящемуся к соответствующему элементу
object Key {get;}	Получает прямой доступ к ключу
object Value {get;}	Получает прямой доступ к значению

В интерфейсе **IComparer** определен метод, который позволяет сравнивать два объекта:

```
int Compare (object v1, object v2)
```

Этот метод возвращает положительное число, если значение **v1** больше значения **v2**, отрицательное, если **v1** меньше **v2**, и ноль, если сравниваемые значения равны. Этот интерфейс можно использовать для задания способа сортировки элементов коллекции.

Интерфейс **IHashCodeProvider** должен быть реализован коллекцией, если необходимо определить собственную версию метода **GetHashCode()**. Все объекты (для получения хеш-кода) наследуют метод **Object.GetHashCode()**, который используется по умолчанию. посредством реализации интерфейса **IHashCodeProvider** можно определить альтернатив-

ный метод.

2.2. Классы коллекций

Классы коллекций делятся на три основных категории: общего назначения, специализированные и ориентированные на побитовую организацию данных. Классы общего назначения можно использовать для хранения объектов любого типа. Битовые предназначены для хранения битовой информации. Коллекции специального назначения разрабатываются для обработки данных конкретного типа. В данной работе рассматриваются только классы коллекций общего назначения. Эти классы перечислены в таблице:

Класс	Описание
ArrayList	Динамический массив, т.е массив, который при необходимости может увеличивать свой размер
HashTable	Хеш-таблица для пар ключ/значение
Queue	Очередь
SortedList	Отсортированный список пар ключ/значение
Stack	Стек

Класс **ArrayList** предназначен для поддержки динамических массивов, которые при необходимости могут увеличиваться или сокращаться. Объект этого класса представляет собой массив переменной длины, элементами которого являются объектные ссылки. Любой объект класса **ArrayList** создается с некоторым начальным размером. При превышении этого размера коллекция автоматически его увеличивает. В случае удаления объектов массив можно сократить.

Класс **ArrayList** реализует интерфейсы **ICollection**, **IList**, **IEnumerable** и **ICloneable**. В классе определены следующие конструкторы:

Прототип конструктора	Описание
public ArrayList()	Создает пустой **ArrayList**-массив с начальной емкостью в 16 элементов
public ArrayList (ICollection c)	Создает массив, который инициализируется элементами и емкостью коллекции **c**
public ArrayList (int capacity)	Создает массив с заданной начальной емкостью

Помимо методов, определенных в интерфейсах, которые реализует класс **ArrayList**, в нем определены и собственные методы:

Метод	Описание
public virtual void AddRange(ICollection c)	Добавляет элементы из коллекции **c** в конец вызывающей коллекции
public virtual int BinarySearch(object v)	Выполняет поиск элемента **v**. Возвращает его индекс или отрицательное значение, если эле-

	мента нет. Вызывающий список должен быть отсортирован
public virtual int BinarySearch(object v, IComparer comp)	Выполняет поиск элемента **v** на основе метода сравнения объектов, заданного параметром **comp**. Возвращает индекс элемента или отрицательное значение, если элемента нет. Вызывающий список должен быть отсортирован
public virtual int BinarySearch(int startIdx, int count, object v, IComparer comp)	Выполняет поиск элемента **v** на основе метода сравнения объектов, заданного параметром **comp**. Поиск начинается с элемента **startIdx** и включает **count** элементов. Возвращает индекс элемента или отрицательное значение, если элемента нет. Вызывающий список должен быть отсортирован
public virtual void CopyTo(Array ar, int startIdx)	Копирует содержимое вызывающей коллекции, начиная с элемента с индексом **startIdx,** в массив **ar**
public virtual void CopyTo(int srcIdx, Array ar, int destIdx, int count)	Копирует **count** элементов вызывающей коллекции, начиная с индекса **startIdx**, в массив **ar**, начиная с элемента с индексом **destIdx**
public virtual ArrayList GetRange(int idx, int count)	Возвращает часть вызывающей коллекции. Диапазон начинается с индекса **idx** и включает **count** элементов.
public static ArrayList FixedSize(ArrayList ar)	Превращает коллекцию **ar** в **ArrayList**-массив с фиксированным размером и возвращает результат
public virtual void InsertRange(int startIdx,ICollection c)	Вставляет элементы коллекции с в вызывающую коллекцию, начиная с индекса **startIdx**
public virtual int LastIndexOf(object v)	Возвращает индекс последнего вхождения объекта **v** в вызывающую коллекцию (-1, если объект не обнаружен)
public static ArrayList ReadOnly(ArrayList ar)	Превращает коллекцию **ar** в **ArrayList**-массив, предназначенный только для чтения, и возвращает результат
public virtual void RemoveRange(int idx, int count)	Удаляет **count** элементов из вызывающей коллекции, начиная с элемента с индексом **idx**
public virtual void Reverse()	Располагает элементы вызывающей коллекции в обратном порядке
public virtual void Reverse(int startIdx, int count)	Располагает в обратном порядке **count** элементов вызывающей коллекции, начиная с индекса **startIdx**
public virtual void SetRange(int startIdx,	Заменяет элементы вызывающей коллекции, начиная с индекса **startIdx**, элементами кол-

ICollection c)	лекции **c**
public virtual void Sort()	Сортирует коллекцию по возрастанию
public virtual void Sort(IComparer comp)	Сортирует вызывающую коллекцию на основе метода сравнения объектов **comp**
public virtual void Sort(int startIdx, int endIdx, IComparer comp)	Сортирует часть вызывающей коллекции на основе метода сравнения объектов **comp**. Сортировка начинается с индекса **startIdx** и заканчивается индексом **endIdx**
public static ArrayList Synchronized(ArrayList list)	Возвращает синхронизированную версию вызывающей коллекции
public virtual object[] ToArray()	Возвращает массив, который содержит копии элементов вызывающего объекта
public virtual Array ToArray(Type type)	Возвращает массив, который содержит копии элементов вызывающего объекта. Тип элементов в этом массиве задается параметром **type**
public virtual void TrimToSize()	Устанавливает свойство **Capacity** равным значению свойства **Count**

Помимо свойств, определенных в интерфейсах, реализуемых классом **ArrayList**, в нем также определено свойство **Capacity**:

```
public virtual int Capacity { get; set; }
```

Оно позволяет узнать или установить емкость вызывающего массива. Емкость – это количество элементов, которые можно сохранить в массиве без его увеличения.

Класс **HashTable** предназначен для создания коллекции, в которой для хранения объектов используется хеш-таблица. Этот класс реализует интерфейсы **IDictionary**, **ICollection**, **IEnumerable**, **ISerializable**, **IDeserializationCallBack** и **ICloneable**.

В классе **HashTable** определено множество конструкторов, включая следующие (они используются чаще всего):

Прототип конструктора	Описание
public HashTable()	Создает стандартный объект класса **HashTable**
public HashTable (IDictionary c)	Инициализирует создаваемый объект элементами коллекции c
public HashTable (int capacity)	Инициализирует емкость создаваемой хеш-таблицы значением **capacity**
public HashTable (int capacity, float fillRatio)	Инициализирует емкость создаваемой хеш-таблицы (значением **capacity**) и коэффициент заполнения (значением **fillRatio**). Значение этого коэффициента (от 0,1 до 1,0) определяет степень заполнения хеш-

таблицы, после чего ее размер увеличивается

В классе **HashTable** помимо методов, определенных в реализуемых им интерфейсах, определены собственные методы. Наиболее употребляемые из них перечислены в таблице:

Метод	Описание
`public virtual bool ContainsKey(object k)`	Возвращает **true**, если в вызывающей хеш-таблице содержится ключ **k**, иначе **false**
`public virtual bool ContainsValue(object v)`	Возвращает **true**, если в вызывающей хеш-таблице содержится значение **v**, иначе **false**
`public virtual IDictionaryEnumerator GetEnumerator()`	Возвращает для вызывающей хеш-таблицы нумератор типа **IDictionaryEnumerator**
`public static HashTable Synchronized(HashTable ht)`	Возвращает синхронизированную версию хеш-таблицы **ht**

В классе **HashTable**, помимо свойств, определенных в реализуемых им интерфейсах, определены два собственных свойства. С их помощью можно из хеш-таблицы получить коллекцию ключей или значений:

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

Важно отметить, что **HashTable**-коллекция не гарантирует сохранения порядка элементов, т.к. хеширование обычно не применяется к отсортированным таблицам.

Класс **SortedList** предназначен для создания коллекции, которая хранит пары ключ/значение в упорядоченном виде, а именно отсортированы по ключу. Этот класс реализует интерфейсы **IDictionary, ICollection, IEnumerable** и **ICloneable.**

В классе **SortedList** определено несколько конструкторов, включая следующие:

Прототип конструктора	Описание
`public SortedList()`	Создает пустую коллекцию емкостью 16 элементов
`public SortedList (IDictionary c)`	Создает **SortedList**-коллекцию и инициализирует ее элементами коллекции **c**
`public SortedList (int capacity)`	Создает пустую коллекцию емкостью **capacity**
`public SortedList (IComparer comp)`	Создает пустую коллекцию емкостью 16 элементов и задает для них метод сравнения **comp**

В классе **SortedList** помимо методов, определенных в реализуемых им интерфейсах, также определены собственные методы. Наиболее часто упо-

требляемые из них перечислены в таблице:

Метод	Описание
`public virtual bool ContainsKey(object k)`	Возвращает **true**, если в вызывающей **SortList**-коллекции содержится ключ **k**, иначе **false**
`public virtual bool ContainsValue(object v)`	Возвращает **true**, если в вызывающей **SortList**-коллекции содержится значение **v**, иначе **false**
`public virtual object GetByIndex(int idx)`	Возвращает значение, индекс которого равен **idx**
`public virtual IDictionaryEnumerator GetEnumerator()`	Возвращает для вызывающей **SortList**-коллекции нумератор типа **IDictionaryEnumerator**
`public virtual object GetKey(int idx)`	Возвращает ключ, индекс которого равен **idx**
`public virtual IList GetKeyList()`	Возвращает **IList**-коллекцию ключей, хранимых в вызывающей **SortList**-коллекции
`public virtual IList GetValueList()`	Возвращает **IList**-коллекцию значений, хранимых в вызывающей **SortList**-коллекции
`public virtual int IndexOfKey(object k)`	Возвращает индекс ключа **k** или -1, если заданного ключа в списке нет
`public virtual int IndexOfValue(object v)`	Возвращает индекс первого вхождения значения v или -1, если этого значения в списке нет
`public virtual void SetByIndex(int idx, object v)`	Устанавливает значение по индексу **idx** равным значению v
`public static SortedList Synchronized(SortedList sl)`	Возвращает синхронизированную версию **SortList**-коллекции **sl**
`public virtual void TrimToSize()`	Устанавливает свойство **Capacity** равным значению свойства **Count**

В классе **SortedList**, помимо свойств, определенных в реализуемых им интерфейсах, определены два собственных свойства. Получить предназначенную только для чтения коллекцию ключей или значений, хранимых в **SortedList**-коллекции, можно с помощью свойств:

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

Класс коллекции, предназначенный для поддержки стека, называется **Stack**. Он реализует интерфейсы **ICollection, IEnumerable** и **ICloneable.** Стек – это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы, причем каждый раз, ко-

гда стек должен расшириться, его емкость удваивается.

В классе **Stack** определены следующие конструкторы:

Прототип конструктора	Описание
`public Stack()`	Создает пустой стек емкостью 10 элементов
`public Stack(int capacity)`	Создает пустой стек с начальной емкостью **capacity**
`public Stack (ICollection c)`	Создает стек, который инициализируется элементами коллекции **c**

Помимо методов, определенных в интерфейсах, которые реализует класс **Stack**, в нем определены также собственные методы, перечисленные в таблице:

Метод	Описание
`public virtual bool Contains(object v)`	Возвращает **true**, если в вызывающем стеке содержится объект **v**, иначе **false**
`public virtual void Clear()`	Очищает стек (устанавливает свойство **Count** равным нулю)
`public virtual object Peek()`	Возвращает элемент, расположенный в вершине стека, но не удаляет его
`public virtual object Pop()`	Возвращает элемент, расположенный в вершине стека, и удаляет его
`public virtual void Push(object v)`	Помещает объект **v** в стек
`public static Stack Synchronized(Stack stk)`	Возвращает синхронизированную версию стека **stk**
`public virtual object[] ToArray()`	Возвращает массив, который содержит копии элементов вызывающего стека

Класс коллекции, предназначенный для поддержки очереди, называется **Queue**. Он реализует интерфейсы **ICollection, IEnumerable** и **ICloneable.** Очередь – это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы, причем каждый раз, когда такая необходимость возникает, текущий размер очереди умножается на коэффициент роста, который по умолчания равен значению 2,0.

В классе **Queue** определены следующие конструкторы:

Прототип конструктора	Описание
`public Queue()`	Создает пустую очередь с начальной емкостью в 32 элемента и коэффициентом роста 2,0
`public Queue(int capacity)`	Создает пустую очередь с начальной емкостью **capacity** и коэффициентом роста 2,0

`public Queue(int capacity, float growFact)`	Создает пустую очередь с начальной емкостью **capacity** и коэффициентом роста **growFact**
`public Queue (ICollection c)`	Создает очередь, которая инициализируется элементами коллекции **c**

Помимо методов, определенных в интерфейсах, которые реализует класс **Queue**, в нем определены также собственные методы, перечисленные в таблице.

Метод	Описание
`public virtual bool Contains(object v)`	Возвращает **true**, если в вызывающей очереди содержится объект **v**, иначе **false**
`public virtual void Clear()`	Очищает очередь (устанавливает свойство **Count** равным нулю)
`public virtual object Dequeue()`	Возвращает объекта из начала вызывающей очереди, при этом объект из очереди удаляется
`public virtual void Enqueue(object v)`	Добавляет объект **v** в конец очереди
`public virtual object Peek()`	Возвращает элемент из начала вызывающей очереди, но не удаляет его
`public static Queue Synchronized(Queue q)`	Возвращает синхронизированную версию очереди **q**
`public virtual object[] ToArray()`	Возвращает массив, который содержит копии элементов вызывающей очереди
`public virtual void TrimToSize()`	Устанавливает свойство **Capacity** равным значению свойства **Count**

3. Пример выполнения работы

Задание: Дан список, содержащий точки с координатами (x,y). Создать новый список, в который попадут точки, лежащие левее оси ординат. Соответственно, в старом списке останутся точки, которые расположены справа от оси ординат. Предусмотреть возможность сортировки нового списка по возрастанию координаты x.

3.1. Визуальное проектирование диалогового окна

Внешний вид работающего приложения с описанием использованных компонентов приведен на рисунке 1.2.

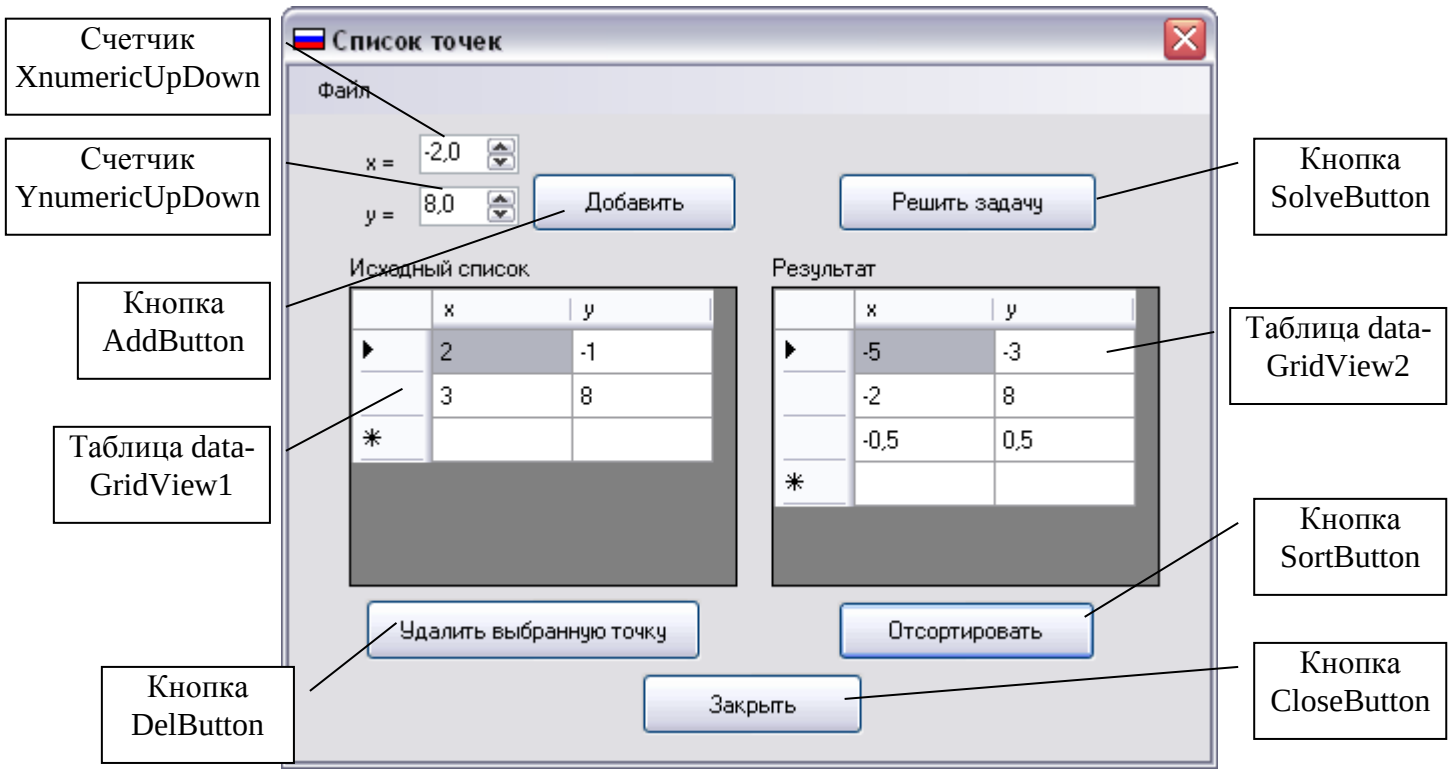


Рис. 1.2. Пример работающего приложения

Для счетчиков **`XnumericUpDown`** и **`YnumericUpDown`** устанавливаются следующие дополнительные свойства:

Свойство	Значение	Описание
`Minimum`	-100	Минимальное значение компонента
`Maximum`	100	Максимальное значение компонента
`Increment`	0,1	Шаг увеличения или уменьшения
`DecimalPlaces`	1	Количество знаков после запятой при отображении

Для таблиц данных **`dataGridView1`** и **`dataGridView2`** устанавливаются следующие дополнительные свойства:

Свойство	Значение	Описание
`ColumnHeadersHeight-SizeMode`	**`AutoSize`**	Регулирует высоту заголовков столбцов
`EditMode`	**`EditProgrammatically`**	Режим редактирования данных (здесь – только программно)
`Columns`	редактируется	Столбцы таблицы

Для добавления столбцов в таблицу следует нажать кнопку свойства **`Columns`**. Откроется диалоговое окно, где нужно добавлять и устанавливать свойства столбцов таблицы (рис. 1.3).

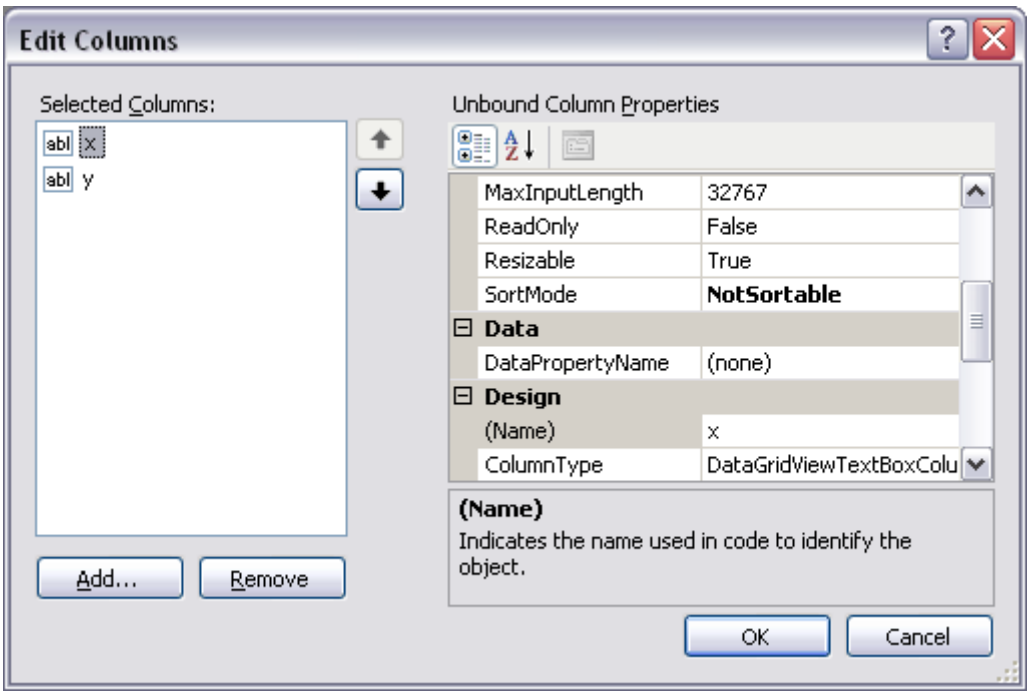


Рис. 1.3. Окно редактирования столбцов таблицы

В нашем случае добавляются 2 столбца с именами **`x`** и **`y`** (свойства **`Name`** и **`HeaderText`**) и для них отменяется режим автоматической сортировки строк (свойство **`SortMode`** устанавливается равным **`NotSortable`**).

3.2. Проектирование программного кода

3.2.1. Разработка класса точки

Проектирование кода начинается с разработки класса, представляющего собой структуру точки. Чтобы добавить класс, нужно выбрать пункт меню **`Project->Add Class…`**, в появившемся диалоговом окне снова выбрать «**`Class`**» и указать имя создаваемого файла (и класса). В нашем случае класс называется «**`Point`**».

Так как в задании необходимо реализовать сортировку списка точек по одной из координат, программе (а именно методу **`Sort`**) необходимо явно указать, по какому именно признаку (по какой координате) следует производить сравнение точек при сортировке. Для этого нужно, чтобы класс реализовывал интерфейс **`IComparable`**.

Созданный класс будет содержать следующие компоненты:

– два закрытых поля типа **`float`**, задающих координаты точки;

– конструктор с двумя параметрами, инициализирующий эти поля;

– два открытых свойства для доступа к полям (чтения и установки им значений);

– реализацию открытого метода **`CompareTo`**, описанного в интерфейсе **`IComparable`** и задающего способ сравнения объектов класса. Этот метод возвращает целочисленное значение: положительное, если вызывающий объект больше объекта параметра, отрицательное, если меньше, и ноль, если объекты равны. Именно в этом методе мы указываем, что сравнение точек

должно производиться по координате **x**.

Ниже приведен листинг описания класса **Point**:

```
//класс реализует интерфейс IComparable
class Point : IComparable
{
    float X;    //координаты точки
    float Y;
    //конструктор класса
    public Point(float X, float Y)
    {
        this.X = X;
        this.Y = Y;
    }
    //свойство для доступа к координате x
    public float x
    {
        get { return X; }   //получить x
        set { X = value; }  //установить x
    }
    //свойство для доступа к координате y
    public float y
    {
        get { return Y; }  //получить y
        set { Y = value; } //установить y
    }
    //установить способ сравнения объектов-точек
    public int CompareTo(object obj)
    {
        //преобразуем параметр obj к типу точки
        Point p = (Point)obj;
        if (x > p.x)   //сравниваем координаты x
            return 1;  //и возвращаем либо положительное,
        if (x == p.x)
            return 0;  //либо нулевое,
        return -1;     //либо отрицательное значение
    }
}
```

3.2.2. Обработчики событий

Прежде всего для работы с коллекциями в начале файла главной формы нужно подключить пространство имен **System.Collections**:

```
using System.Collections;
```

Далее необходимо в классе формы объявить два динамических массива: исходный массив точек **points** и массив-результат **res**. Обоим массивам следует выделить память либо в конструкторе класса, либо в обработчике события **Load**:

```
ArrayList points, res;   //объявление динамических массивов
```

```
private void MainForm_Load(object sender, EventArgs e)
{
    points = new ArrayList(); //выделение памяти
    res = new ArrayList();
}
```

При нажатии кнопки «Добавить» необходимо взять текущие значения компонентов-счетчиков, создать объект класса точки с этими значениями в качестве координат, добавить этот объект в коллекцию и отобразить координаты новой точки в таблице на экране:

```
private void AddButton_Click(object sender, EventArgs e)
{
    //берем значения из компонентов-счетчиков
    float x = (float)XnumericUpDown.Value;
    float y = (float)YnumericUpDown.Value;
    //создаем новую точку
    Point p = new Point(x,y);
    //добавляем ее в коллекцию
    points.Add(p);
    //добавляем в таблицу на форме
    dataGridView1.Rows.Add(p.x,p.y);
}
```

При удалении точки из списка следует сначала проверить, выбрана ли строка таблицы для удаления. Реализуется это стандартными средствами обработки исключений (операторы **try**… **catch**). Если точка выбрана, то надо получить номер соответствующей строки из таблицы, удалить точку с этим номером из коллекции и из таблицы на форме:

```
private void DelButton_Click(object sender, EventArgs e)
{
    try
    {
        //получаем номер выбранной строки
        int num = dataGridView1.SelectedRows[0].Index;
        //удаляем точку с данным номером из коллекции
        points.RemoveAt(num);
        //удаляем выбранную строку из таблицы
       dataGridView1.Rows.Remove(dataGridView1.SelectedRows[0]);
    }
    catch
    {
        MessageBox.Show("Выберите строку!!!");
    }
}
```

При нажатии на кнопку «Решить задачу» должны выполняться следующие действия. Прежде всего необходимо очистить массив-коллекцию результатов **res**, а также обе таблицы на форме, т.к. данные в них будут обновляться. Далее, путем использования методов **Add** и **Remove** классов коллекций

добавляем и удаляем элементы массивов в соответствии с условиями задачи и отображаем эти массивы в таблицах на форме:

```
private void SolveButton_Click(object sender, EventArgs e)
{
    //очистить коллекцию res
    res.Clear();
    //очистить содержимое таблиц на форме
    dataGridView1.Rows.Clear();
    dataGridView2.Rows.Clear();
    //просматриваем исходный массив
    foreach (Point p in points)
        if (p.x < 0)    //при выполнении условия
            res.Add(p); //добавляем точку в массив-результат
    foreach (Point p in res)
    {
        //удаляем найденные точки из исходного массива
        points.Remove(p);
        //отображаем 2-й массив в таблице на форме
        dataGridView2.Rows.Add(p.x, p.y);
    }
    foreach (Point p in points)
        //отображаем 1-й массив в таблице на форме
        dataGridView1.Rows.Add(p.x, p.y);
}
```

Метод-обработчик кнопки «Отсортировать» достаточно простой. Реализация классом **Point** интерфейса **IComparable** позволяет применять к коллекции из объектов этого класса метод **Sort()**, после чего нужно только обновить содержимое отсортированного массива в таблице на форме:

```
private void Sortbutton_Click(object sender, EventArgs e)
{
    //сортируем
    res.Sort();
    //очищаем содержимое таблицы на форме
    dataGridView2.Rows.Clear();
    foreach (Point p in res)
        //обновляем таблицу на форме
        dataGridView2.Rows.Add(p.x, p.y);
}
```

4. Варианты заданий для самостоятельной работы

1. Дан стек, содержащий фамилии студентов и средний балл сессии каждого студента. Создать новый список, в который войдут студенты, средний балл у которых не меньше "4", а в стеке останутся все остальные. Отсортировать новый список по убыванию среднего балла.

2. Дана очередь данных о клиентах пункта проката автомобилей: ФИО, адрес (улица, дом, квартира) и марка машины. Во второй массив записать отсортированные по алфавиту данные только тех людей, кто ездит на "Audi".

3. Дан список английских глаголов: тип (правильный/неправильный), первая форма, вторая форма, третья форма. Вывести только те глаголы, у которых все три формы совпадают. Вывести только правильные (неправильные) глаголы в алфавитном порядке.

4. Рациональное число можно представить записью с двумя полями: числитель и знаменатель. Дан стек из N рациональных чисел. Создать новый список из дробей, обратных исходным (числитель и знаменатель меняются местами), отсортировать его по убыванию дробей. Удалить из этого списка максимальное и минимальное значения.

5. Дан массив данных о работниках фирмы: ФИО и дата поступления на работу (месяц, год). Во второй массив записать только данные тех из них, кто на сегодняшний день проработал уже не менее 5 лет. Отсортировать данные по алфавиту.

6. Дан список данных о работниках фирмы: фамилия, имя, отчество, адрес (улица, дом, квартира) и дата поступления на работу (месяц, год). Отсортировать этот список по году поступления на работу . Определить, есть ли в списке Петровы (Петров, Петрова), если есть, то вывести их адрес (адреса) и записать их в стек.

7. Дан список иногородних студентов из ***n*** человек: ФИО, адрес (город, улица, дом-квартира), приблизительное расстояние до Краснодара. Для них в общежитии выделено ***k*** мест. Вывести очередь студентов, которых необходимо селить в общежитие в первую очередь. Критерий отбора: расстояние до города.

8. Дан массив данных о студентах некоторой группы: фамилия, имя, отчество и дата рождения (день, месяц, год). Вывести на экран фамилию и имя тех студентов, у кого сегодня день рождения (сегодняшнюю дату вводить с клавиатуры). Удалить сведения об этих студентах из исходного списка.

9. Дан массив, содержащий сведения о студентах некоторой группы: ФИО, оценки по пяти экзаменационным дисциплинам. Вывести сначала студентов, получающих повышенную стипендию, затем обычную, и, наконец, без стипендии. Организовать поиск студента по фамилии с выводом информации о нем.

10. Дан стек книг: название, автор, количество страниц. Вывести все книги А.С. Пушкина в алфавитном порядке.

11. Дан массив, в котором хранятся данные о расписании поездов на сегодняшний день: номер поезда, название (т.е. откуда – куда, например, Новороссийск-Москва), время прибытия на станцию и время отправления (часы, минуты). По данному времени определить, какие из поездов стоят сейчас на станции.

12. Для каждого из N студентов группы известны ФИО и оценки (в баллах) по пяти дисциплинам. Сформировать очередь из фамилий и имен тех из них, у которых количество положительных оценок больше, чем отрицательных.

13. Рациональное число можно представить записью с двумя полями: числитель и знаменатель. Дан список рациональных чисел. Записать в другой

список все неправильные дроби, предварительно удалив их из исходного списка и преобразовав в правильные. Неправильной называется дробь, у которой числитель больше знаменателя. Определить количество элементов каждого списка. Отсортировать оба списка по убыванию числителя.

14.Дана хеш-таблица, содержащий сведения о книгах: название, жанр, автор (где название – это ключ хеш-таблицы, остальные поля - значение). Вывести названия книг только классического жанра, отсортировав их по фамилии автора. Найти количество таких книг, насколько их больше (меньше), чем остальных.

15.Рациональное число можно представить записью с двумя полями: числитель и знаменатель. Разработать функцию *сократить(m)* приведения рационального числа *m* к несократимому виду. Дан стек рациональных чисел. Сократить эти числа и записать их в список, отсортировав по убыванию.

16.Дана очередь данных о работниках фирмы: ФИО и адрес (улица, дом, квартира). Во второй массив записать только тех из них, которые живут на улице Красной. Вывести их на экран в алфавитном порядке.

17.Дан список, содержащий числовые данные. Отсортировать его по возрастанию и сформировать два новых списка таким образом, чтобы половина элементов исходного списка попала в первый новый список (1, 3, 5, …), а вторая половина – во второй новый (2, 4, 6, …).

18.Дана очередь, содержащая перечень товаров различных фирм. Из элементов этого списка создать новый список, который будет содержать товары, изготовленные фирмой SONY. Отсортировать их по алфавиту, определить количество таких товаров и их долю от общего количества товаров.

19.Дан список, содержащий информацию о клиентах пункта проката автомобилей: ФИО и марка машины. Отсортировать его марке машины. Записать в стек данные только тех людей, кто ездит на "Audi", предварительно удалив их из исходного списка. Организовать запрос на наличие в исходном списке требуемого человека.

20.Даны два списка, содержащие перечни товаров, производимых концернами SHARP и LG. Создать список товаров, выпускаемых как одной, так и другой фирмой. Отсортировать его по алфавиту. Организовать поиск в этом списке заданного товара.

21.Дан список четырехзначных номеров лотерейных билетов. Отсортировать его по возрастанию сумм цифр числа. Сформировать очередь из чисел, которые состоят из одинаковых цифр (1111, 2222, 3333…).

22.В магазине формируется список лиц, записавшихся на покупку товара повышенного спроса. Каждая запись этого списка содержит: порядковый номер, ФИО, домашний адрес покупателя и дату постановки на учет. Удалить из списка все повторные записи, проверяя ФИО и домашний адрес. Данные организовать в виде хеш-таблицы, где порядковый номер – это ключ, а все остальные данные – значение.

23.Информация о сотрудниках двух отделов предприятия записана в стеке и содержит: ФИО, название отдела, должность, дату начала работы. Вывести списки сотрудников по отделам в порядке убывания стажа.

24.Для книг, хранящихся в библиотеке, задаются: регистрационный номер книги, автор, название, год издания, издательство, количество страниц. Данные организованы в виде хеш-таблицы, где регистрационный номер – это ключ, а все остальные данные – значение. Создать список типа ArrayList книг с фамилиями авторов в алфавитном порядке, изданных после заданного года.

25.Различные цехи завода выпускают продукцию нескольких наименований. Сведения о выпущенной продукции включают: наименование, количество, номер цеха. Для заданного цеха необходимо вывести количество выпущенных изделий по каждому наименованию в порядке убывания количества. Исходные данные задаются в виде очереди, результат – в виде динамического массива ArrayList.

26.Стек данных о группе студентов содержит следующую информацию: ФИО, рост и вес. Сформировать и вывести список ФИО студентов, рост и вес которых являются в списке уникальными. Отсортировать список по алфавиту.

27.Информация об участниках спортивных соревнований двух команд содержит: наименование страны, название команды, ФИО игрока, игровой номер, возраст. Вывести и отсортировать по игровому номеру информацию о самой молодой команде.

28.Дана очередь данных о служащих предприятия: фамилия, отдел, зарплата. Сформировать список данных по алфавиту о работниках бухгалтерии и определить их суммарную зарплату.

29.Ведомость абитуриентов, сдавших вступительные экзамены в институт, содержит: ФИО, адрес, оценки. Определить количество абитуриентов, проживающих в Краснодаре, и сдавших экзамены со средним баллом не ниже 4,5, записать их в отдельный список, отсортировать по фамилии и удалить из исходного списка.

30.На междугородней АТС информация о разговорах содержит дату разговора, код и название города, время разговора, тариф, номер телефона в этом городе и номер телефона абонента. Сформировать и вывести хеш-таблицу, где ключом будет код и название города, а значением – общее время разговоров с ним и сумма.

Лабораторная работа №6. Наследование классов

1 Цель работы

Цель работы – ознакомиться с основными механизмами наследования классов и интерфейсов изучить понятия виртуальных функций и абстрактных классов.

2 Порядок выполнения работы

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя по вариантам;
- разработать и отладить программу;
- составить и защитить отчет по лабораторной работе у преподавателя.

3 Содержание отчета

- титульный лист;
- краткое теоретическое описание;
- задание на лабораторную работу, включающее математическую формулировку задачи;
- результаты выполнения работы, включающие схему алгоритма, тексты программ, результаты вычислений;

4 Краткая теория

4.1 Общие сведения

Объекты разных классов и сами классы могут находиться в отношении наследования, при котором формируется иерархия объектов, соответствующая заранее предусмотренной иерархии классов.

Иерархия классов позволяет определять новые классы на основе уже имеющихся. Имеющиеся классы обычно называют *базовыми* (иногда порождающими), а новые классы, формируемые на основе базовых, - *производными* (порожденными), иногда классами-потомками или наследниками. Производные классы «получают наследство» – данные и методы своих базовых классов – и, кроме того, могут пополняться собственными компонентами (данными и собственными методами). Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то они ищутся автоматически и незаметно для программиста в базовом классе (см. рис. 4.1). Общая форма объявления класса, который наследует базовый класс, имеет такой вид:

```
class имя__производного_класса : имя_базового_класса {
// тело класса
}
```

Для создаваемого производного класса можно указать только один базовый класс. В C# (в отличие от C++) не поддерживается наследование нескольких базовых классов в одном производном классе.

Основное достоинство наследования состоит в том, что, создав базовый класс, который определяет общие атрибуты для множества объектов, его можно использовать для создания любого числа более специализированных производных классов. В каждом производном классе можно затем точно "настроить" собственную классификацию.

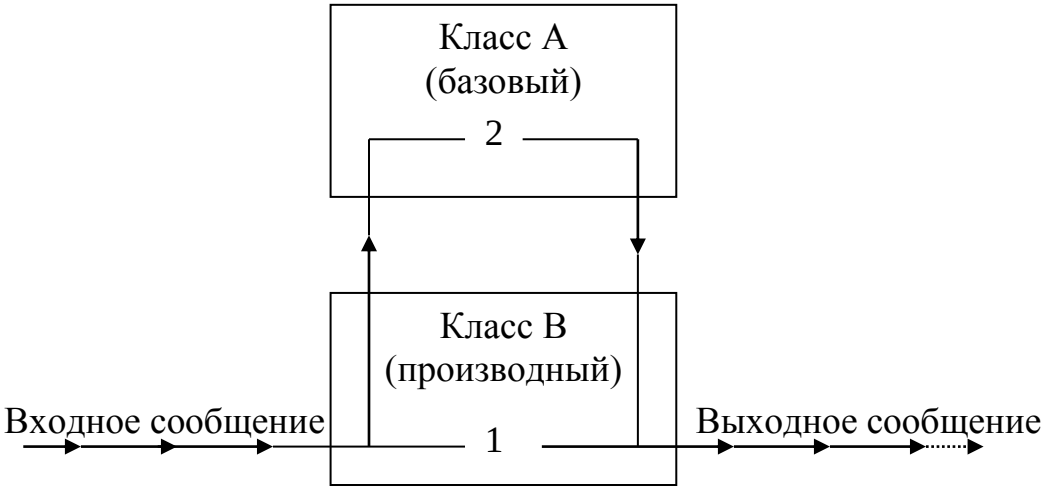


Рисунок 4.1 – Схема обработки сообщений в иерархии объектов:
1 – обработка сообщения методами производного класса;
2 – обработка сообщения методами базового класса.

При описании нового класса, производного от какого-то одного или нескольких базовых классов, можно добавлять новые функции-элементы и данные-элементы, сохраняя при этом все компоненты родителей, а можно родительские компоненты переопределить или перегрузить. В производном классе доступны все открытые и защищенные элементы базового класса (прямого или косвенного предшественника). Закрытые компоненты базового класса в производном классе недоступны.

C#-программисты обычно предоставляют доступ к закрытым членам класса посредством открытых методов или путем превращения их в свойства. Во многих случаях используют защищённые члены классов. Защищенным является член, который открыт для своей иерархии классов, но закрыт вне этой иерархии. Защищенный член создается с помощью модификатора доступа protected. При объявлении protected-члена он, по сути, является закрытым, но с одним исключением. Это исключение вступает в силу, когда защищенный член наследуется. В этом случае защищенный член базового класса становится защищенным членом производного класса, а следовательно, и доступным для производного класса. Таким образом, используя модификатор доступа protected, можно создавать закрытые (для "внешнего мира") члены класса, но вместе с тем они будут наследоваться с возможностью до-

ступа со стороны производных классов.

Наследование класса не отменяет ограничения, связанные с закрытым доступом. Таким образом, несмотря на то, что производный класс включает все члены базового класса, он не может получить доступ к тем из них, которые объявлены закрытыми.

C#-программисты обычно предоставляют доступ к закрытым членам класса посредством открытых методов или путем превращения их в свойства.

Во многих случаях используют защищённые члены классов. Защищенным является член, который открыт для своей иерархии классов, но закрыт вне этой иерархии.

Защищенный член создается с помощью модификатора доступа ***protected***. При объявлении ***protected***-члена он, по сути, является закрытым, но с одним исключением. Это исключение вступает в силу, когда защищенный член наследуется. В этом случае защищенный член базового класса становится защищенным членом производного класса, а следовательно, и доступным для производного класса. Таким образом, используя модификатор доступа ***protected***, можно создавать закрытые (для "внешнего мира") члены класса, но вместе с тем они будут наследоваться с возможностью доступа со стороны производных классов.

Подобно модификаторам public и private модификатор protected остается со своим членом независимо от реализуемого количества уровней наследования. Следовательно, при использовании производного класса в качестве базового для создания другого производного класса любой защищенный член исходного базового класса, который наследуется первым производным классом, также наследуется в статусе защищенного и вторым производным классом.

4.2 Конструкторы

В иерархии классов как базовые, так и производные классы могут иметь собственные конструкторы. При этом возникает важный вопрос: какой конструктор отвечает за создание объекта производного класса? Конструктор базового класса создает часть объекта, соответствующую базовому классу, а конструктор производного класса — часть объекта, соответствующую производному классу.

Если конструктор определяется только в производном классе, процесс создания объекта несложен: просто создается объект производного класса. Часть объекта, соответствующая базовому классу, создается автоматически с помощью конструктора поумолчанию.

Если конструкторы определены и в базовом, и в производном классе, процесс создания объектов несколько усложняется, поскольку должны выполниться конструкторы обоих классов. В этом случае необходимо использовать еще одно ключевое сло-

во***C#base***, которое имеет два назначения: вызвать конструктор базового класса и получить доступ к члену базового класса, который скрыт "за" членом производного класса.

Производный класс может вызывать конструктор, определенный в его базовом классе, используя расширенную форму объявления конструктора производного класса и ключевое слово ***base***. Формат расширенного объявления таков:

конструктор_производиого_класса(список_параметров) :
base (список_аргументов) {
// тело конструктора
}

Здесь с помощью элемента ***список_аргументов***задаются аргументы, необходимые конструктору в базовом классе.

При задании производным классом ***base***-"метода" вызывается конструктор непосредственного базового класса. Таким образом, ключевое слово ***base*** всегда отсылает к базовому классу, стоящему в иерархии классов непосредственно над вызывающим классом. Это справедливо и для многоуровневой иерархии. Чтобы передать аргументы конструктору базового класса, достаточно указать их в качестве аргументов "***метода***" ***base()***. При отсутствии ключевого слова ***base*** автоматически вызывается ***конструктор базового класса, действующий по умолчанию.***

Производный класс может определить член, имя которого совпадает с именем члена базового класса. В этом случае член базового класса становится скрытым в производном классе. Существует вторая форма использования ключевого слова ***base***, которая действует подобно ссылке ***this***, за исключением того, что ссылка ***base*** всегда указывает на базовый класс производного класса, в котором она используется. В этом случае формат ее записи такой: ***base.член.*** Здесь в качестве элемента ***член*** можно указывать либо метод, либо переменную экземпляра. Эта форма ссылки ***base*** наиболее применима в тех случаях, когда имя члена в производном классе скрывает член с таким же именем в базовом классе.

4.3 Виртуальные методы и их переопределение

Виртуальным называется метод, объявляемый с помощью ключевого слова ***virtual*** в базовом классе и переопределяемый в одном или нескольких производных классах. Таким образом, каждый производный класс может иметь собственную версию виртуального метода. Виртуальные методы представляют интерес с такой позиции: что произойдет, если виртуальный метод будет вызван посредством ссылки на базовый класс. Какую именно версию метода нужно вызвать, C# определяет по типу объекта, на который указывает эта ссылка, причем решение принимается динамически, во время выполнения программы. Следовательно, если имеются ссылки на различные объекты, будут выполняться различные вер-

сии виртуального метода. Другими словами, именно тип объекта, на который указывает ссылка (а не тип ссылки) определяет, какая версия виртуального метода будет выполнена. Таким образом, если базовый класс содержит виртуальный метод и из этого класса выведены производные классы, то при наличии ссылки на различные типы объектов (посредством ссылки на базовый класс) будут выполняться различные версии этого виртуального метода.

Чтобы объявить метод в базовом классе виртуальным, его объявление необходимо предварить ключевым словом ***virtual*** . При переопределении виртуального метода в производном классе используется модификатор ***override***. Итак, процесс переопределения виртуального метода в производном классе иногда называется ***замещением метода*** (***methodoverriding***). При переопределении метода сигнатуры типа у виртуального и метода-заменителя должны совпадать. Кроме того, виртуальный метод нельзя определять как статический (с использованием слова ***static***) или абстрактный (с использованием слова ***abstract***).

Переопределение виртуального метода формирует базу для одной из самых мощных концепций C#: динамической диспетчеризации методов. Динамическая диспетчеризация методов — это механизм вызова переопределенного метода во время выполнения программы, а не в период компиляции. Именно благодаря механизму диспетчеризации методов в C# реализуется динамический полиморфизм.

Производный класс не предоставляет собственную версию виртуального метода, используется версия, определенная в базовом классе. Если производный класс не переопределяет виртуальный метод в случае многоуровневой иерархии, то будет выполнен первый переопределенный метод, который обнаружится при просмотре иерархической лестницы в направлении снизу вверх.

Свойства также можно модифицировать с помощью ключевого слова ***virtual***, а затем переопределять с помощью ключевого слова ***override***.

4.4 Использование абстрактных классов

Иногда полезно создать базовый класс, определяющий только своего рода "пустой бланк", который унаследуют все производные классы, причем каждый из них заполнит этот "бланк" собственной информацией. Такой класс определяет "суть" методов, которые производные классы должны реализовать, но сам при этом не обеспечивает реализации одного или нескольких методов. Подобная ситуация может возникнуть, когда базовый класс попросту не в состоянии реализовать метод.

Абстрактный метод создается с помощью модификатора типа ***abstract***. Абстрактный метод не содержит тела и, следователь-

но, не реализуется базовым классом. Поэтому производный класс должен его переопределить, поскольку он не может использовать версию, предложенную в базовом классе. Нетрудно догадаться, что абстрактный метод автоматически является виртуальным, поэтому и нет необходимости в использовании модификатора ***virtual***. Более того, совместное использование модификаторов ***virtual*** и ***abstract*** считается ошибкой.

Для объявления абстрактного метода используйте следующий формат записи.

abstract ТИП ИМЯ(список_параметров) ;

Тело абстрактного метода отсутствует. Модификатор ***abstract*** можно использовать только применительно к обычным, а не к ***static-методам***. Свойства также могут быть абстрактными.

Класс, содержащий один или несколько абстрактных методов, также должен быть объявлен как абстрактный с помощью спецификатора ***abstract***, который ставится перед объявлением ***class***. Поскольку абстрактный класс нереализуем в полном объеме, невозможно создать его экземпляры, или объекты. Таким образом, попытка создать объект абстрактного класса с помощью оператора ***new*** приведет к возникновению ошибки времени компиляции.

Если производный класс выводится из абстрактного, он может реализовать все абстрактные методы базового класса. В противном случае такой производный класс также должен быть определен как абстрактный. Таким образом, атрибут ***abstract*** наследуется до тех пор, пока реализация класса не будет полностью достигнута.

5 Примеры программ

Создать класс двумерной фигуры. Используя наследование, создать классы треугольника и прямоугольника с методами вычисления площади, и определения их типов. Поскольку для не определенной заранее двумерной фигуры понятие площади не имеет смысла, в следующей версии предыдущей программы метод ***area()*** в классе ***TwoDShape*** объявляется как абстрактный, как, впрочем, и сам класс ***TwoDShape***. Безусловно, это означает, что все классы, выведенные из ***TwoDShape***, должны переопределить метод ***area().***

```
using System;
abstract class TwoDShape {
double pri_width; // Закрытый член,
double pri_height; // Закрытый член,
string pri_name; // Закрытый  член.
public TwoDShape() {// Конструктор по умолчанию,
width = height = 0.0;
name = "null";
}
```

```
        // Конструктор с параметрами.
        public TwoDShape(double w, double h, string n) {
        width = w;height = h;name = n;
        }
        // Создаем объект, у которого ширина равна высоте,
        public TwoDShape(double x, string n) {
        width = height = x;name = n;
        }
        // Создаем объект из объекта,
        public TwoDShape(TwoDShape ob) {
        width = ob.width;height = ob.height; name = ob.name;
        }
        // Свойства width, height и name,
        public double width {
        get { return pri_width; }set { pri_width = value; }
        }
        public double height {
        get { return pri_height; }set { pri_height = value; }
        }
        public string name {
        get { return pri__name; }set { pri_name = value; }
        }
        public void showDim() {
        Console.WriteLine("Ширина и высота равны " +
        width + " и " + height);
        }
        // Теперь метод area() абстрактный,
        public abstract double area();
        }
// Класс треугольников, производный от класса TwoDShape.
        class Triangle : TwoDShape {
        string style; // Закрытый член.
        // Конструктор по умолчанию,
        public Triangle() {
        style = "null";
        }
        // Конструктор с параметрами.
        public Triangle(string s, double w, double h) :
        base(w, h, "triangle") {
        style = s;
        }
        // Создаем равнобедренный треугольник.
        public Triangle(double x) : base(x, "треугольник") {
        style = "равнобедренный";
        }
```

```
        // Создаем объект из объекта. ,
        public Triangle(Triangle ob) : base(ob) {
        style = ob.style;
        }
        // Переопределяем метод area() для класса Triangle,
        public override double area() {return width * height / 2 ;}
        // Отображаем тип треугольника,
        public void showStyle() {
        Console.WriteLine("Треугольник " + style);
        }
        }
//Класс прямоугольников, производный от класса TwoDShape.
        class Rectangle : TwoDShape {
        // Конструктор с параметрами.
        public Rectangle(double w, double h) :
        base(w, h, "прямоугольник"){ }
        // Создаем квадрат.
        public Rectangle (double x) :base(x, "прямоугольник") { }
        // Создаем объект из объекта.
        public Rectangle(Rectangle ob) : base(ob) { }
        // Метод возвращает значение true, если
        // прямоугольник является квадратом,
        public bool isSquare() {
        if (width == height) return true;
        return false;
        }
        // Переопределяем метод area() для класса Rectangle,
        public override double area() {
        return width * height;
        }}
        class AbsShape {
        public static void Main() {
        TwoDShape [] shapes == new TwoDShape[4];
        shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
        shapes[1] = new Rectangle(10);
        shapes[2] = new Rectangle(10, 4);
        shapes[3] = new Triangle (7.0);
        for(int i=0; i < shapes.Length; i++) {
        Console.WriteLine("Объектом является " +shapes[i].name);
        Console.WriteLine("Площадь равна " +shapes[i].area());
        Console.WriteLine();
        }}}
```

6 Контрольные вопросы

1. В чем заключается механизм простого наследования? Как описать производный класс?

2. Как соблюдается принцип инкапсуляции (сокрытия данных) при простом наследовании?

3. Можно ли в классе-наследнике переопределять компонентные функции базового класса? Как это сделать?

4. Что такое полиморфизм? В каких случаях он применяется?

5. Нужно ли объявлять функцию виртуальной в производном классе, если она объявлена таковой в базовом классе?

6. Дайте определение чистой виртуальной функции.

7. В каких случаях целесообразно создание абстрактных классов?

8. Можно ли создать объект абстрактного класса? А указатель на него?

7 Варианты заданий для самостоятельного решения

Заданы названия базовых и производных классов. Необходимо разработать поля и методы, наследуемые из базового класса, и собственные компоненты производных классов. Базовый класс может быть абстрактным.

1-2. Первый базовый класс – средство передвижения. Поля в нем: вес, мощность мотора, скорость. Во втором базовом классе описать страны-производители. Производные классы – автомобиль (1 вариант), самолет (2 вариант); производные второго поколения – спортивный автомобиль, грузовой автомобиль (1 вариант), военный самолет, транспортный самолет (2 вариант).

3-4. Первый базовый класс – млекопитающие; поля – способ питания, вес, среда обитания. Во втором базовом классе описываются географические регионы. Производные классы – хищники (3 вариант) и травоядные (4 вариант).

5-6. Первый базовый класс – личность; поля – фамилия, пол, адрес. Во втором базовом классе задается структура университета – факультеты, кафедры, службы. Производные классы – студенты (5 вариант) и сотрудники университета (6 вариант).

7-8. Первый базовый класс – библиотека. Второй базовый класс – система УДК или ключевых слов. Производные классы – техническая (7 вариант) и художественная (8 вариант) литература.

9-10. Первый базовый класс – документ предприятия. Во втором базовом классе описываются корреспонденты предприятия. Производные классы – приказы (9 вариант), письма (10 вариант).

11-12. Первый базовый класс – точка. Производные классы – геометрические фигуры (11 вариант) и текст, который вписывается в фигуру (12 вариант).

13-14. Первый базовый класс – магазин; поля – номер, телефон, адрес, владелец, номер лицензии. Во втором базовом классе описываются поставщики товаров. Производные классы – отделы (13 вариант), продавцы (14 вариант).

15. Первый базовый класс – товар; поля – наименование, количество,

производитель, цена за единицу изделия, общая цена. Во втором базовом классе описываются качественные характеристике товара. Производный класс – товар на складе. Создать массив указателей на класс. Вывести все данные о товарах, отсортированные по возрастанию цены за единицу изделия.

16. Создать абстрактный базовый класс с виртуальной функцией – площадь. Создать производные классы: прямоугольник, круг, прямоугольный треугольник, трапеция со своими функциями площади. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов. Площадь трапеции: $S = (a+b)\frac{h}{2}$.

17. Создать абстрактный класс с виртуальной функцией – норма. Создать производные классы: комплексные числа, вектор из 10 элементов, матрица 2x2. Определить функцию нормы: для комплексных чисел – модуль в квадрате, для вектора – корень квадратный из суммы элементов по модулю, для матрицы – максимальное значение по модулю.

18. Создать абстрактный класс «кривые» вычисления координаты *y* для некоторого *x*. Создать производные классы: прямая, эллипс, гипербола со своими функциями вычисления *y* в зависимости от входного параметра *x*. Уравнение прямой: $y = ax + b$, эллипса: $\frac{x^2}{a^2} + \frac{y^2}{a^2} = 1$, гиперболы: $\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$

19. Создать абстрактный базовый класс с виртуальной функцией – сумма прогрессии. Создать производные классы: арифметическая прогрессия и геометрическая прогрессия. Каждый класс имеет два поля типа **`double`**. Первое – первый член прогрессии, второе – постоянная разность (для арифметической) и постоянное отношение (для геометрической). Определить функцию вычисления суммы, где параметром является количество элементов прогрессии.

Арифметическая прогрессия $a_j = a_0 + jd,\ j = 0,1,2,...$

Сумма арифметической прогрессии: $s_n = (n+1)\frac{a_0 + a_n}{2}$

Геометрическая прогрессия: $a_j = a_0 r^j,\ j = 0,1,2,...$

Сумма геометрической прогрессии: $s_n = \frac{a_0 - a_n r}{1 - r}$

20. Создать базовый класс «список» с виртуальными функциями вставки и извлечения. Реализовать на базе списка производные классы стека и очереди.

21. Создать базовый класс – фигура, и производные: круг, прямоугольник, трапеция. Определить виртуальные функции: площадь, периметр и вывод на печать.

22. Создать базовый класс – работник, и производные классы – служащий с почасовой оплатой, служащий в штате и служащий с процентной ставкой. Определить функцию начисления зарплаты.

23. Описать абстрактный базовый класс с виртуальной функцией – площадь поверхности. Создать производные классы: параллелепипед, тетраэдр, шар со своими функциями площади поверхности. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов. Площадь поверхности параллелепипеда: $S = 6xy$. Площадь поверхности шара: $S = 4\pi r^2$. Площадь поверхности тетраэдра: $S = a^2\sqrt{3}$

24. Создать класс «человек», производные от которого мужчины и женщины. Определить виртуальную функцию реакции человека на вновь увиденного другого человека.

25. Создать абстрактный базовый класс с виртуальной функцией – объем. Создать производные классы: параллелепипед, пирамида, тетраэдр, шар со своими функциями объема. Для проверки определить массив указателей на абстрактный класс, которым присваиваются значения указателей на объекты производных классов. Объем параллелепипеда: $V = xyz$ (x, y, z – стороны пирамиды). Объем пирамиды: $V = xyh$ (x, y – стороны, h – высота). Объем тетраэдра: $V = \frac{a^3\sqrt{2}}{12}$. Объем шара: $V = \frac{4}{3}\pi r^3$

26. Создать абстрактный класс – млекопитающие. Определить производные классы – животные и люди. У животных определить производные классы собак и коров. Определить виртуальные функции описания человека, собаки и коровы.

27. Создать базовый класс – предок, у которого есть фамилия. Определить виртуальную функцию печати. Создать производный класс – ребенок, у которого функция печати дополнительно выводит имя. Создать производный класс от последнего класса – внук, у которого есть отчество. Написать свою функцию печати.

28. Создать абстрактный базовый класс с виртуальной функцией – корни уравнения. Создать производные классы: класс линейных уравнений и класс квадратных уравнений. Определить функцию вычисления корней уравнений.

Лабораторная работа №7. Обработка исключений

1 Цель работы

Цель работы – ознакомиться с понятием исключительных ситуаций и способами их обработки, изучить методы создания классов исключений и их объектов, способов формирования исключений в программе.

2 Порядок выполнения работы

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя по вариантам;
- разработать и отладить программу;
- составить и защитить отчет по лабораторной работе у преподавателя.

3 Содержание отчета

- титульный лист;
- краткое теоретическое описание;
- задание на лабораторную работу, включающее математическую формулировку задачи;
- результаты выполнения работы, включающие схему алгоритма, тексты программ, результаты вычислений;

4 Краткая теория

Исключительнаяситуация (или ***исключение***) — это ошибка, которая возникает во время выполнения программы. Используя C#-***подсистему*** обработки исключительных ситуаций, с такими ошибками можно справляться. Эта подсистема в C# включает в себя усовершенствованные методы, используемые в языках C++ и Java. Обработка исключений в C# отличается ясностью и полнотой реализации. Преимущество подсистемы обработки исключений состоит в автоматизации создания большей части кода, который ранее необходимо было вводить в программы "вручную".

В C# исключения представляются классами. Все классы исключений должны быть выведены из встроенного класса исключений Exception, который является частью пространства имен System. Таким образом, все исключения — подклассы класса Exception.

Из класса Exception выведены классы SystemException и ApplicationException. Они поддерживают две общие категории исключений, определенные в C#: те, которые генерируются C#-системой динамического управления, или общеязыковым средством управления (CommonLanguageRuntime — CLR), и те, которые генерируются прикладными программами. Но ни класс SystemException, ни класс ApplicationException не привносят ничего нового в дополнение к членам класса Exception. Они просто определяют вершины двух различных иерархий классов исключений. C# определяет встроенные исключения, которые выводятся из класса SystemException. Например, при

попытке выполнить деление на нуль генерируется исключение класса DivideByZeroException. Вы сможете создавать собственные классы исключений, выводя их из класса ApplicationException.

Управление C#-механизмом обработки исключений основывается на четырех ключевых словах: try, catch, throw и finally . Они образуют взаимосвязанную подсистему, в которой использование одного из них предполагает использование другого.

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в try-блок. Если исключение возникает в этом блоке, онодает знать о себе выбросом определенного рода информации. Это выброшенное исключение может быть перехвачено программным путем с помощью catch-блока и обработано соответствующим образом. Системные исключения автоматически генерируются C#-системой динамического управления. Чтобы сгенерировать исключение вручную, используется ключевое слово throw. Любой код, который должен быть обязательно выполнен при выходе из try-блока, помещается в блок finally.

Ядром обработки исключений являются блоки try и catch. Эти ключевые слова работают "в одной связке"; нельзя использовать слово try без catch или catch без try. Формат записи try/catch-блоков обработки исключений:

```
try {
// Блок кода, подлежащий проверке на наличие ошибок.
}
catch {ExcepType1 exOb) {
// Обработчик для исключения типа ExcepType1.
}
catch (ExcepType2  exOb) {
// Обработчик для исключения типа ExcepType2.
}…
```

Здесь ExcepType — это тип сгенерированного исключения.

После "выброса" исключение перехватывается соответствующей инструкцией catch, которая его обрабатывает. Из формата записи try/catch-блоков видно, с try-блоком может быть связана не одна, а несколько catch-инструкций. Какая именно из них будет выполнена, определит тип исключения. Другими словами, будет выполнена та catch-инструкция, тип исключения которой совпадает с типом сгенерированного исключения (а все остальные будут проигнорированы). После перехвата исключения параметр exOb примет его значение.

Одно из основных достоинств обработки исключений состоит в том, что она позволяет программе отреагировать на ошибку и продолжить выполнение. После обработки исключение удаляется из системы.

С try-блоком можно связать не одно, а несколько catch-инструкций. Все catch-инструкции должны перехватывать исключения различного типа.

Каждая catch-инструкция реагирует только на собственный тип исключения. В общем случае catch-выражения проверяются в том порядке, в котором они встречаются в программе. Выполняется только инструкция, тип исключения которой совпадает со сгенерированным исключением. Все остальные catch-блоки игнорируются.

Иногда требуется перехватывать все исключения, независимо от их типа. Для этого используйте catch-инструкцию без параметров.

Один try-блок можно вложить в другой. Исключение, сгенерированное во внутреннем try-блоке и не перехваченное catch-инструкцией, которая связана с этим try-блоком, передается во внешний try-блок.

Многие программисты используют внешний ***try-блок*** для перехвата самых серьезных ошибок, позволяя внутренним ***try-блокам*** обрабатывать менее опасные.

Помимо генерации исключений генерируемых средствами C#, можно сгенерировать исключение программно, используя инструкцию ***throw***. Формат ее записан таков:

throw exceptOb;

Элемент exceptOb — это объект класса исключений, производного от класса Exception.

Иногда возникает потребность определить программный блок, который должен выполняться по выходу из try/catch-блока. Например, исключение может вызвать ошибку, которая завершает текущий метод и, следовательно, является причиной преждевременного возврата. Однако такой метод может оставить открытым файл или соединение с сетью, которые необходимо закрыть. Подобные обстоятельства — обычное явление в программировании, и C# предоставляет удобный путь выхода из них: блок finally.

Чтобы определить блок кода, подлежащий выполнению по выходу из try/catch-блока, включите в конец try/catch-последовательности блок finally. Общая форма записи последовательности try/catch-блоков, содержащей блок finally, выглядит следующим образом.

```
try {
// Блок кода, предназначенный для обработки ошибок.
}
catch (ExcepTypel exOb) {
```

```
// Обработчик для исключения типа ExcepTypel.
}
catch (ExcepType2 exOb) {
// Обработчик для исключения типа ExcepType2.
finally {
// Код завершения обработки исключений.
}
```

Блок finally будет выполнен после выхода из try/catch-блока, независимо от условий его выполнения. Другими словами, при нормальном завершении try-блока или в условиях возникновения исключения содержимое finally-блока будет безусловно отработано. Блок finally выполнится и в том случае, если любой код внутри try-блока или любая из его catch-инструкций определены внутри метода.

5 Пример программы

1 Известно, что попытка индексировать массив за пределами его границ вызывает ошибку нарушения диапазона. В этом случае ***C#-система*** динамического управления генерирует исключение типа ***IndexOutOfRangeException***, которое представляет собой стандартное исключение, определенное языком ***C#***.

```
using System; // Демонстрация обработки исключений,
class ExcDemo1 {
public static void Main() {
int[] nums = new int [4];
try {
Console.WriteLine(
"Перед генерированием исключения.");
// Генерируем исключение, связанное с попаданием
// индексавнедиапазона,
for(int i=0; i < 10; i++) {
nums[i] = i;
Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
}
Console.WriteLine("Этот текст не отображается.");
}
catch (IndexOutOfRangeException) {
// Перехватываем исключение.
Console.WriteLine("Индекс вне диапазона!");
```

```
}
Console.WriteLine("После catch-инструкции.");
}}
```

При выполнении этой программы получаем такие результаты:

```
Перед генерированием исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Индекс вне диапазона!
```

2 Например, следующая программа перехватывает как ошибку нарушения границ массива, так и ошибку деления на нуль.

```
// Использование нескольких catch-инструкций.
usingSystem;
class ExcDemo3 {
public static void Main() {
// Здесь массив numer длинне массива denom.
int [] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
int[] denom = { 2, 0, 4, 4, 0 , 8 };
for(int i=0; i < numer.Length;i++){
try {
Console.WriteLine(numer[i] + " / " +denom[i] + " равно " +
numer[i]/denom[i]);
}
catch (DivideByZeroException) {// Перехватываем исключение
Console.WriteLine("Делитьнанульнельзя!");
}
catch (IndexOutOfRangeException) {
// Перехватываем исключение.
Console.WriteLine("Нет соответствующего элемента.")
}}}}
```

Эта программа генерирует следующие результаты:

```
4/2
равно 2
Делить на нуль нельзя!
16/4
равно 4
32/4
равно 8
Делить на нуль нельзя!
128/8 равно 16
Нет соответствующего элемента.
Нет соответствующего элемента.
```

3 Использование catch-инструкции для “глобального перехвата".

```
using System;
class ExcDemo4 {
public static void Main() {
// Здесь массив numer длиннее массива denoiti.
int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
int[] denom = { 2 , 0, 4, 4, 0, 8 } ;
for(int i=0; i < numer.Length; i++){
try {
Console.WriteLine(numer[i] + " / " +
denom[i] + " равно " +
numer[i]/denom[i]);
}
catch {
Console.WriteLine(
"Произошло некоторое исключение.");
}}}}
```

Вот как выглядят результаты выполнения этой программы:

```
4/2
равно 2
Произошло некоторое исключение.
16/4 равно 4
32/4 равно 8
Произошло некоторое исключение.
128/8 равно 16
Произошло некоторое исключение.
Произошло некоторое исключение.
```

6 Контрольные вопросы

1. Что такое исключительная ситуация? Приведите примеры программных исключений.
2. Как выделить в программе контролируемый блок?
3. С помощью какого ключевого слова осуществляется генерация исключительных ситуаций? Укажите существующие формы этого оператора.
4. Перечислите известные Вам формы описания обработчиков исключений.
5. Каким образом осуществляется обработка исключений во вложенных контролируемых блоках?

7 Варианты заданий для самостоятельного решения

Для каждого варианта необходимо создать три массива *a*, *b* и *c* размерами соответственно *n1*, *n2* и *n3* (*n1≠n2≠n3*). В массив *a* занести значения функции *f(x)* согласно варианту (при возникновении исключения заносить нули). Массив *b* заполнить случайными числами (среди них должны быть и

отрицательные числа и нули). Массив *c* формируется согласно варианту. Предусмотреть и обработать возникающие при этом исключительные ситуации (деление на ноль, корень из отрицательного числа, арифметическое переполнение, выход за пределы диапазона индексов массива и т.п.). Варианты заданий приведены в таблице 7.1.

Таблица 7.1 – Варианты заданий

Вариант	Функция *f(x)*	Способ формирования массива *c*
1.	$\ln(x-1), x \in [0;10], \Delta x = 0{,}5$	$c_i = a_i + 1/b_i$
2.	$\ln(x^2-1), x \in [0;4], \Delta x = 0{,}2$	$c_i = a_i - 1/b_i$
3.	$\ln(x-1)^3, x \in [0;6], \Delta x = 0{,}3$	$c_i = a_i / b_i$
4.	$\ln(x+1)^3, x \in [-3;3], \Delta x = 0{,}3$	$c_i = a_{i-1} + 1/b_{i+1}$
5.	$\ln(1-x), x \in [-1;3], \Delta x = 0{,}2$	$c_i = a_{i+!} - 1/b_{i-1}$
6.	$\ln\frac{1}{x+1}, x \in [-2;4], \Delta x = 0{,}3$	$c_i = a_i / b_{i+1}$
7.	$\ln\frac{1}{x-1}, x \in [-1;9], \Delta x = 0{,}5$	$c_i = \sqrt{a_i \cdot b_i}$
8.	$\lg\frac{x-1}{x+1}, x \in [-3;7], \Delta x = 0{,}5$	$c_i = \sqrt{a_i + b_i}$
9.	$\lg((x-1)\cdot(x+1)), x \in [-3;3], \Delta x = 0{,}3$	$c_i = \sqrt{\frac{a_i}{b_i}}$
10.	$\lg\frac{1}{x-1}, x \in [-2;4], \Delta x = 0{,}3$	$c_i = \sqrt{a_i - b_{i-1}}$
11.	$\ln\frac{x+1}{x-1}, x \in [-2;8], \Delta x = 0{,}5$	$c_i = \sqrt{a_{i+1} \cdot b_{i+1}}$
12.	$\lg\frac{1}{x+1}, x \in [-2;2], \Delta x = 0{,}2$	$c_i = \sqrt{a_{i-1} + b_{i+1}}$
13.	$\ln((x-1)\cdot(x+1)), x \in [-3;5], \Delta x = 0{,}4$	$c_i = \sqrt{\frac{a_{i+1}}{b_{i-1}}}$
14.	$\lg(1-x)^3, x \in [-3;3], \Delta x = 0{,}3$	$c_i = \sqrt{a_{i+1} - b_i}$
15.	$\sqrt{x^2-1}, x \in [-2;2], \Delta x = 0{,}2$	$c_i = \ln(a_i + b_i)$
16.	$\sqrt{x^3+1}, x \in [-2;2], \Delta x = 0{,}2$	$c_i = \ln(\frac{a_i}{b_i})$
17.	$\sqrt{1-x^2}, x \in [-2;2], \Delta x = 0{,}2$	$c_i = \lg(a_i - b_i)$
18.	$\sqrt{1-x^3}, x \in [-2;2], \Delta x = 0{,}2$	$c_i = \ln(a_{i-1} + b_{i+1})$
19.	$\sqrt{\frac{1}{x^2}}, x \in [-2;2], \Delta x = 0{,}2$	$c_i = \ln(\frac{a_{i+1}}{b_{i-1}})$
20.	$\frac{1}{x-2}, x \in [-1;5], \Delta x = 0{,}3$	$c_i = \lg(a_i - \frac{1}{b_i})$
21.	$\frac{1}{x+1}, x \in [-2;0], \Delta x = 0{,}1$	$c_i = a_{i+1} + 1/(b_{i+1}-1)$

22.	$\frac{1}{x^2-1}, x \in [-2;0], \Delta x = 0{,}1$	$c_i = a_{i-!} - 2/(b_{i-1}+1)$
23.	$\sqrt{\frac{1}{x^3-2}}, x \in [0;10], \Delta x = 0{,}5$	$c_i = \sqrt{a_{i-1} + b_i - 1}$
24.	$\frac{1}{\sqrt{1-x^2}}, x \in [-3;3], \Delta x = 0{,}3$	$c_i = \sqrt{\frac{a_{i+1}}{2b_i}}$
25.	$\sqrt{x^2-1}, x \in [-2;2], \Delta x = 0{,}2$	$c_i = \sqrt{1 - a_{i+1} \cdot b_i}$
26.	$\arcsin(x+1), x \in [-3;1], \Delta x = 0{,}2$	$c_i = \sqrt{2(a_{i-1} + b_i)}$
27.	$\arccos(\frac{x}{2}-1), x \in [-1;5], \Delta x = 0{,}3$	$c_i = \ln \frac{2a_i}{b_{i-1}+1}$
28.	$\arcsin(\frac{2}{x}+1), x \in [-2;0], \Delta x = 0{,}1$	$c_i = \lg(a_{i+1} + \frac{2}{b_i})$

Лабораторная работа №8. Разработка приложений с применением управляемых провайдеров ADO.NET

1. Цель работы

Изучить общую структуру и основные виды провайдеров технологии ADO.NET. Научиться применять классы и методы, используемые при работе с управляемым провайдером OLE DB

2. Сведения из теории

В платформе .NET определено множество типов (организованных в соответствующие пространства имен), которые помогают обеспечить взаимодействие с локальными и удаленными хранилищами данных. Общее название пространств имен с этими типами – ADO.NET.

Все типы ADO.NET предназначены для выполнения единого набора задач: установить соединение с хранилищем данных, создать и заполнить данными объект **`DataSet`**, отключиться от хранилища данных и вернуть изменения, внесенные в объект **`DataSet`**, обратно в хранилище данных. Объект **`DataSet`** – это тип данных, представляющий локальный набор таблиц и информацию об отношениях между ними.

После создания объекта **`DataSet`** и его заполнения данными можно программными средствами производить запросы к нему и перемещаться по таблицам. Можно выполнять все операции как при работе с обычными базами данных: добавлять в таблицы новые записи, удалять и изменять существующие, применять к ним фильтры и т.п. После того как клиент завершит внесение изменений, информация о них будет отправлена в хранилище данных для обработки.

2.1. Общие сведения об управляемых провайдерах ADO.NET

Управляемый провайдер в ADO.NET – это шлюз к хранилищу данных (например, на сервере баз данных), при помощи которого можно произвести загрузку данных из этого внешнего хранилища в объект **`DataSet`**. Взаимодействие управляемых провайдеров .NET Framework и объекта **`DataSet`** приведено на рисунке 3.1.

Вместе с ADO.NET поставляются несколько управляемых провайдеров:

– провайдер **OLE DB**, который реализуется при помощи типов, определенных в пространстве имен **`System.Data.OleDb`**. Этот провайдер позволяет обращаться к данным, расположенным в любом хранилище, к которому можно подключиться по протоколу OLE DB, например, в базах данных SQL Server 6.5 или ранее, MS Access;

– провайдер **SQL** – предлагает прямой доступ к хранилищам данных на MS SQL Server 7.0 и последующих версий. Типы, используемые провайдером SQL, содержатся в пространстве имен **`System.Data.SqlClient`**;

– провайдер **ODBC** – предназначен для приложений, использующих источники данных ODBC. Типы, используемые этим провайдером, содер-

жатся в пространстве имен **`System.Data.Odbc`**;

– провайдер **Oracle** – предназначен для приложений, использующих источники данных Oracle. Типы, используемые этим провайдером, содержатся в пространстве имен **`System.Data.OracleClient`**.

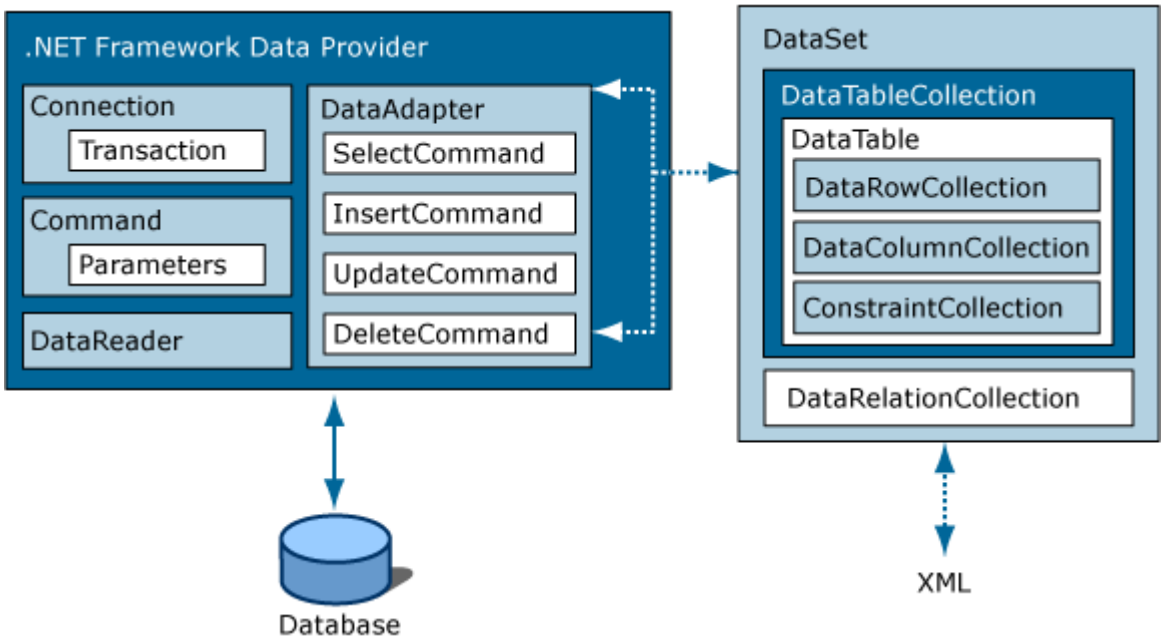


Рис. 3.1. Архитектура ADO.NET

В пространстве имен **`System.Data.Common`** определено множество абстрактных типов, которые обеспечивают общий интерфейс для всех управляемых провайдеров. Так, все провайдеры реализуют интерфейс **`IDbConnection`**, который используется для конфигурирования и открытия сеанса подключения к источнику данных. Типы, которые реализуют другой интерфейс – **`IDbCommand`** – используются для выполнения SQL-запросов к базам данных. **`IDataReader`** обеспечивает считывание данных при помощи однонаправленного курсора только для чтения. Типы, которые реализует **`IDbDataAdapter`**, ответственны за заполнение объекта **`DataSet`** данными из базы данных.

В данной работе будет рассматриваться доступ к данным с использованием управляемого провайдера OLE DB.

2.2. Управляемый провайдер OLE DB

Основные классы пространства имен **`System.Data.OleDb`** представлены в таблице:

Класс	Описание
`OleDbCommand`	Представляет запрос SQL, производимый к источнику данных
`OleDbConnection`	Представляет открытое соединение с источником данных
`OleDbDataAdapter`	Представляет соединение с БД и набор ко-

	манд, используемых для заполнения объекта **DataSet**, а также обновления исходной БД после внесения изменений в **DataSet**
OleDbDataReader	Обеспечивает метод считывания потока данных из источника в одном направлении (вперед)
OleDbErrorCollection **OleDbError** **OleDbException**	**OleDbErrorCollection** представляет набор ошибок и предупреждений, возвращаемых источником данных. Сами эти ошибки и предупреждения представлены объектами **OleDbError**. При возникновении ошибки может быть сгенерировано исключение, представленное объектом **OleDbException**
OleDbParameterCollection **OleDbParameter**	Используются для передачи параметров процедуре, хранимой на источнике данных. Параметры представлены объектами **OleDbParameter**, весь набор – объектом **OleDbParameterCollection**

2.2.1. Установление соединения при помощи типа `OleDbConnection`

При работе с управляемым провайдером OLE DB первое, что нужно сделать – установить соединение с источником данных при помощи класса **OleDbConnection**. Для этого класса предусмотрено использование строки подключения, состоящей из пар имя – значение. С ее помощью можно задать имя компьютера или файла, к которому производится подключение, параметры безопасности подключения, имя базы данных, а также имя самого провайдера OLE DB.

Для подключения к базе данных MS Access используется провайдер Microsoft.Jet.OLEDB.4.0.

После настройки строки подключения следующее, что нужно сделать, - открыть сеанс соединения с источником данных, после этого выполнить нужные действия и разорвать соединение.

Основные компоненты класса **OleDbConnection** приведены в таблице:

Компонент	Описание
BeginTransaction() **CommitTransaction()** **RollBackTransaction()**	Используются для того, чтобы программным образом начать транзакцию, завершить ее или отменить
Close()	Закрывает соединение с источником данных
ConnectionString	Позволяет настроить строку подключения при установлении соединения или получить ее содержание
ConnectionTimeout	Позволяет получить или установить время тайм-

	аута при установке соединения
Database	Позволяет получить или установить название текущей базы данных во время подключения
DataSource	Позволяет получить или установить имя сервера или файла с источником данных
Open()	Открывает соединение с базой данных, используя текущие настройки свойств соединения
Provider	Позволяет получить или установить имя провайдера
State	Позволяет получить информацию о текущем состоянии соединения

2.2.2. Построение команды SQL

Объектно-ориентированным представлением запроса на языке SQL в ADO.NET является класс **OleDbCommand**. Сам текст команды определяется через свойство **CommandText**. Множество типов ADO.NET принимают объект **OleDbCommand** в качестве параметра для того, чтобы передать запрос к источнику данных. Также в классе **OleDbCommand** предусмотрено множество других компонентов, которые позволяют определить характеристики запроса (см. таблицу).

Компонент	Описание
Cancel()	Прекращает выполнение команды
CommandText	Позволяет получить или задать текст запроса на языке SQL, который будет передан источнику данных
CommandTimeout	Позволяет получить время тайм-аута при выполнении команды. По умолчанию это время равно 30 секундам
CommandType	Позволяет получить или установить значение, определяющее, как именно будет интерпретирован текст запроса
Connection	Позволяет получить ссылку на объект **OleDbConnection**, для которого используется данный объект **OleDbCommand**
ExecuteReader()	Возвращает объект **OleDbDataReader**
Parameters	Возвращает коллекцию параметров **OleDbParameterCollection**
Prepare()	Готовит команду к выполнению (например, она будет откомпилирована) на источнике данных

2.2.3. Работа с OleDbDataReader

После того, как соединение с источником данных открыто и создан объект – команда SQL, следующая задача – передать эту команду (запрос) источнику данных. Это можно сделать несколькими способами, но использование **OleDbDataReader** – это наиболее простой, наиболее быстрый способ по-

лучения информации от источника данных … и наименее гибкий. Этот класс представляет однонаправленный (только вперед), доступный только для чтения поток данных, который за один раз возвращает одну строку в ответ на запрос SQL.

Класс **OleDbDataReader** очень полезен, когда необходимо последовательно обработать большое количество данных и в то же время не нужно выполнять с этими данными какие-либо операции в памяти. В отличие от **DataSet**, при использовании **OleDbDataReader** соединение с базой данных не закрывается автоматически, а сохраняется активным до тех пор, пока не будет закрыто явным образом.

2.2.4. Класс OleDbDataAdapter

Класс **OleDbDataAdapter** используется в тех случаях, когда необходимо заполнить полученными с сервера данными объект **DataSet** и выполнить с ними определенные операции.

Основное назначение этого класса – извлечь информацию из источника данных и заполнить ею объект **DataTable** в **DataSet** при помощи метода **OleDbDataAdapter.Fill()**. Метод **Fill()** многократно перегружен, ниже приведены два наиболее часто используемых варианта (возвращаемое значение **int** позволяет получить информацию о количестве записей, полученных из источника данных):

```
//Заполняет объект DataSet данными, полученными из таблицы
//на источнике данных с указанным именем
public int Fill(DataSet ds, string tableName);
//Также заполняет данными, но только теми, которые находятся
//в указанных границах
public int Fill(DataSet ds, string tableName, int startRecord, int maxRecord);
```

Конечно, перед тем, как вызывать этот метод, нужно иметь уже созданный объект **OleDbDataAdapter**. Конструктор **OleDbDataAdapter** также многократно перегружен, но обычно необходимо указать информацию о параметрах подключения к базе данных и команду SELECT на языке SQL, которая будет использована для заполнения **DataTable.**

OleDbDataAdapter позволяет не только заполнять объект **DataTable** внутри **DataSet** данными, полученными из источника, но и помещать измененные данные обратно в источник данных при помощи стандартных команд SQL. Ниже в таблице представлены компоненты класса **OleDbDataAdapter,** которые позволяют это сделать, а также некоторые другие важнейшие элементы этого класса.

Компонент	Описание
DeleteCommand **InsertCommand** **SelectCommand** **UpdateCommand**	Используются для определения того, какая именно команда SQL будет передана на источник данных при вызове метода **Update()**. Каждое из этих свойств определяется при помощи объектов

	OleDbCommand
Fill()	Заполняет указанную таблицу в **DataSet** определенным пользователем количеством записей
GetFillParameters()	Возвращает все параметры, использованные при выполнении SELECT к источнику данных
Update()	Вызывает соответствующие команды INSERT, UPDATE, DELETE к источнику данных для каждой вставленной, измененной или удаленной строки в таблице объекта **DataSet**

При использовании свойств **DeleteCommand**, **InsertCommand**, **SelectCommand**, **UpdateCommand** объект **OleDbDataAdapter** автоматически переводит внесенные изменения в таблицу данных в **DataSet** в соответствующие команды на языке SQL, сохраняя, таким образом, внесенные изменения в источнике данных. Эти свойства позволяют определить соответствующие команды SQL в деталях.

3. Пример выполнения работы

Задание: База данных из двух связанных таблиц «Телефонная книжка» создана в Microsoft Access. Реализовать с помощью управляемого провайдера OLE DB доступ к этой базе данных.

2.1. Создание базы данных в Microsoft Access

База данных **Organizer** состоит из двух связанных таблиц: **Contacts** и **Phones**. Структура таблиц приведена ниже:

Название поля	Тип поля	Размер поля	Примечание	Назначение поля
Таблица Contacts				
Id	Счетчик	Длинное целое	Ключевое поле	Номер записи
Fam	Текстовый	20		Фамилия
Name	Текстовый	20		Имя
Таблица Phones				
PhoneId	Счетчик	Длинное целое	Ключевое поле	Номер записи
ContactId	Числовой	Длинное целое		Идентификатор контакта
Phone	Текстовый	20		Телефон

Таблицы связываются между собой по номеру записи в таблице контактов. Данная связь показана на рисунке 3.2.

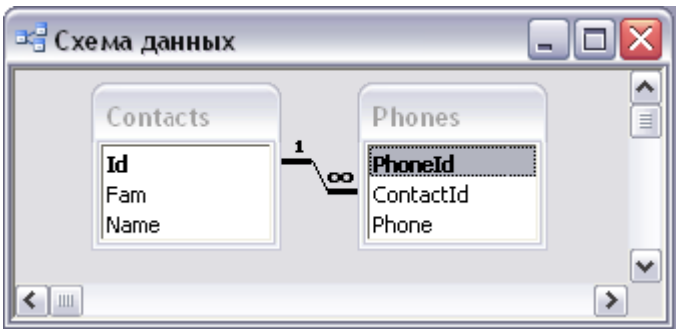


Рис. 3.2. Связывание таблиц БД

3.2. Доступ к данным с помощью управляемого провайдера OLE DB

Прежде всего, следует поместить на форму компонент **`oleDbConnection`** и настроить его свойство **`ConnectionString`**, выбрав пункт **`<New connection...>`**. Открывается диалоговое окно, представленное на рисунке 3.3. С помощью кнопки **`Change`**... в правом верхнем углу окна надо установить вид источника данных (**`Data source`**) как **`Microsoft Access Database file`**. Далее с помощью кнопки **`Browse`**... в строке **`Database file name`** нужно указать путь и имя файла с базой данных. Кнопка **`Test Connection`** позволяет проверить правильность установления соединения.

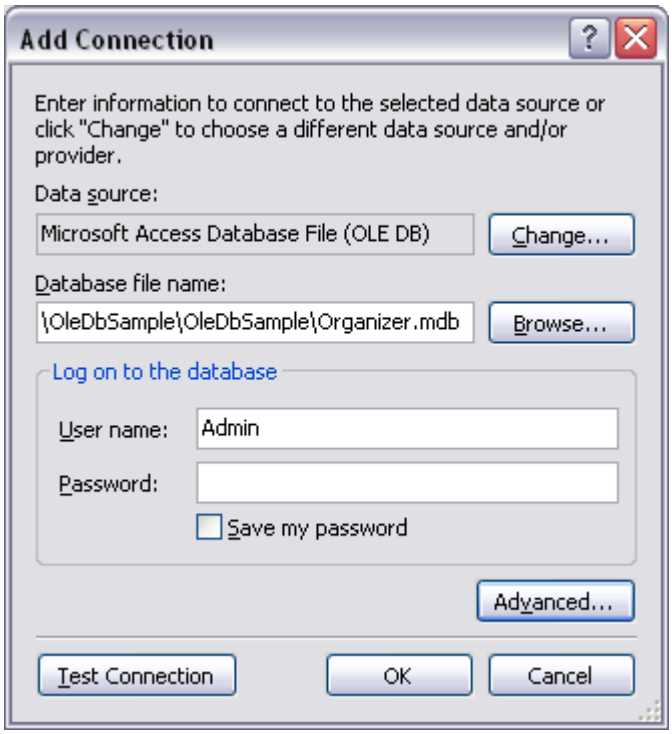


Рис. 3.3. Установление соединения с базой данных

Следующим шагом является создание адаптера – компонента, выполняющего непосредственную передачу данным между приложением и базой данных. Для начала необходимо поместить на форму компонент **`OleDbDataAdapter`**. При этом на экране появляется окно мастера настройки адаптера (рис. 3.4). Здесь следует проверить правильность указания пути к

файлу БД.

Примечание. После нажатия кнопки **Next** мастером настройки будет предложен вариант работы с базой данных, при котором при каждом запуске программе будет создаваться копия базы данных в папке приложения. На данном шаге следует нажать кнопку «NO»!

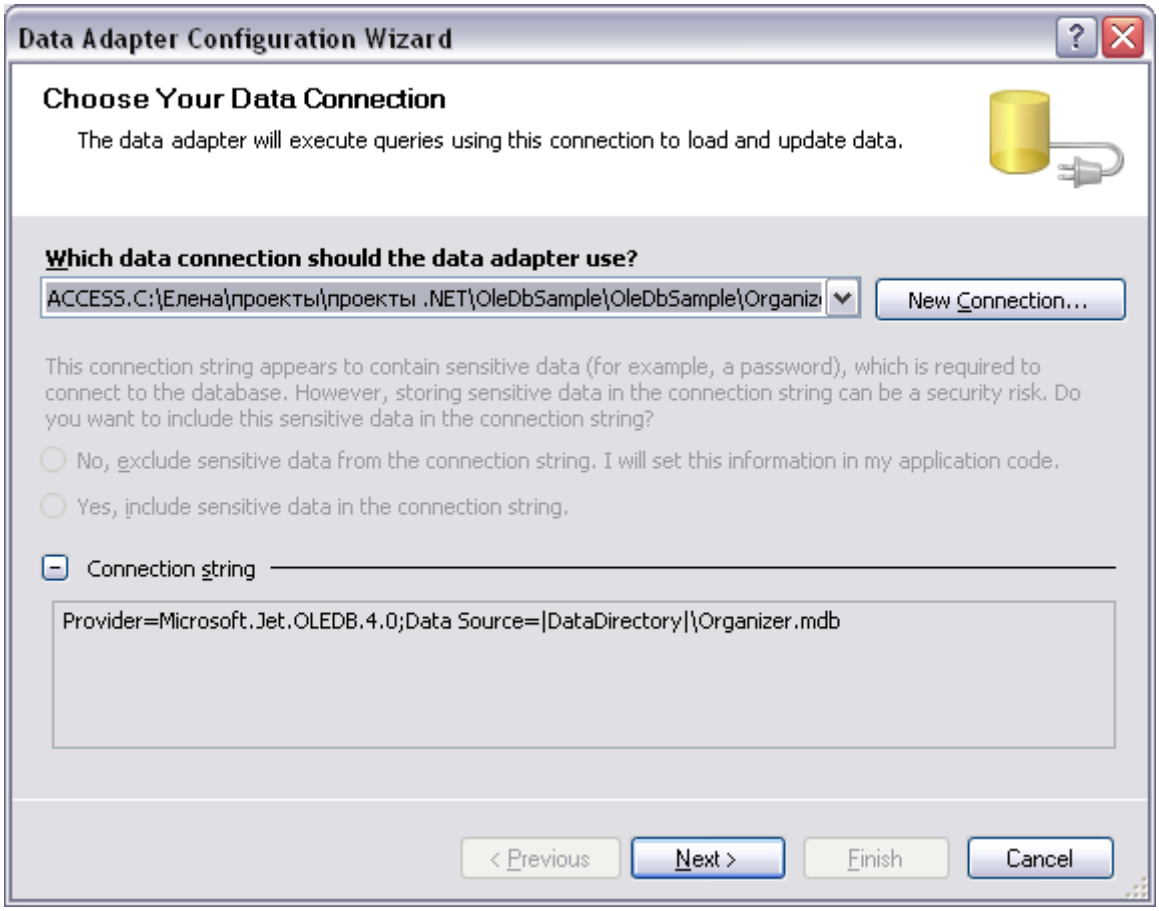


Рис. 3.4. Конфигурирование адаптера данных

Далее мастер предлагает задать команды SQL для загрузки данных в приложение. Кнопка **Query Builder**... данного окна предлагает визуальный метод построения данной команды. В открывшемся окне **Add Table** следует добавить в **Query Builder** главную таблицу БД – таблицу **Contacts**, а затем отметить в этой таблице все ее столбцы (рис. 3.5).

На этом создание адаптера таблицы **Contacts** завершается. Следующим шагом является формирование набора данных – объекта **DataSet**. Для этого нужно в режиме дизайнера формы вызвать контекстное меню и выбрать в нем пункт **Generate DataSet**... Открывается соответствующее диалоговое окно (рис. 3.6), с помощью которого создается новый набор данных.

Аналогичным образом добавляется адаптер и для второй таблицы БД – таблицы **Phones**. Процесс создания данного адаптера аналогичен адаптеру таблицы **Contacts** (в окно **Query Builder**, естественно, нужно добавлять таблицу **Phones**). Чтобы добавить таблицу **Phones** в **DataSet**, следует щелкнуть правой кнопкой мыши на адаптере данной таблицы, в контекстном ме-

ню выбрать **Generate DataSet**... и в открывшемся диалоговом окне выбрать существующий объект **DataSet** (т.е. оставить настройки по умолчанию).

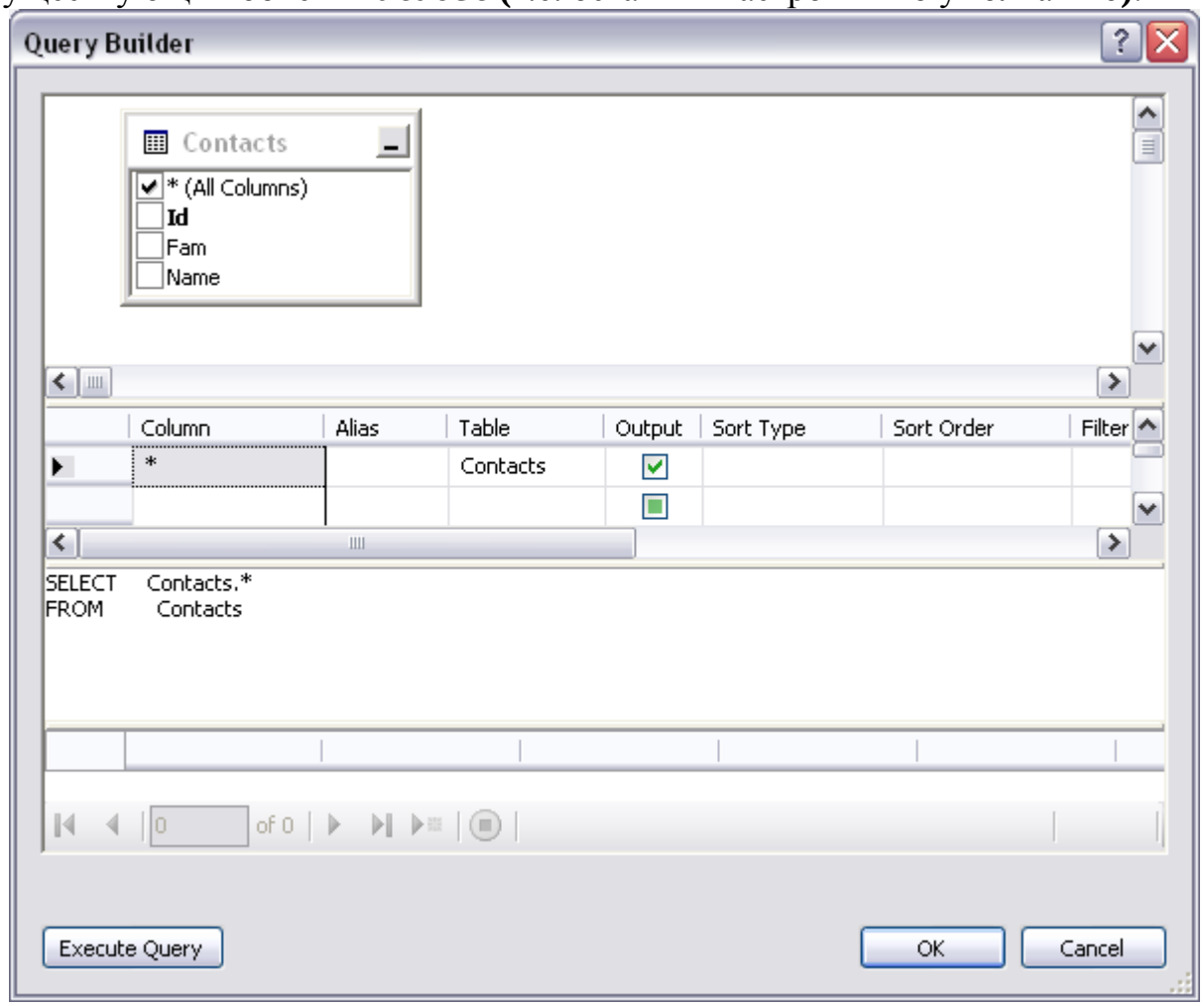


Рис. 3.5. Построение команды SQL для доступа к таблице **Contacts**

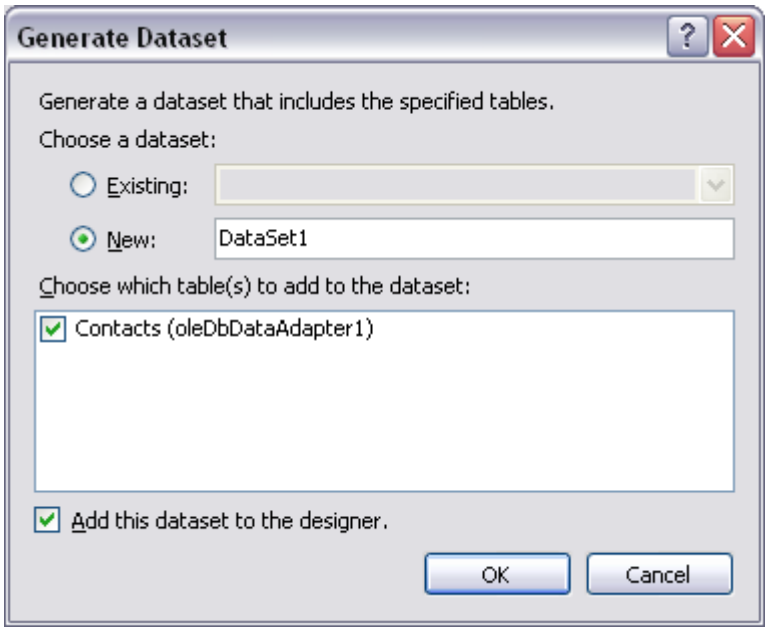


Рис. 3.6. Добавление в проект набора данных (**DataSet**)

После этого в сгенерированный набор данных можно добавлять другие

необходимые элементы: остальные таблицы БД, связи между ними, запросы и т.п. Для этого нужно в окне **`Solution Expolrer`** выбрать (дважды щелкнуть мышью) элемент **`DataSet1.xsd`**. Так, для нашего приложения в набор данных следует добавить отношение – связь между таблицами БД: в окне **`DataSet1.xsd`** в контекстном меню выбирается пункт **`Add->Relation`**... Открывается диалоговое окно редактирования отношения, где можно задать имя создаваемого отношения, родительскую и подчиненную таблицы, ключевые поля, а также режимы автоматических изменений дочерней таблицы при изменении родительской таблицы. Для разрабатываемого приложения создается отношение с именем **`FK_Contacts_Phones`**. Его настройки приведены на рисунке 3.7.

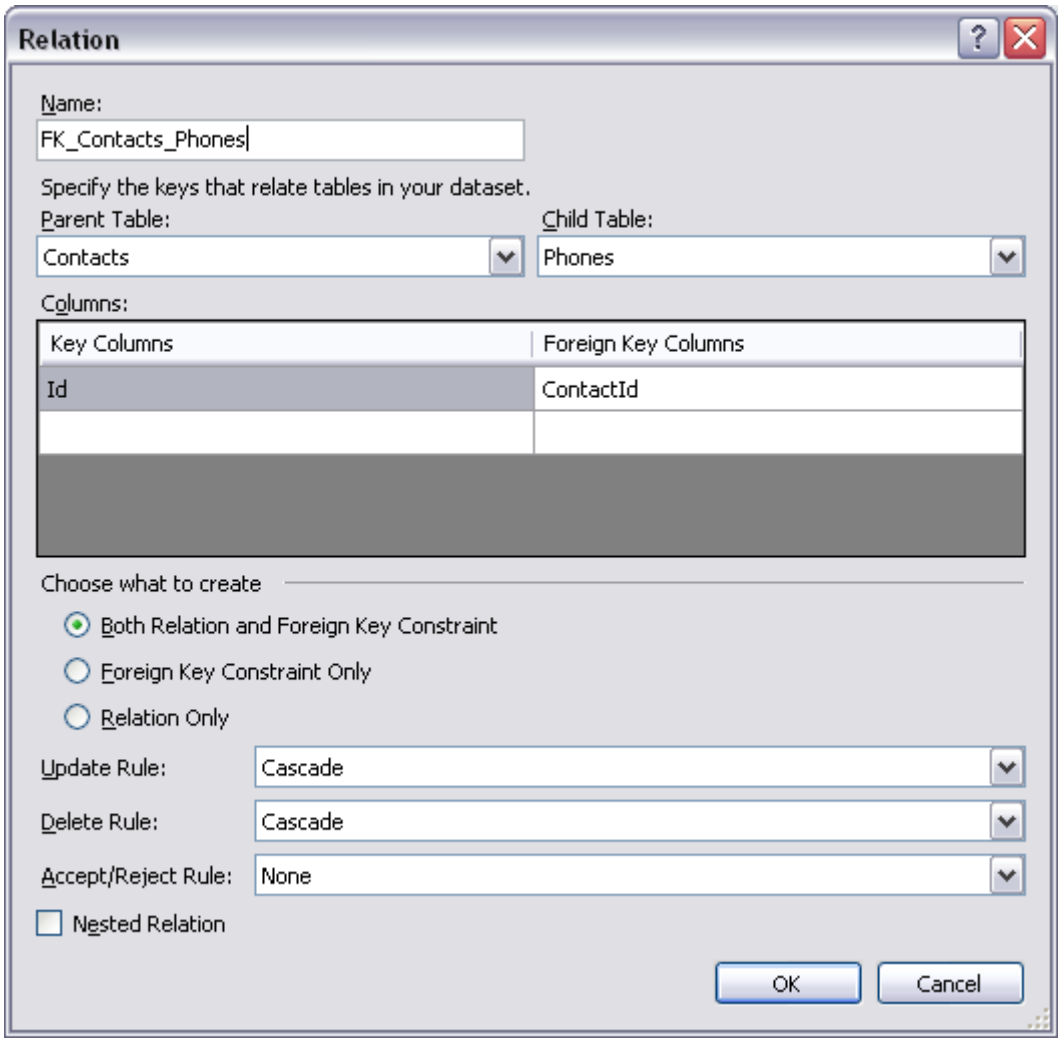


Рис. 3.7. Создание отношения между таблицами

3.3. Визуальное проектирование диалогового окна

Внешний вид работающего приложения приведен на рисунке 3.8.

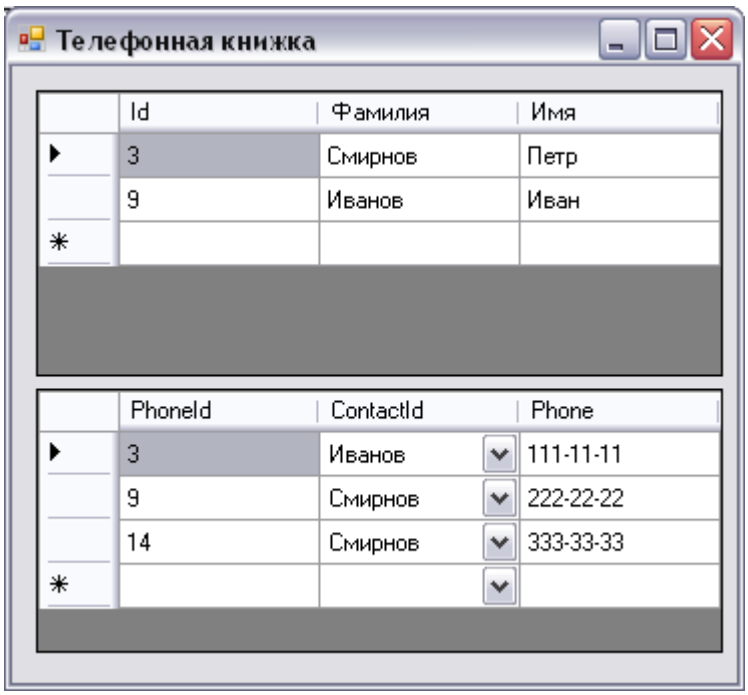


Рис. 5.8. Главная форма приложения

Для отображения данных, полученных из базы, в программе будут использоваться компоненты **DataGridView**, по одному для каждой таблицы базы данных. Имена их по умолчанию устанавливаются как **dataGridView1** и **dataGridView2.**

Компонентам-таблицам **dataGridView1** и **dataGridView2** нужно установить свойство **DataSource**, задающее источник данных для отображения в таблице, в «**contactsBindingSource**» и «**phonesBindingSource**» соответственно. Кроме того, при необходимости можно отредактировать заголовки и другие настройки столбцов таблиц (свойство **Columns**). В данном приложении в столбце **ContactId** компонента **dataGridView2** для наглядности отображается не числовой идентификатор человека, которому принадлежит телефон, а его фамилия, причем для отображения используется тип столбца «**DataGridViewComboBoxColumn**» (свойство **ColumnType**). Настройка остальных свойств данного столбца приведена на рисунке 3.9.

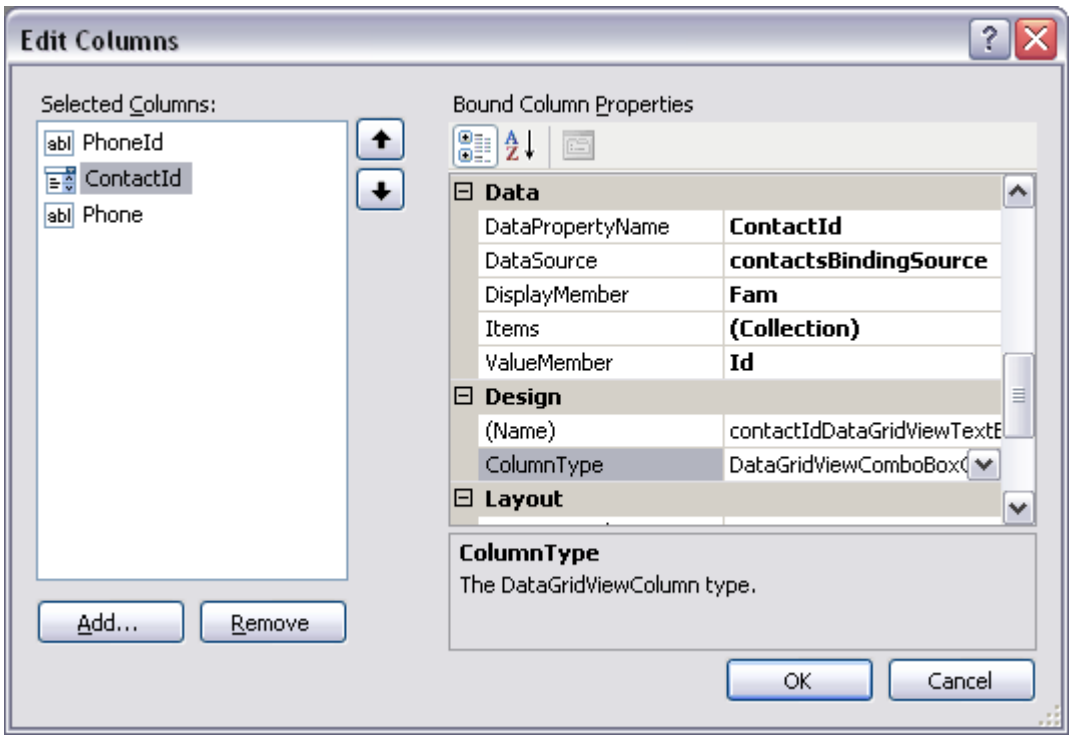


Рис. 3.9. Настройка столбца **ContactId** компонента **dataGridView2**

3.4. Проектирование программного кода

Для отображения в компонентах **DatagridView** данных из таблиц необходимо прежде всего открыть настроенное соединение с базой данных, а затем заполнить таблицы, созданные в объекте **DataSet** данными из таблиц в базе. В нашем случае делается это при загрузке главного окна программы, в обработчике события **Load**:

```
private void Form1_Load(object sender, EventArgs e)
{
    oleDbConnection1.Open(); //открыть соединение
    //заполнить таблицы в объекте DataSet
    oleDbDataAdapter1.Fill(dataSet11.Contacts);
    phonesTableAdapter.Fill(dataSet11.Phones);
}
```

При закрытии формы необходимо отключить соединение с базой данных:

```
private void Form1_FormClosing(object sender,
FormClosingEventArgs e)
{
    oleDbConnection1.Close();   //закрыть соединение
}
```

4. Варианты заданий для самостоятельной работы

В каждом варианте необходимо разработать базу данных минимум из двух связанных между собой таблиц. В каждой таблице – не менее трех полей. Реализовать доступ к созданной базе с помощью управляемого провай-

дера OLE DB.

№ варианта	БД	Таблицы
1.	Студенты университета	Студенты, факультеты, специальности и т.п.
2.	Склад магазина	Товары, поставщики, категории товаров и т.п.
3.	Персонал предприятия	Сотрудники, отделы, документы отдела кадров и т.п.
4.	Владельцы автомобилей	Автомобили, автовладельцы и т.д.
5.	Библиотека	Книги, читатели, книги на руках у читателей и т.п.
6.	Очередь на жилье	Список жилья, список очередников и т.д.
7.	Аптека	Лекарства, категории лекарств, виды болезней и т.п.
8.	Касса аэропорта	Рейсы, проданные билеты и т.д.
9.	Банковские кредиты	Заемщики, виды кредитов, поручители и т.п.
10.	Гостиница	Список номеров, категории номеров, постояльцы и т.п.
11.	Риэлтерская фирма	Квартиры, покупатели, сделки и т.д.
12.	Справка по языку C#	Пространства имен, классы, методы и т.п.
13.	Учет операций с акциями	Виды акций, владельцы, операции и т.д.
14.	Таксопарк	Транспортные средства, водители, рейсы и т.п.
15.	Кафе	Продукты, рецепты, поставщики и т.д.
16.	АЗС	Виды топлива, поставщики, продажи и т.п.
17.	Поликлиника	Врачи, пациенты, консультации и т.д.
18.	Семейный бюджет	Виды поступлений, виды затрат, покупки и т.д.
19.	Табель рабочего времени	Сотрудники, виды работ, табель и т.п.
20.	Туристическая фирма	Виды туров, клиенты, заказы и т.п.
21.	Справочник географа	Континенты, страны, реки, моря и т.д.
22.	ЖЭК	Дома, жильцы, виды обслуживания, заявки и т.п.

23.	Адвокатура	Адвокатские конторы, адвокатские услуги, адвокаты
24.	Пресса	Виды периодического издания, издания, авторы и т.д.
25.	Банкомат	Карточки, виды операций, совершенные операции и т.п.
26.	Касса стадиона	Матчи, категории билетов, проданные билеты и т.д.
27.	АТС	Абоненты, категории звонков, звонки и т.п.
28.	Web-форум	Посетители, темы, сообщения и т.д.
29.	Кинопрокат	Кинотеатры, фильмы в прокате, жанры и т.п.
30.	Учебные курсы	Области знаний курсов, преподаватели, курсы и т.д.

Лабораторная работа №9 Доступ к данным с помощью технологии ADO.NET

1. Цель работы

Изучить принципы доступа к данным с помощью технологии ADO.NET. Получить навыки работы с пространствами имен, классами, методами, используемыми для работы с данными.

2. Сведения из теории

Все возможности ADO.NET заключены в типах, определенных в соответствующих пространствах имен, главным из которых является **System.Data**. Именно это пространство имен и будет рассматриваться в этой лабораторной работе.

2.1. Типы пространства имен System.Data

Эти типы предназначены для представления данных, полученных из источника (но не для установления соединения непосредственно с источником). В основном эти типы представляют собой объектные представления примитивов для работы с базами данных – таблицами, строками, столбцами, ограничениями и т.п. Наиболее часто используемые типы **System.Data** представлены в таблице. Кроме того, в этом пространстве имен определены важные исключения, которые могут быть сгенерированы при работе с БД (**NoNullAllowedException, RowNotInTableException, MissingPrimaryKeyException** и т.п.).

Тип	Назначение
DataColumnCollection DataColumn	**DataColumn** представляет собой один столбец в объекте **DataTable, DataColumnCollection** – все столбцы
ConstraintCollection Constraint	**Constraint** – объектно-ориентированная оболочка вокруг ограничения (например, внешнего ключа или уникальности), наложенного на один или несколько объектов **DataColumn, ConstraintCollection** – все ограничения в объекте **DataTable**
DataRowCollection DataRow	**DataRow** представляет собой единственную строку в **DataTable, DataRowCollection** – все строки в **DataTable**
DataRowView DataView	**DataRowView** позволяет создавать настроенное представление единственной строки, **DataView** – созданное программным образом представление объекта **DataTable**, которое может быть использовано для сортировки, фильтрации, по-

	иска, редактирования и перемещения
DataSet	Объект, создаваемый в оперативной памяти на клиентском компьютере. **DataSet** состоит из множества объектов **DataTable** и информации об отношениях между ними
ForeignKeyConstraint **UniqueConstraint**	**ForeignKeyConstraint** представляет ограничение, налагаемое на набор столбцов в таблицах, связанных отношениями первичный - внешний ключ. **UniqueConstraint** – ограничение, при помощи которого гарантируется, что в столбце не будет повторяющихся записей
DataRelationCollection **DataRelation**	Тип **DataRelationCollection** представляет набор всех отношений (то есть объектов **DataRelation**) между таблицами в **DataSet**
DataTableCollection **DataTable**	Тип **DataTableCollection** представляет набор всех таблиц (объектов **DataTable**) в **DataSet**

2.2. Тип DataColumn

Тип **DataColumn** представляет отдельный столбец в таблице (которая, в свою очередь, должна быть представлена объектом **DataTable**). Наиболее важные свойства этого класса представлены в таблице.

Свойство	Описание
AllowDbNull	Определяет, может ли столбец содержать значения типа **Null** (пустые значения). По умолчанию – может (свойство равно **true**)
AutoIncrement **AutoIncrementSeed** **AutoIncrementStep**	Используются для настройки автоматического приращения значений в таблице. Это может быть полезно, если необходимо обеспечить уникальность значений в столбце (например, для первичного ключа). По умолчанию автоматическое приращение значений в столбцах отключено
Caption	Определяет заголовок столбца для отображения в пользовательском приложении (например, этот заголовок может быть использован в **DataGrid**)
ColumnMapping	Определяет, как будет представлен столбец (объект **DataColumn**) при сохранении **DataSet** в формате XML
ColumnName	Позволяет получить или установить имя столбца в коллекции **Columns** (внутренняя коллекция для столбцов в **DataTable**). Если имя столбца не определено явно, будут использованы значения по умолчанию: **Column1**, **Column2**, **Column3** и т.д.
DataType	Определяет тип данных (**boolean**, **string**, **float** и

	т.п.), используемый для значений в столбце
DefaultValue	Позволяет установить или получить значение по умолчанию для столбца. Это значение будет автоматически использовано, если при вставке новой строки не укажете явно другое значение
Expression	Позволяет получить или установить выражение, используемое для фильтрации новых строк, вычисления значения в столбце или создания столбцов с агрегатными значениями
Ordinal	Позволяет установить порядковый номер столбца в коллекции **Columns** в **DataTable**
ReadOnly	Определяет, будет ли столбец только для чтения. По умолчанию имеет значение **false**
Table	Возвращает **DataTable**, которой принадлежит данный объект **DataColumn**
Unique	Позволяет определить, будут ли в столбце допускаться повторяющиеся значения. Если столбец является первичным ключом, то это свойство должно иметь значение **true**

2.3. Тип DataRow

Как было сказано выше, структура таблицы определяется как коллекция объектов **DataColumn**. Для хранения этой коллекции в объекте **DataTable** используется внутренний объект **DataColumnCollection**. Помимо этого, для **DataTable** важна еще одна коллекция: коллекция объектов **DataRow**, которая и определяет собственно данные, хранящиеся в таблице. При помощи компонентов этого класса можно производить операции вставки, изменения и удаления строк таблицы, а также сравнивать значения, которые содержатся в строках.

Наиболее важные компоненты этого класса представлены в таблице. Кроме того, класс **DataRow** определяет индексатор, при помощи которого можно получить значение из поля строки по порядковому номеру. Конечно, то же самое значение можно будет получить и по имени столбца.

Компонент	Назначение
AcceptChanges() **RejectChanges()**	Для записи в строку (или отказа от них) всех изменений, произведенных начиная с момента, когда последний раз был вызван метод **AcceptChanges()**
BeginEdit() **EndEdit()** **CancelEdit()**	Начать, завершить, прекратить операции редактирования для объекта **DataRow**
Delete()	Помечает строку для удаления при следующем вызове метода **AcceptChanges()**
HasErrors	Свойство **HasErrors** возвращает логическое зна-

GetColumnsInErrors() **GetColumnError()** **ClearErrors()** **RowError**	чение, определяющее, присутствуют ли ошибки в значениях столбцов для данной строки. Если такие ошибки есть, то для получения значений, которые нарушают установленные правила, можно использовать метод **GetColumnsInErrors()**. Для получения описания ошибки можно использовать **GetColumnError()**, а **ClearErrors()** просто удаляет все ошибочные значения из строки. Свойство **RowError** позволяет настроить текстовое описание для ошибки в столбце
IsNull()	Возвращает информацию о том, содержит ли строка в указанном поле пустое значение (типа **NULL**)
ItemArray	Позволяет получить или установить значения всех полей строки при помощи массива объектов
RowState	Позволяет получать информацию о текущем состоянии объекта **DataRow**. Используются значения из перечисления **RowState**
Table	Это свойство используется для получения указателя на таблицу, содержащую текущий объект **DataRow**

Главное назначение свойства **RowState** – определять (в процессе выполнения программы), в каком состоянии находятся выбранные строки в таблице: были ли они изменены, только что вставлены и т.п. Для этого свойства используются значения из перечисления **DataRowState**:

Значение	Описание
Deleted	Строка была изменена при помощи метода **DataRow.Delete**
Detached	Строка была создана, но она еще не является частью **DataRowCollection**. Обычно строка находится в таком состоянии непосредственно после вставки или после принудительного вывода из коллекции **DataRowCollection**
Modified	Строка была изменена, но метод **AcceptChanges()** еще не вызывался
New	Строка была добавлена в коллекцию **DataRowCollection**, но метод **AcceptChanges()** еще не был вызван
Unchanged	Строка не была изменена с момента последнего вызова метода **AcceptChanges()**

2.4. Тип DataTable

Класс **DataTable** используется для создания в оперативной памяти моделей табличных наборов данных. Можно создавать объекты **DataTable** про-

граммным образом, однако чаще в приложениях объект **DataTable** создается автоматически с помощью возможностей **DataSet** и типов, определенных в пространствах имен **System.Data.OleDb** и **System.Data.SqlClient**. Наиболее важные свойства **DataTable** представлены в таблице:

Свойство	Описание
CaseSensitive	Определяет, будет ли при сравнении символьных данных в таблице учитываться регистр символов. По умолчанию – **false** (не будет)
ChildRelations	Возвращает коллекцию подчиненных отношений (**DataRelationCollection**) для объекта **DataTable** (если такие отношения есть)
Columns	Возвращает набор столбцов для таблицы
Constraints	Позволяет получить коллекцию ограничений, определенных в столбце (**ConstraintCollection**)
DataSet	Позволяет получить ссылку на объект **DataSet**, к которому принадлежит данная таблица (если такой объект есть)
DefaultView	Позволяет получить настроенное представление для таблицы, которое может включать в себя, например, только некоторые выбранные пользователем столбцы или данные о положении курсора
MinimumCapacity	Позволяет получить или установить исходное количество строк в таблице (по умолчанию – 25)
ParentRelations	Позволяет получить коллекцию родительских отношений для данного объекта **DataTable**
PrimaryKey	Позволяет получить или установить массив столбцов, которые являются первичным ключом в таблице
Rows	Возвращает набор строк, относящихся к таблице
TableName	Позволяет получить имя таблицы или определить его. Значение для этого свойства может быть установлено через конструктор таблицы

Графическая схема наиболее важных компонентов **DataTable** представлена на рисунке 4.1. Следует обратить внимание, что схема не имеет никакого отношения к иерархии классов (к примеру, класс **DataRow** не является классом, производным от **DataRowCollection**). Эта схема представляет логические отношения «иметь» (“has-a”) между наиболее важными компонентами класса **DataTable** (например, объекты **DataRow** принадлежат к объекту **DataRowCollection**).

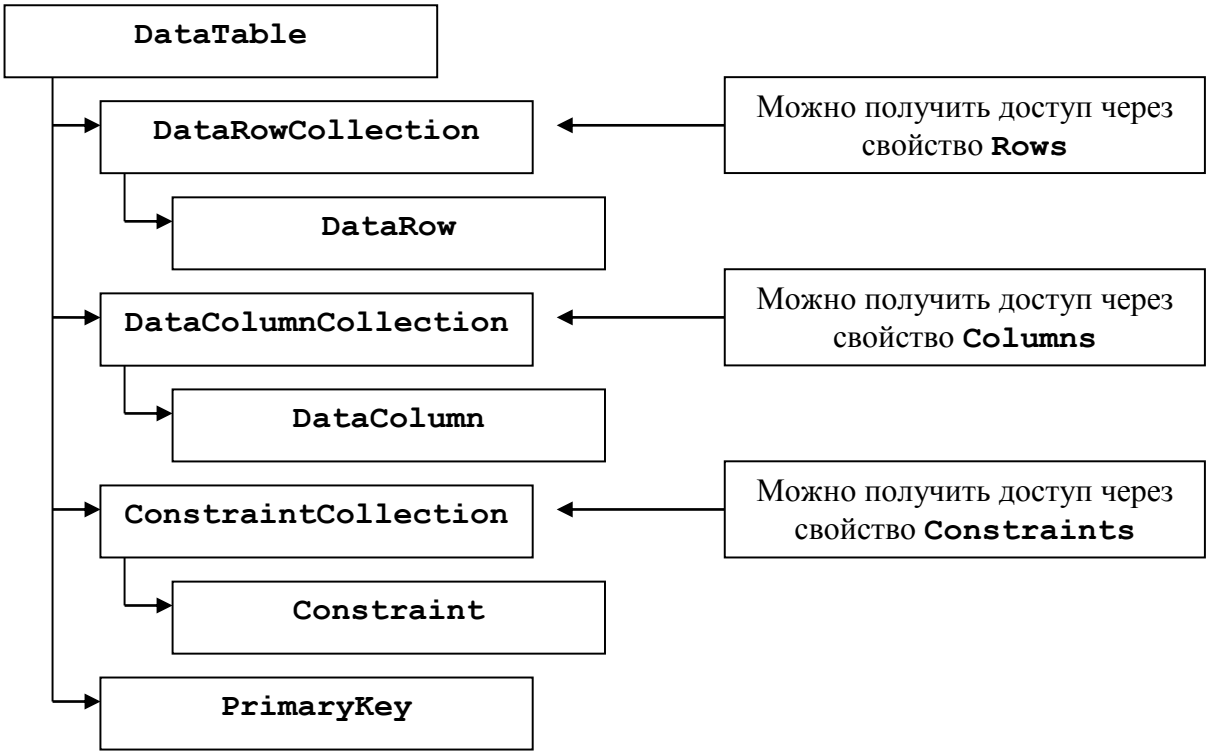


Рис. 4.1. Отношения компонентов **DataTable**

Наиболее важные методы **DataTable** представлены в таблице:

Метод	Описание
AcceptChanges()	Подтверждает все изменения, сделанные в таблице после предыдущего вызова метода
Clear()	Очищает все данные объекта **DataTable**
Compute(String expr, String filter)	Выполняет вычисление выражения **expr** в строках таблицы, удовлетворяющих фильтру **filter**
Copy()	Копирует структуру и данные объекта **DataTable**
NewRow()	Создает новую запись (объект **DataRow**) данной таблицы
RejectChanges()	Отменяет все изменения, сделанные после загрузки таблицы либо после последнего вызова метода **AcceptChanges**
Reset()	Сбрасывает объект **DataTable** в исходное состояние
Select()	Возвращает массив всех записей таблицы (объектов **DataRow**)
Select(String filter)	Возвращает массив записей таблицы (объектов **DataRow**), соответствующих фильтру **filter**

2.5. Возможности класса DataSet

При работе с базами данных чаще всего таблицы используются не сам по себе, а во взаимодействии с другими таблицами. В ADO.NET возможно-

сти работы с наборами таблиц, связанными друг с другом, предоставляет класс **DataSet.**

Объект **DataSet** – это создаваемый в оперативной памяти набор таблиц (объектов **DataTable**), связанных между собой отношениями и снабженными средствами проверки целостности данных (для них в **DataSet** предусмотрены свои объекты). Иерархия классов, входящих в **DataSet**, представлена на рисунке 4.2.

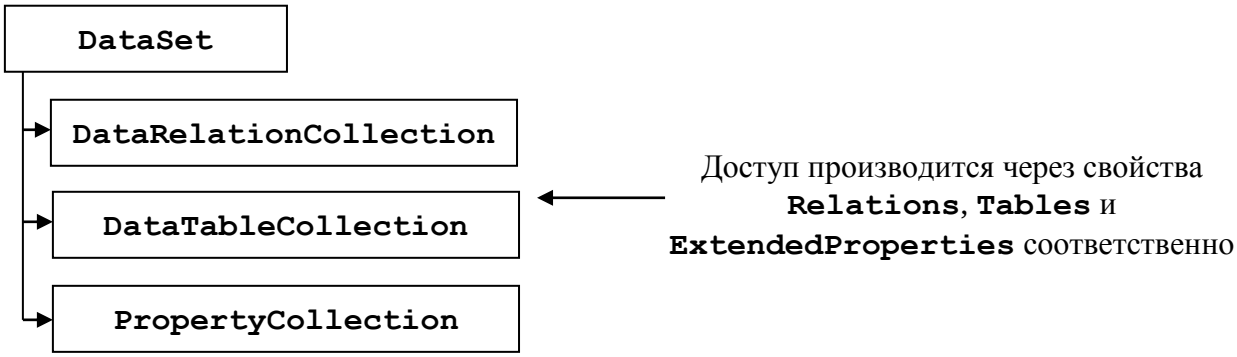


Рис. 4.2. Внутренние коллекции **DataSet**

Основные свойства класса **DataSet** представлены в таблице:

Свойство	Описание
CaseSensitive	Определяет, будет ли во время операций по сравнению текстовых строк в объектах **DataTable** учитываться регистр букв
DataSetName	Позволяет получить или задать имя для данного объекта **DataSet**. Обычно значение этого свойства задается как параметр, передаваемый конструктору
EnforceConstraints	Позволяет включить или отключить проверку соответствия ограничениям при выполнении операций обновления данных в **DataSet**
HasErrors	Позволяет получить значение, определяющее наличие ошибок в **DataSet** (т.е. ошибок в любой строке любой таблицы **DataSet**)
Relations	Позволяет обратиться к коллекции отношений между таблицами **DataSet**
Tables	Позволяет получить доступ к коллекции таблиц **DataSet**

Многие методы **DataSet** дублируют возможности, которые обеспечиваются свойствами. Самые важные методы **DataSet** представлены в таблице:

Метод	Описание
AcceptChanges()	Позволяет сохранить в **DataSet** все изменения, произведенные с момента последнего вызова это-

	го метода
Clear()	Полная очистка **DataSet** – удаляются все строки из всех таблиц
Clone()	Клонирует структуру **DataSet**, включая структуру таблиц, отношения между таблицами и ограничения
Copy()	Копирует **DataSet** (структуру вместе с данными)
GetChanges()	Возвращает копию **DataSet**, которая содержит все изменения, внесенные в оригинальный **DataSet** с момента последнего вызова для него метода **AcceptChanges()**
GetChildRelations()	Возвращает коллекцию подчиненных отношений для указанной таблицы
GetParentRelations()	Возвращает коллекцию родительских отношений для указанной таблицы
HasChanges()	Этот перегруженный метод позволяет получить информацию об изменениях, внесенных в **DataSet** (отдельно по вставленным, удаленным и измененным строкам)
Merge()	Этот перегруженный метод позволяет производить слияние разных объектов **DataSet**
ReadXml() **ReadXmlSchema()**	Позволяют считывать данные в формате XML в **DataSet** из потока (файла, оперативной памяти, сетевого ресурса)
RejectChanges()	Отменяет все изменения , внесенные в **DataSet** с момента его создания или последнего вызова метода **AcceptChanges()**
WriteXml() **WriteXmlSchema()**	Позволяют записывать данные в формате XML из **DataSet** в поток

2.6. Класс DataRelation

После того как в **DataSet** появилось несколько объектов таблиц, можно определить отношения между этими таблицами. Объектно-ориентированную оболочку вокруг отношений между таблицами представляет класс **DataRelation**. При создании объекта этого класса необходимо указать имя отношений, а также родительскую и подчиненную таблицы. Чтобы отношение было успешно установлено, в каждой из таблиц должен быть столбец с одинаковым названием и типом данных.

Объекты **DataRelation** хранятся в коллекции **DataRelationCollection**, поддерживаемой **DataSet**. В типе **DataRelation** предусмотрены свойства, которые позволяют получать ссылки на родительскую и подчиненную таблицу, участвующую в отношении, определять имя отношения и т.п. Наиболее часто используемые свойства представлены в таблице:

Свойство	Описание
ChildColumns **ChildKeyConstraint** **ChildTable**	Позволяют получить информацию о подчиненной таблице, участвующей в отношении, а также ссылку на саму эту таблицу
DataSet	Позволяет получить ссылку на объект **DataSet**, к которому принадлежит данное отношение
ParentColumns **ParentKeyConstraint** **ParentTable**	Позволяют получить информацию о родительской таблице, участвующей в отношении, а также ссылку на саму эту таблицу
RelationName	Позволяет получить или задать имя для данного отношения

Перемещение между таблицами производится при помощи методов, определенных в классе **DataRow**. Так, метод **GetChildRows()** позволяет считывать строки из подчиненной таблицы, а метод **GetParentRows()** – из родительской таблицы.

3. Пример выполнения работы

Задание. Реализовать с помощью технологии ADO.NET программный интерфейс управления базой данных, созданной в предыдущей лабораторной работе. Предусмотреть возможность добавления, редактирования, удаления, фильтрации записей.

3.1. Визуальное проектирование диалогового окна

Внешний вид работающего приложения приведен на рисунке 4.3.

Настройки таблиц **dataGridView1** и **dataGridView1** приведены в описании предыдущей лабораторной работы.

Текстовые поля **FamTextBox** и **NameTextBox** используются для добавления новой записи или редактирования записи, выделенной в таблице. Для удобства редактирования при выделении строки в таблице данные об имени и фамилии человека копируются в соответствующие текстовые поля.

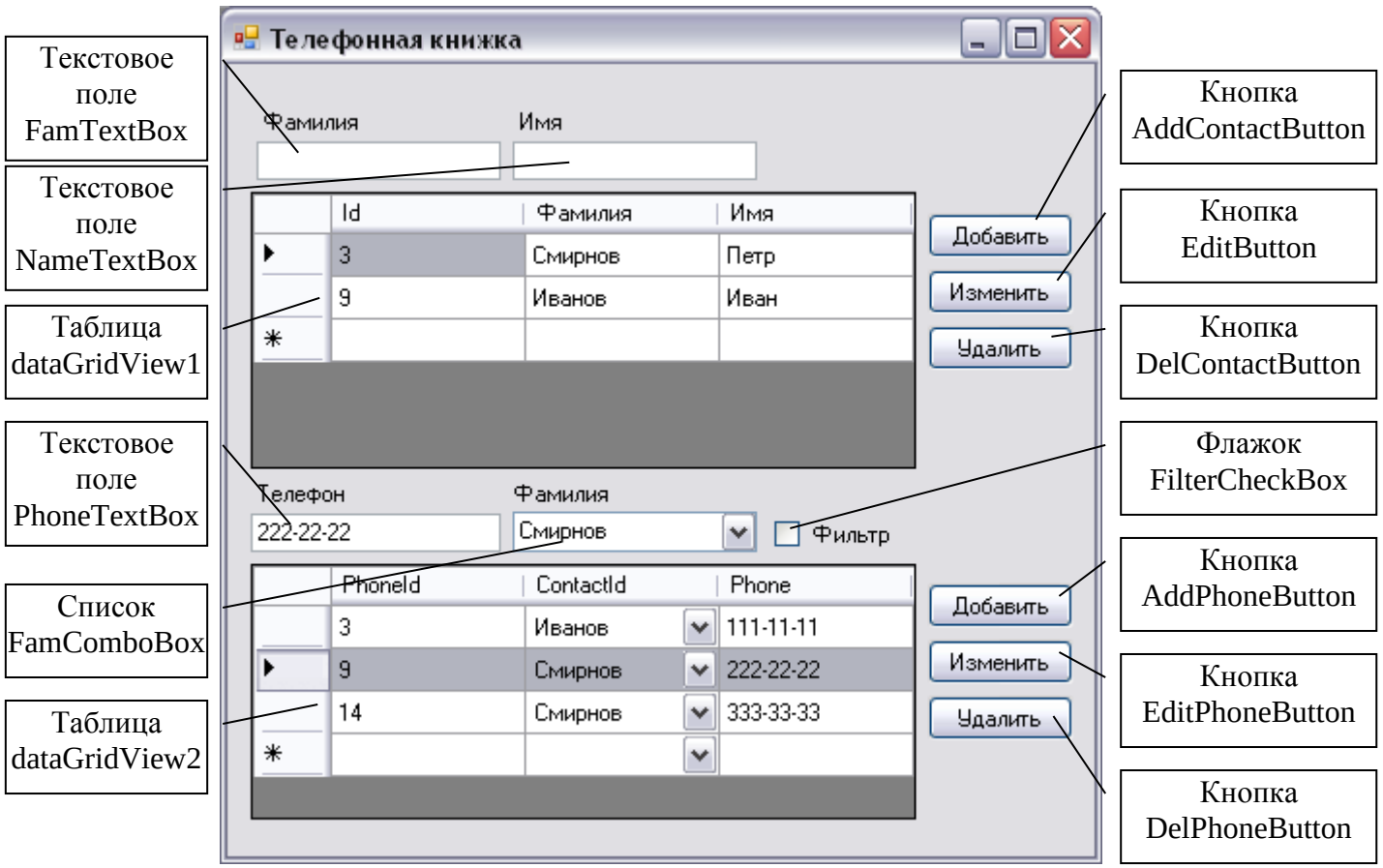


Рис. 4.3. Главная форма приложения

3.2. Проектирование программного кода

3.2.1. Обновление содержимого главного окна приложения

Обновлять содержимое главного окна необходимо, когда пользователь производит какие-либо изменения с записями базы данных: добавление, редактирование, удаление записей таблиц, а также при открытии соединения с базой данных в начале работы программы.

Добавить в класс новый метод можно либо вручную, либо с использованием визуальных средств разработки. Для этого выбрать в меню **`View`** пункт **`Class View`**. В результате откроется окно, где можно просматривать список всех классов, описанных в пространстве имен приложения. Щелкнув правой кнопкой на класс главной формы, следует выбрать в контекстном меню пункт **`View Class Diagram`**. После этого в рабочей области откроется файл диаграммы классов, где можно создавать новые классы, интерфейсы, делегаты, перечисления и т.п., а также добавлять в существующие классы методы, поля, свойства, события и т.д.

Чтобы добавить в класс метод, нужно щелкнуть правой кнопкой в заголовке класса формы и выбрать в контекстном меню пункт **`Add->Method`**. После этого в окне **`Properties`** можно будет установить свойства для добавляемого метода: его имя, вид доступа, тип возврата и т.д.

Для обновления содержимого формы следует добавить в класс формы метод **`UpdateContacts()`**. Тип возврата метода – **`void`**, вид доступа – **`private`**, параметров нет. Программный код метода следующий:

```
private void UpdateContacts()
{
    //обновить содержимое таблиц базы данных
    phonesTableAdapter.Update(dataSet11);
    oleDbDataAdapter1.Update(dataSet11);
    //очистить DataSet
    dataSet11.Clear();
    //заполнить таблицы в объекте DataSet
    oleDbDataAdapter1.Fill(dataSet11.Contacts);
    phonesTableAdapter.Fill(dataSet11.Phones);
    //очистить содерижмое списка фамилий
    FamComboBox.Items.Clear();
    //заполнить список фамилий значениями из таблицы
    foreach (DataRow row in dataSet11.Contacts.Rows)
        FamComboBox.Items.Add(row["Fam"]);
    //в списке фамилий - ни одна фамилия не выделена
    FamComboBox.Text = "";
}
```

Прежде всего, необходимо сохранить сделанные в программе изменения непосредственно в базу данных. Делается это с помощью метода **Update()**, применяемого к адаптерам таблиц. Далее содержимое объекта **DataSet** очищается (**Clear()**) и заполняется заново методом **Fill()**.

Выпадающий список **FamComboBox** содержит список фамилий людей. При изменении содержимого базы данных его также нужно обновить. Сначала его строки методом **Clear()** очищаются, а затем заполняются заново фамилиями из обновленной таблицы (метод **Add()**). Цикл **foreach** используется для просмотра всех записей таблицы.

Так как обновление данных должно выполняться и при запуске программы, то содержимое обработчика события **FormLoad** изменится следующим образом:

```
private void Form1_Load(object sender, EventArgs e)
{
    oleDbConnection1.Open(); //открыть соединение
    UpdateContacts();        //обновить главное окно
}
```

3.2.2. Добавление нового контакта

Добавление нового контакта происходит тогда, когда пользователь щелкает кнопку **Добавить,** расположенную в верхней части главного окна приложения (рис. 8.3). Предварительно пользователь должен ввести имя и фамилию человека в поля **Имя** и **Фамилия** соответственно.

Для этой кнопки необходимо подготовить следующий программный код:

```
private void AddContactButton_Click(object sender, EventArgs e)
{
```

```
    //если текстовые поля не пусты,
    if (FamTextBox.Text != "" && NameTextBox.Text != "")
    {
        //то создать новую запись в таблице Contacts,
        DataRow row = dataSet11.Contacts.NewRow();
        //заполнить ее столбцы
        row["Fam"] = FamTextBox.Text;
        row["Name"] = NameTextBox.Text;
        //и добавить запись в таблицу
        dataSet11.Contacts.Rows.Add(row);
        //обновить содержимое главного окна
        UpdateContacts();
    }
}
```

Здесь нужно убедиться в том, что пользователь ввел как имя, так и фамилию. Если поле **fnTextBox** или **lnTextBox** пустое, обработчик события завершает свою работу без выполнения других дополнительных действий. В том случае, когда пользователь ввел все необходимые данные, обработчик события создает новую строку в таблице **Contacts**. Для этого используется метод **NewRow()**. На следующем этапе введенные строки имени и фамилии записываются в соответствующие столбцы строки. Подготовленная таким способом строка добавляется в таблицу.

Далее обработчик события последовательно вызывает метод с именами **UpdateContacts()**, который предназначен для обновления содержимого формы приложения. Этот метод описан далее.

3.2.3. Добавление номера телефона

Выделив в верхнем списке имя человека, пользователь может добавить для него один или несколько телефонов. Эта операция выполняется при помощи кнопки **Добавить,** расположенной возле списка телефонов. Вот обработчик событий для этой кнопки:

```
private void AddPhoneButton_Click(object sender, EventArgs e)
{
    //если поле ввода телефона не пустое
    //и выбрана какая-либо фамилия
    if (PhoneTextBox.Text != "" && FamComboBox.Text != "")
    {
        try
        {
            //создать новую запись таблицы Phones
            DataRow row = dataSet11.Phones.NewRow();
            //заполнить столбец "номер телефона"
            row["Phone"] = PhoneTextBox.Text;
            //получить из выпадающего списка фамилию,
            //для которой добавляется телефон
            string fio = FamComboBox.SelectedItem.ToString();
            //составить условие для поиска этого человека
```

```
            //в таблице Contacts
            string str = "Fam='" + fio + "'";
            //получить id этого человека в таблице Contacts
            DataRow[] contacts = dataSet11.Contacts.Select(str);
            //заполнить столбец "ContactId" добавляемой записи
            row["ContactId"] = contacts[0]["id"];
            //добавить запись в таблицу
            dataSet11.Phones.Rows.Add(row);
            //обновить форму
            UpdateContacts();
        }
        catch (Exception)
        {
        }
    }
}
```

Здесь вначале проверяется, что поле нового телефона **phoneTextBox** не пустоеи в выпадающем списке выбран человек, для которого будет добавляться телефон. Если это так, то путем вызова метода **NewRow()** в таблице **Phones** создается новая строка как объект класса **DataRow.**

Номер добавляемого телефона сохраняется в столбце **Phone**. Что же касается столбца **ContactId,** то в него нужно записать идентификатор строки таблицы **Contacts** (грубо говоря, порядковый номер человека, которому добавляется телефон). Для этого прежде всего нужно получить фамилию человека из списка **FamComboBox** с помощью свойства **SelectedItem**, затем найти этого человека по фамилии в таблице **Contacts** и получить его **id**. Поиск данных в таблице осуществляется с помощью метода **Select()**, где в качестве параметра выступает условие отбора данных. Это условие записывается в строку **str** и имеет вид: **Fam = '<фамилия человека>'**. Столбец **ContactId** новой записи заполняется найденным значением.

Заполненная строка добавляется в таблицу **Phones** методом **Add().**

И, наконец, после добавления строки обработчик событий обновляет содержимое базы данных в окне программы методом **UpdateContacts().**

3.2.4. Выбор записей в таблицах

Если требуется изменить или удалить данные из базы, то для этого необходимо указать запись, с которой будут производиться эти действия. В данном приложении запись выбирается путем выделения соответствующей строки компонента **DataGridView**. При этом желательно для обеих таблиц на форме установить свойство **MultiSelect** в **false**, чтобы запретить выбор более чем одной строки.

Номер выбранной записи будет сохраняться в переменных: **RowId** для таблицы **Contacts** и **PhoneId** для таблицы **Phones**. Эти переменные следует добавить в класс главной формы либо визуальным способом (как описано в пункте 2.2.1), либо вручную следующим образом:

```
private int RowId;
private int PhoneId;
```

Выбор нужной записи осуществляется в обработчике события **RowHeaderMouseClick** компонентов **DataGridView**. Это событие происходит, когда пользователь щелкает мышью в столбце заголовка строки, т.е. выделяет запись. Код данного обработчика для таблицы **Contacts** представлен ниже:

```
private void dataGridView1_RowHeaderMouseClick(object sender,
DataGridViewCellMouseEventArgs e)
{
    try
    {
        //получить номер выделенной строки
        RowId = e.RowIndex;
        //отобразить фамилию и имя выбранного человека
        //в текстовых полях
        FamTextBox.Text =
          dataSet11.Contacts.Rows[RowId]["Fam"].ToString();
        NameTextBox.Text =
          dataSet11.Contacts.Rows[RowId]["Name"].ToString();
    }
    catch (Exception)
    {
    }
}
```

Когда пользователь выбирает строку в таблице, этот обработчик вначале определяет индекс строки, которая стала выделенной в результате выполнения этой операции, и записывает его в переменную **RowId**. Здесь параметр e обработчика содержит параметры, описывающие характеристики нажатия мышью заголовка строки, в том числе и номер строки (свойство **RowIndex**). После получения номера выбранной строки выполняется копирование полей фамилии и имени человека из таблицы в соответствующие текстовые поля.

Код аналогичного обработчика для таблицы **Phones** выглядит следующим образом:

```
private void dataGridView2_RowHeaderMouseClick(object sender,
DataGridViewCellMouseEventArgs e)
{
    try
    {
        //получить номер выделенной строки
        PhoneId = e.RowIndex;
        //найти запись с полученным номером в DataSet
        DataRow p = dataSet11.Phones.Rows[PhoneId];
        //отобразить номер телефона найденной записи в текстовом поле
        PhoneTextBox.Text = p["Phone"].ToString();
        //получить записи из таблицы Contacts, связанные с данной
```

```
            DataRow[] fams = p.GetParentRows(
              dataSet11.Relations["FK_Contacts_Phones"]);
            //выбрать фамилию человека (из текущей записи) в списке
            FamComboBox.SelectedItem = fams[0]["Fam"].ToString();
        }
        catch (Exception)
        {
        }
}
```

Начало обработчика сходно с описанием предыдущего метода: получение номера выделенной строки в переменную **PhoneId** и копирование номера телефона найденной записи в текстовое поле. Но в данном случае необходимо также отобразить (сделать выбранным) в компоненте **ComboBox,** фамилию человека, к которому относится этот номер телефона. Т.к. фамилии находятся в таблице **Contacts**, то нужно получить список записей, связанных с выбранной через отношение **FK_Contacts_Phones** (см. предыдущую лабораторную работу). Делается это с помощью метода **GetParentRows()**. Далее из полученных «родительских» записей извлекается поле фамилии (**"Fam"**), и данная фамилия становится выбранной в выпадающем списке (свойство **SelectedItem**).

3.2.5. Редактирование записей таблицы Contacts

Для того чтобы отредактировать имя или фамилию человека, нужно выделить требуемую строку в таблице **DataGridView1**, содержащей записи таблицы **Contacts**. После этого только что рассмотренный обработчик события **RowHeaderMouseClick** запишет имя и фамилию в текстовые поля редактирования **Имя** и **Фамилия** соответственно.

Отредактировав имя или фамилию, пользователь должен щелкнуть кнопку **Изменить.** Ниже приведен исходный текст обработчика событий для этой кнопки, изменяющего содержимое ячеек соответствующей строки в таблице **Contacts.**

```
private void EditButton_Click(object sender, EventArgs e)
{
    //если выбрана строка в таблице
    if (dataGridView1.SelectedRows.Count != 0)
    {
        //получить содержимое выбранной строки
        DataRow row = dataSet11.Contacts.Rows[RowId];
        //изменить фамилию и имя на введенные значения
        row["Fam"] = FamTextBox.Text;
        row["Name"] = NameTextBox.Text;
        //сохранить изменения и обновить содержимое формы
        UpdateContacts();
    }
    else
        MessageBox.Show("Выберите строку для редактирования",
```

```
            "Ошибка");
}
```

Здесь проверяется, выбрана ли строка для редактирования. Если нет, то выдается сообщение об ошибке и обработчик завершает работу. Иначе надо получть содержимое выбранной строки в объект типа **DataRow**. Далее значения полей **Fam** и **Name** этой строки заменяются на новые, введенные пользователем в текстовых полях, и вызывается метод **UpdateContacts()** для сохранения изменений в БД и обновления таблиц на форме.

Т.к. таблицы БД связаны между собой, и при связывании было указано каскадное изменение и удаление записей, то после сохранения изменений в БД эти изменения отобразятся и в таблице **Phones**.

3.2.6. Изменение номера телефона

Для изменения номера телефона используется кнопка **Изменить,** расположенная справа от списка телефонов. Исходный текст обработчика событий для этой кнопки представлен ниже:

```
private void EditPhoneButton_Click(object sender, EventArgs e)
{
    //если выбрана запись для редактирования
    if (dataGridView2.SelectedRows.Count != 0)
    {
        try
        {
            //получить содержимое выбранной строки
            DataRow row = dataSet11.Phones.Rows[PhoneId];
            //изменить номер телефона на введенное значение
            row["Phone"] = PhoneTextBox.Text;
            //получить из выпадающего списка фамилию,
            //для которой добавляется телефон
            string fio = FamComboBox.SelectedItem.ToString();
            //составить условие для поиска этого человека
            //в таблице Contacts
            string str = "Fam='" + fio + "'";
            //получить id этого человека в таблице Contacts
            DataRow[] contacts = dataSet11.Contacts.Select(str);
            //заполнить столбец "ContactId" редактируемой записи
            row["ContactId"] = contacts[0]["id"];
        }
        catch (Exception)
        {
        }
        //сохранить изменения и обновить содержимое формы
        UpdateContacts();
    }
    else
        MessageBox.Show("Выберите строку для редактирования",
          "Ошибка");
}
```

Начало обработчика идентично предыдущему: если выбрана строка для редактирования, то получить ее содержимое и изменение значение номера телефона.

Далее, если нужно изменить и человека, к которому этот номер телефона относится, то используется технология, описанная в пункте 2.2.3: поиск человека в таблице **Contacts** с помощью метода **Select()**, получение его порядкового номера (**id**) и заполнение этим значением поля **ContactId** таблицы **Phones.**

3.2.7. Удаление записей таблицы Contacts

Для того чтобы удалить строку, выделенную пользователем в списке контактов, нужно воспользоваться кнопкой **Удалить**. Ниже приведен исходный текст метода, выполняющего обработку событий от этой кнопки:

```
private void DelContactButton_Click(object sender, EventArgs e)
{
    //если выбрана запись для удаления
    if (dataGridView1.SelectedRows.Count != 0)
    {
        if (MessageBox.Show("Вы действительно хотите удалить
          запись?", "Подтверждение",
          MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            try
            {
                //удалить выбранную строку
                dataSet11.Contacts.Rows[RowId].Delete();
            }
            catch (Exception)
            {
            }
            //обновить БД и ее содержимое на форме
            UpdateContacts();
        }
    }
    else
        MessageBox.Show("Выберите строку для удаления",
          "Ошибка");
}
```

После проверки, выбрана ли в таблице строка для удаления и вывода подтверждающего сообщения, непосредственное удаление записи выполняется методом **Delete()**, после чего изменения сохраняются в базу данных и отображаются на форме.

При связывании таблиц **Contacts** и **Phones** в списке **Delete rule** было выбрано значение **Cascade,** задающее режим автоматического внесения изменений в дочернюю таблицу при удалении записей родительской таблицы.

Поэтому при удалении строк родительской таблицы происходит удаление соответствующих строк дочерней таблицы.

3.2.8. Удаление номера телефона

Для удаления номера телефона предназначена кнопка **Удалить,** расположенная возле списка телефонов. Далее приведен обработчик событий для этой кнопки:

```
private void DelPhoneButton_Click(object sender, EventArgs e)
{
    //если выбрана запись для удаления
    if (dataGridView2.SelectedRows.Count != 0)
    {
        if (MessageBox.Show("Вы действительно хотите удалить
          запись?", "Подтверждение",
          MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            try
            {
                //удалить выбранную строку
                dataSet11.Phones.Rows[PhoneId].Delete();
            }
            catch (Exception)
            {
            }
            //обновить БД и ее содержимое на форме
            UpdateContacts();
        }
    }
    else
        MessageBox.Show("Выберите строку для удаления",
          "Ошибка");
}
```

Код обработчика идентичен предыдущему (естественно, работа идет с таблицей **Phones** и компонентом **dataGridView2**).

3.2.9. Фильтрация данных

В данном приложении приведен пример фильтра, который позволяет просматривать номера телефона одного, выбранного человека. Фамилия этого человека выбирается в выпадающем списке **FamComboBox**. Режим фильтрации включается флажком **FilterCheckBox**. Код обработки щелчка по этому флажку приведен ниже:

```
private void FilterCheckBox_CheckedChanged(object sender,
EventArgs e)
{
    //если установлен флажок и выбрана фамилия
    if (FilterCheckBox.Checked && FamComboBox.Text != "")
```

```
    {
        //получить выбранную фамилию
        string fio = FamComboBox.SelectedItem.ToString();
        //составить условие для поиска нужного человека
        //в таблице Contacts
        string str = "Fam='" + fio + "'";
        //найти нужного человека в таблице Contacts
        DataRow[] contacts = dataSet11.Contacts.Select(str);
        //составить условие для фильтра
        str = "ContactId=" + contacts[0]["id"];
        //применить фильтр
        phonesBindingSource.Filter = str;
    }
    else
        //отменить фильтрацию
        phonesBindingSource.Filter = "";
}
```

В начале работы метода проверяется, установил ли пользователь флажок фильтрации и задал ли требуемую фамилию.

Так как фильтрация будет задаваться для таблицы **Phones**, а поле «фамилия» находится в таблице **Contacts**, необходимо получить идентификатор (порядковый номер) требуемой фамилии, чтобы потом найти его в таблице телефонов. Получить выбранную фамилию из выпадающего списка можно методом **SelectedItem**, а отобрать запись с этой фамилией из таблицы **Contacts** – методом **Select().**

Далее из полученной записи выделяется значение поля **id**, на основе которого задается условие для фильтрации: **ContactId = <значение поля id>**. Сформированный фильтр применяется к компоненту **phonesBindingSource** с помощью свойства **Filter**. Этот компонент генерируется автоматически при добавлении адаптеров доступа к базе данных.

Если флажок **FilterCheckBox** не отмечен или в списке не выбрана фамилия, свойству **Filter** присваивается значение пустой строки, т.е. фильтрация отменяется.

4. Задание для самостоятельной работы

Добавить в программу предыдущей лабораторной работы возможность добавления, редактирования и удаления записей базы данных, а также фильтрацию по двум (любым) полям. При этом хотя бы один фильтр должен использовать связи между таблицами (фильтрация осуществляется в одной таблице, а поле для фильтра берется из другой).

Лабораторная работа №10. Разработка графических приложений с использованием GDI+. Построение графиков функций

1. Цель работы

Изучить пространства имен и классы интерфейса графических устройств .NET, основные свойства и методы этих классов, применяемые при сеансах вывода графики. Научиться использовать классы GDI+ для рисования графиков функций.

2. Сведения из теории

2.1. Пространства имен GDI+

Аббревиатура GDI расшифровывается как Graphics Device Interface (интерфейс графического устройства). Этим термином обозначается подсистема Windows, предназначенная для вывода графических изображений (а Windows вся основана на использовании графики) на экран и на принтер. GDI+ – это новый набор программных интерфейсов, используемых в .NET.

В .NET предусмотрено множество пространств имен, предназначенных для вывода двумерных графических изображений. Помимо ожидаемых стандартных типов (например, для работы с цветом, шрифтами, пером, кистью, изображениями) в этих пространствах имен предусмотрены типы для выполнения достаточно изощренных операций, таких как геометрические преобразования, сглаживание неровностей, подготовка палитры, поддержка вывода на принтер и многие другие. Перечень наиболее важных пространств имен для работы с графическими изображениями представлен в таблице.

Пространство имен	Описание
`System.Drawing`	Важнейшее пространство имен GDI+, содержит основные типы для вывода графики (работы со шрифтами, перьями, кистью и т.п.), а также очень важный тип `Graphics`
`System.Drawing.Drawing2D`	Содержит типы для выполнения более сложных операций с двумерной графикой (градиентная заливка, геометрические преобразования и т.п.)
`System.Drawing.Imaging`	Здесь определены типы, которые позволяют напрямую работать с графическими изображениями (менять палитру, работать с метафайлами и т.п.)
`System.Drawing.Printing`	Определяет типы для вывода графики на принтер и взаимодействия с принтером
`System.Drawing.Text`	Позволяет работать с системными шрифтами

2.2. Пространство имен `System.Drawing`. Служебные типы

Большинство типов для работы с графикой находятся в пространстве

имен **System.Drawing**. Некоторые наиболее важные типы этого пространства имен представлены в таблице:

Тип	Описание
Bitmap	Содержит файл изображения и определяет набор методов для выполнения операций с этим изображением
Brush **Brushes** **SolidBrush** **SystemBrushes** **TextureBrush**	Объекты **Brush** (кисть) используются для заполнения пространства внутри геометрических фигур (прямоугольников, эллипсов, многоугольников). Тип **Brush** – абстрактный базовый класс, остальные – производные от него
BufferedGraphics	Новый тип .NET 2.0, обеспечивающий графический буфер для двойной буферизации, которая используется для уменьшения или полного исключения влияния эффекта мелькания, возникающего при перерисовке изображений
Color **SystemColors**	Определяют ряд статических свойств, доступных для чтения и используемых для получения нужного цвета при использовании различных перьев и кистей
Font **FontFamily**	Тип **Font** содержит характеристики шрифта (имя, начертание, размер и т.д.). **FontFamily** представляет набор шрифтов, которые относятся к одному семейству, но имеют небольшие отличия
Graphics	Представляет реальную поверхность для изображения, а также предлагает ряд методов для вывода текста, изображений и геометрических фигур
Icon **SystemIcons**	Представляют пользовательские и системные иконки
Image **ImageAnimator**	**Image** – это абстрактный базовый класс для поддержки возможностей типов **Bitmap, Icon** и **Cursor**. Тип **ImageAnimator** обеспечивает показ изображений через указанные интервалы времени
Pen **Pens** **SystemPens**	**Pen** – это класс, используемый для построения линий и кривых. Тип **Pen** определяет ряд статических свойств, позволяющий получить перо с заданными свойствами (например, с заданным цветом)
Point **PointF**	Эти структуры обеспечивают работу с координатами точки. **Point** работает со значениями типа **int**, а **PointF** – с типом **float**
Rectangle **RectangleF**	Структуры, предназначенные для работы с прямоугольными областями (**int/float**)
Size **SizeF**	Эти структуры обеспечивают работу с размерами: высотой и шириной (**int/float**)
StringFormat	Используется для форматирования текста (выравнива-

	ние, междустрочный интервал и т.д.)
Region	Определяет область, занятую геометрической фигурой

Многие методы требуют указания положения или области для вывода графического объекта. Другим часто используемым методам необходимо передавать размеры (высоту или ширину) прямоугольной области, в которую будет производиться вывод, или, если область вывода будет не прямоугольной, задать эту область другим образом.

Для передачи методам подобной информации в пространстве имен **System.Drawing** предусмотрены служебные типы **Point, Rectangle, Region** и **Size**. Основные свойства данных типов приведены в таблице:

Структура	Свойство	Назначение свойства
Point **PointF**	**X**	Координата **x**
	Y	Координата **y**
Size **SizeF**	**Width**	Ширина
	Height	Высота
Rectangle **RectangleF**	**Left**	Координата **x** левой грани
	Right	Координата **x** правой грани
	Top	Координата **y** верхней грани
	Bottom	Координата **y** нижней грани
	Width	Ширина прямоугольника
	Height	Высота прямоугольника
	X	Аналогично **Left**
	Y	Аналогично **Top**
	Location	Левый верхний угол
	Size	Размер прямоугольника

2.3. Сеансы вывода графики. Класс Graphics

Работа с графическими устройствами, такими, как принтер, дисплей в Windows вообще и в .NET в частности является аппаратно-независимой. Это значит, что при программировании под Windows средств прямого доступа к аппаратуре нет. Все взаимодействие с ней производится через специальные методы. При этом для вывода на графические устройства используется один и тот же набор функций.

Для того чтобы определить, на какое устройство осуществляется вывод (весь экран, клиентская область окна, принтер и т.д.), используется понятие *контекста устройства* (device context). Это объект класса **Graphics,** содержащий все методы для построения изображения в окне. Кроме того, он содержит данные о графическом устройстве вывода. Для осуществления вывода создается контекст устройства и тем самым определяется конкретное устройство для вывода. А далее к созданному объекту можно применять все имеющиеся методы класса **Graphics.**

Этот класс не только представляет «поверхность» для размещения изоб-

ражения (форма, элемент управления или область в памяти), но определяет также множество методов для вывода текста, изображений, геометрических фигур. Частичный список методов данного класса представлен в таблице:

Метод	Описание
FromHdc()	Статические методы, обеспечивающие возможность получения объекта **Graphics** из элемента управления или изображения
FromHwnd()	
FromImage()	
Clear()	Закрашивает объект **Graphics** заданным цветом
DrawArc()	Рисует сегмент круга между заданными углами
DrawBezier()	Рисует гладкую кривую через заданные 4 точки
DrawBeziers()	Рисует гладкую кривую через заданный массив точек
DrawCurve()	Рисует гладкую кривую через заданный массив точек
DrawEllipse()	Рисует эллипс
DrawIcon()	Рисует изображение с заданной иконкой
DrawLine()	Рисует одиночную прямую линию
DrawLines()	Рисует серию ломаных линий по массиву точек
DrawPie()	Рисует сектор эллипса
DrawPath()	Рисует замкнутый контур на основе серии точек и кривых
DrawRectangle()	Рисует прямоугольник
DrawRectangles()	Рисует несколько прямоугольников
DrawString()	Рисует текст
FillEllipse()	Рисует закрашенный эллипс
FillPath()	Рисует закрашенный замкнутый контур
FillPie()	Рисует закрашенный сектор эллипса
FillPolygon()	Рисует закрашенный замкнутый контур по серии точек
FillRectangle()	Рисует закрашенный прямоугольник

Класс **Graphics** не допускает непосредственного создания своего объекта с помощью ключевого слова **new**, поскольку этот класс не имеет открытых конструкторов. Поэтому контекст устройства можно только получить из элемента управления или изображения, а затем ссылку на него присвоить объекту класса **Graphics**. Например:

```
Graphics G = PictureBox1.CreateGraphics();
```

2.4. Системы координат GDI+

Система координат по умолчанию использует в качестве единицы измерения пиксели, а в качестве исходной точки – верхний левый угол. Координата X определяет смещение вправо, а координата Y – смещение вниз.

Разница между измерением положения относительно верхнего левого угла документа и относительно верхнего левого угла экрана настолько важна, что в GDI+ предусмотрены специальные наименования для этих координатных систем:

– *мировые координаты* (world coordinates) – указывают позицию точки, измеренную в пикселях от левого верхнего угла документа;

– *страничные координаты (page coordinates) – указывают позицию точки, измеренную в пикселях от левого верхнего угла клиентской области;*

– *координаты устройства (device coordinates) – подобны страничным координатам за исключением того, что в качестве единиц измерения используются не пиксели, а другие единицы измерения, определяемые пользователем через свойство PageUnit класса Graphics (дюймы, миллиметры и т.д.).*

Как было сказано выше, по умолчанию точкой отсчета для системы координат является верхний левый угол клиентской области окна. Однако бывают ситуации, когда удобнее, чтобы точка отсчета была расположена в другом месте. Для этого используется метод **`TranslateTransform()`** класса **`Graphics`**. Например, установить точку отсчета в положение 100, 100 относительно системы координат по умолчанию можно следующим образом:

```
g.TranslateTransform(100, 100);
```

2.5. Графический вывод текста. Работа со шрифтами

Основной класс для работы со шрифтами в GDI+ – это класс **`Font`**. Объекты этого класса представляют конкретные шрифты, установленные на компьютере. В этом классе предусмотрено множество перегруженных конструкторов, но наиболее часто используются следующие варианты:

```
//создаем объект Font, указывая имя шрифта и его размер
Font f = new Font("Times New Roman", 12);
//создаем объект Font, указывая имя, размер и стиль
Font f2 = new Font("WingDings", 50, FontStyle.Bold |
      FontStyle.Underline);
```

При создании **`f2`** использовались стили из перечисления **`FontStyle`**. При этом можно задавать несколько стилей одновременно. Значения из перечисления **`FontStyle`** представлены в таблице:

Элемент перечисления `FontStyle`	**Стиль**
`Bold`	Полужирный
`Italic`	Курсив
`Regular`	Обычный текст
`Strikeout`	Зачеркнутый
`Underline`	Подчеркнутый

После настройки необходимых параметров объекта **`Font`** нужно передать их методу **`DrawString()`** класса **`Graphics`**. Несмотря на то что этот метод многократно перегружен, как правило, приходится указывать стандартный набор информации: текстовую строку, которая будет выводиться, используе-

мые шрифт и кисть (цвет текста), а также координаты вывода. Например:

```
G.DrawString("My string",f,new SolidBrush(Color.Red),40,40);
```

Для того чтобы предоставить пользователю возможность выбрать нужный шрифт для вывода, используется стандартное диалоговое окно выбора шрифта (рис. 1.1).

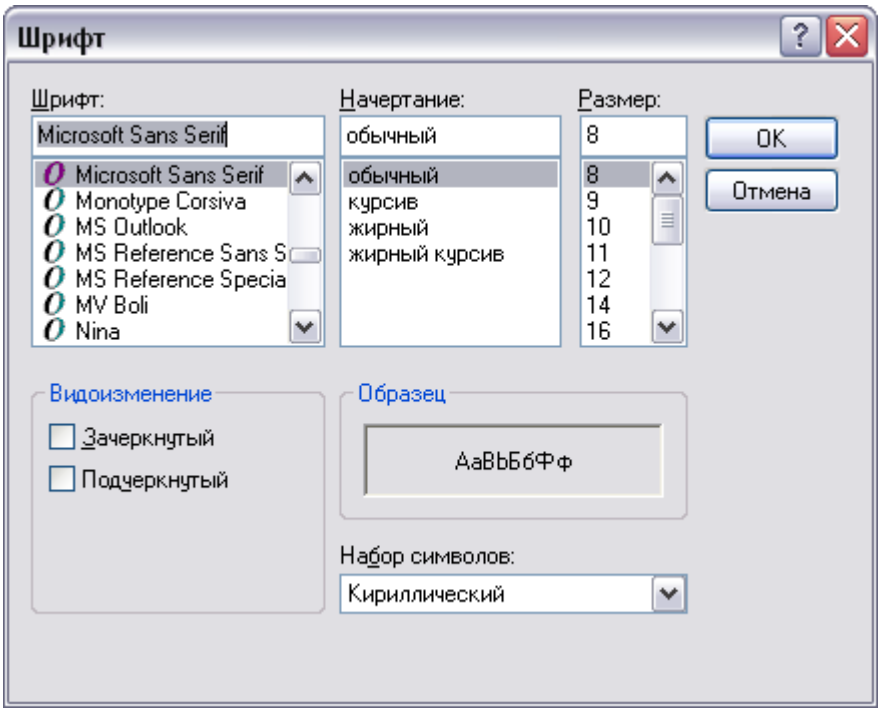


Рис. 1.1. Стандартное диалоговое окно выбора шрифта

Для управления данным диалоговым окном в библиотеке .NET Framework служит класс **`FontDialog`**. Чтобы вызвать на экран такое окно, надо создать объект класса **`FontDialog`** и применить к нему метод **`ShowDialog()`**. При визуальном проектировании Windows-приложения вместо того, чтобы создавать объект вручную, можно просто поместить на форму компонент **`FontDialog`** из панели инструментов, тогда объект класса диалога будет создан автоматически.

Далее с помощью свойства **`Font`** можно получить выбранный пользователем шрифт. Список основных свойств класса **`FontDialog`** приведен в таблице:

Свойство	Описание
`AllowVerticalFonts`	Разрешает / запрещает выбирать вертикальные шрифты
`Color`	Цвет шрифта
`Font`	Шрифт
`MaxSize`	Максимально допустимый размер шрифта
`MinSize`	Минимально допустимый размер шрифта
`ShowColor`	Разрешает / запрещает выбирать цвет шрифта
`ShowEffects`	Разрешает / запрещает выбирать эффекты (подчерк-

	нутый, зачеркнутый шрифты и т.п.)

2.6. Рисование графиков функций

Любой график функции можно рассматривать как некоторую кривую. Кривая – это гладкая линия, представляющая собой *сплайн*. Настоящий сплайн - это гибкий прут, с помощью которого конструкторы когда-то вычерчивали кривые, изгибая его на плоскости вокруг торчащих из нее гвоздей. Сплайн должен пройти через фиксированный набор точек так, чтобы получающаяся линия была гладкой и не имела изломов. Форма линии зависит от гибкости прута.

Для рисования графика функции можно использовать метод **`DrawCurve()`** класса **`Graphics`**. Чтобы нарисовать кривую с помощью этого метода, нужно определить ключевые точки сплайна и его упругость. Простейшая форма метода **`DrawCurve()`** такова:

```
DrawCurve(перо, массив_точек, упругость);
```

Если упругость равна нулю, то соседние точки будут соединены ломаными линиями, т.е. никакого сглаживания не будет. По мере увеличения значения упругости график будет все более и более сглаживаться.

3. Пример выполнения работы

Задание. Построить график функции $y = e^{-4x} \cdot |cos(15x)|$. Самостоятельно выбрать удобные параметры настройки. Пользователь задает количество точек графика, коэффициент упругости, а также шрифт для подписей осей координат.

3.1. Визуальное проектирование диалогового окна

Внешний вид окна приложения с описанием используемых компонентов приведен на рисунке 1.2.

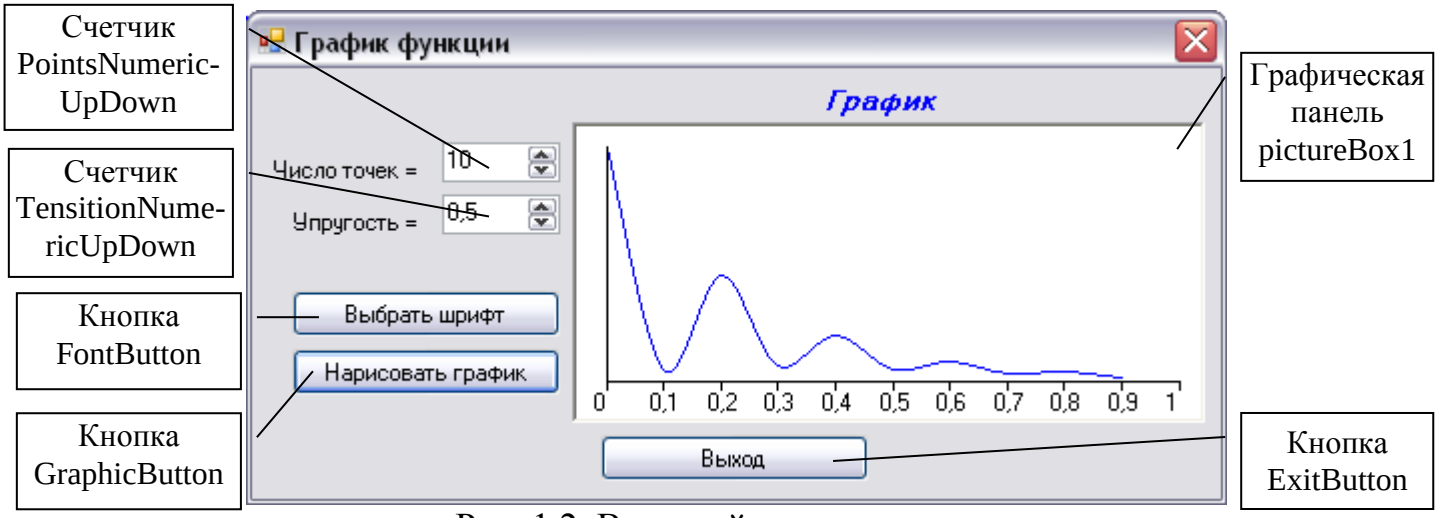


Рис. 1.2. Внешний вид приложения

Для графической панели **`pictureBox1`** устанавливаются следующие дополнительные свойства:

Свойство	Значение	Описание
BorderStyle	**Fixed3D**	Стиль границы
BackColor	**White**	Цвет фона

Коэффициент упругости графика может принимать значение в диапазоне от 0 до 1. Поэтому для счетчика **TensitionNumericUpDown** устанавливаются следующие свойства:

Свойство	Значение	Описание
Minimum	0	Минимальное значение
Maximum	1	Максимальное значение
Increment	0,1	Шаг инкремента
DecimalPlaces	1	Количество знаков после запятой

Также на форму нужно поместить компонент **FontDialog** (диалоговое окно выбора шрифта), для которого следует установить в **true** свойство **ShowColor**. Благодаря этому свойству в окне можно будет выбирать не только характеристики шрифта, но и его цвет. При желании можно также изменить шрифт по умолчанию (свойство **Font**).

3.2. Проектирование программного кода

Для обеспечения графического вывода данному приложению понадобятся кисть и шрифт. Поэтому в класс формы необходимо добавить объекты классов **Font** (шрифт) и **SolidBrush** (сплошная кисть):

```
Font font;
SolidBrush brush;
```

Создание этих объектов может производиться в обработчике события **Load** или в конструкторе формы:

```
//шрифт берем установленный по умолчанию
font = fontDialog1.Font;
//создаем сплошную кисть черного цвета
brush = new SolidBrush(Color.Black);
```

При нажатии на кнопку «Выбрать шрифт» на экране должно появляться соответствующее диалоговое окно. Если после выбора настроек шрифта пользователь нажимает кнопку OK, то выбранные установки должны передаться объектам **font** и **brush**:

```
private void FontButton_Click(object sender, EventArgs e)
{
    //если выбор шрифта завершен нажатием кнопки OK
    if (fontDialog1.ShowDialog() == DialogResult.OK)
    {
```

```
        //получить параметры шрифта из диалогового окна
        font = fontDialog1.Font;
        //получить цвет шрифта из того же окна
        brush.Color = fontDialog1.Color;
    }
}
```

Функцию, для которой будет рисоваться график, можно задать отдельно:

```
private double f(double x)
{
    return Math.Exp(-4 * x) * Math.Abs(Math.Cos(15 * x));
}
```

Обработчик нажатия кнопки «Нарисовать график» состоит из нескольких частей:

–описание и инициализация вспомогательных переменных, задающих начало координат, масштабы по осям x и y, а также число точек графика;

–создание и инициализация графического объекта;

–создание и заполнение массива точек, по которым будет строиться график, при этом производится преобразование физических координат точек в экранные с учетом значений масштабов;

–непосредственно рисование графика и осей координат;

–разметка оси x с выводом числовых значений.

Более подробное описание кода содержится в комментариях к листингу:

```
private void GraphicButton_Click(object sender, EventArgs e)
{
    //Начало координат графика
    int x0 = 15;
    int y0 = (int)(pictureBox1.Height * 0.85);
    //Масштаб по оси X
    int Mx = pictureBox1.Width - 2 * x0;
    //Масштаб по оси Y
    int My = -y0 + 10;
    //Число точек графика
    int M = (int)PointsNumericUpDown.Value;

    //Создание графического объекта
    Graphics G = pictureBox1.CreateGraphics();
    //Очистка PictureBox1
```

```
    G.Clear(Color.White);

    //Описание и создание массива точек
    Point[] p = new Point[M];
    //Цикл по числу точек графика
    for (int n = 0; n < M; n++)
    {
        //Физические координаты
        double x = (double)n / M;
       double y = f(x);
        //Экранные координаты
       int xi = (int)(x0 + Mx * x);
       int yi = (int)(y0 + My * y);
        //заносим в массив вычисленные значения координат
       p[n] = new Point(xi, yi);
    }
    //коэффициент упругости графика
    float tensition = (float)TensitionNumericUpDown.Value;

    //рисование графика
    G.DrawCurve(Pens.Blue, p, tensition);
    //Рисование оси X
    G.DrawLine(Pens.Black, x0, y0, x0 + Mx, y0);
    //Рисование оси Y
    G.DrawLine(Pens.Black, x0, y0, x0, y0 + My);

    //Разметка оси X
    for (int n = 0; n <= 10; n++)
    {
        //физическая координата штриха
        double x = n / 10.0;
        //экранная координата штриха
        int xi = (int)(x0 + Mx * x);
        //Наносим штрих
        G.DrawLine(Pens.Black, xi, y0, xi, y0 + 4);
        //Наносим число
        G.DrawString(x.ToString(), font, brush, xi - 9, y0 + 4);

    }
}
```

После создания данного обработчика при нажатии на кнопку «Нарисовать график» в компоненте **PictureBox** отображается график функции. Однако, если окно программы перекрыть другим окном или перетащить его (или только часть окна с графиком) за пределы экрана, а затем вернуть назад, то рисунок (или его часть) исчезнет. Происходит это из-за того, что каждый раз при появлении Windows-окна на экране его необходимо перерисовывать. Перерисовка самой формы и компонентов (кнопок, флажков и т.д.) производится автоматически операционной системой, а перерисовка всей выводимой в окне графики должна производиться программистом.

Если форма или какой-то компонент поврежден, генерируется событие

Paint, которое и нужно обрабатывать для перерисовки. Сгенерировать событие **Paint** вручную можно с помощью метода **Invalidate().**

Таким образом, для данного приложения весь код рисования нужно перенести из обработчика щелчка кнопки «Нарисовать график» в обработчик события **Paint** компонента **pictureBox1**. При этом строка

```
Graphics G = pictureBox1.CreateGraphics();
```

заменяется на

```
Graphics G = e.Graphics;
```

т.е. графический объект для рисования мы получаем из параметра обработчика.

Обработчик кнопки «Нарисовать график» теперь будет содержать всего одну строчку – принудительный вызов перерисовки графической панели.

```
private void GraphicButton_Click(object sender, EventArgs e)
{
    pictureBox1.Invalidate();
}
```

Теперь график будет выведен в панели постоянно: при запуске программы, при перекрытии окна, при нажатии кнопки (с новыми параметрами).

4. Варианты заданий для самостоятельной работы

Построить график функции, вывести, разметить и подписать оси координат (обе!). Предусмотреть возможность установки количества точек и коэффициента упругости графика, а также возможность выбора шрифта с помощью стандартного диалогового окна. Подобрать параметры осей координат, обеспечивающие наглядность (для этого сначала можно построить график в Excel, чтобы оценить его параметры).

1. $y=\dfrac{x^{\cos x}}{\left|x+e^{x}\right|+tgx}$ при $0{,}4 \le x \le 6$.
2. $y=\dfrac{2\cos(x-\pi/6)}{1/2+\sin^2(1+\dfrac{x^2}{3})}$ при $0 \le x \le 1$.
3. $y=x\sin x^3+x+\dfrac{x^2}{2!}+\dfrac{x^3}{3!}$ при $1 \le x \le 3$.
4. $y=\left|\cos^2 x\right|\dfrac{x-\dfrac{2}{x-2}}{1+(1-x)^2}$ при $-1 \le x \le 1$.
5. $y=e^{-2x}\sin(2x+1)-\sqrt{|x+2|}$ при $-0{,}5 \le x \le 4{,}5$.
6. $y=\sqrt{x^2+5}-10\sin^3(x+1)$ при $-2 \le x \le 8$.
7. $y=e^{-x}\dfrac{tgx^2+x^3}{x\cos^2 2x}$ при $0{,}05 \le x \le 0{,}55$.

8. $y=\frac{x^2(x+1)}{2}-\sin^2\sqrt{x+2}$ при $-1\le x\le 1$.

9. $y=\frac{x^2+1}{\sin 3x}+\frac{\sqrt{x/2}}{\cos 3x}$ при $0{,}2\le x\le 2{,}2$.

10. $y=\sin^3\left(x^2+\pi/3\right)^2-\sqrt{\frac{x}{2}}$ при $2\le x\le 12$.

11. $y=10^{-x}\sqrt{\left|\sin 2x+\cos x^2\right|}$ при $0\le x\le 3$.

12. $y=\frac{e^{-\frac{x}{2}}tg^2 3x}{1+\left|\cos x^3\right|}$ при $-2{,}3\le x\le -1{,}8$.

13. $y=2tg^2 x-\frac{1}{\frac{1}{2}\sin^2\frac{x}{2}}$ при $2{,}4\le x\le 3{,}4$.

14. $y=\ln\left(1+x^2\right)+\sin^2\left(\frac{x}{3}\right)$ при $-2\le x\le 1$.

15. $y=\frac{2^{3x}+10^{-x}\cos(x+\pi/3)}{x\sin x}$ при $1{,}2\le x\le 2{,}2$.

16. $y=\frac{x^{\sin x}}{\left|x+e^x\right|+\ln x}$ при $0{,}4\le x\le 4$.

17. $y=\frac{2\cos(x-\pi/6)}{1/2-\cos^2(1+\frac{x^2}{3})}$ при $0\le x\le 2{,}5$.

18. $y=x\cos x^3+x+\frac{x^2}{2!}-\frac{x^3}{3!}$ при $0\le x\le 4$.

19. $y=\left|\sin^2 x\right|\frac{x-\frac{2}{x-2}}{1+(1+x)^2}$ при $-2\le x\le 1{,}6$.

20. $y=e^{-2x}\sin(2x+1)+\sqrt{|x-2|}$ при $0\le x\le 3$.

21. $y=\sqrt{x^2+5}-10\sin^2(x-1)$ при $2\le x\le 10$.

22. $y=e^{-x}\frac{tgx^2-x^3}{x\sin^2 2x}$ при $0{,}2\le x\le 0{,}5$.

23. $y=\frac{x^2(x-1)}{2}+\sin^2\sqrt{x-2}$ при $2\le x\le 4$.

24. $y=\frac{x^2+1}{\sin 2x}-\frac{\sqrt{x/2}}{\cos 2x}$ при $0{,}5\le x\le 2{,}5$.

25. $y=\cos^3\left(x^2+\pi/3\right)^2+\sqrt{\frac{x}{2}}$ при $0 \le x \le 4$.

26. $y=10^{-x}\sqrt{\left|\sin 2x-\cos x^2\right|}$ при $0{,}5 \le x \le 2{,}5$.

27. $y=\frac{e^{-\frac{x}{2}}tg^2 3x}{1+\left|\sin x^3\right|}$ при $-2 \le x \le 1$.

28. $y=2tg^2 x+\frac{1}{\frac{1}{2}\sin^2\frac{x}{3}}$ при $1{,}4 \le x \le 1{,}8$.

29. $y=\ln\left(1+x^2\right)-\cos^2\left(\frac{x}{3}\right)$ при $-1 \le x \le 2$.

30. $y=\frac{2^{3x}-10^{-x}\sin\left(x+\pi/3\right)}{x\sin x}$ при $0{,}2 \le x \le 2{,}2$.

Лабораторная работа №11 Разработка приложения, имитирующего движение графических объектов

1. Цель работы

Изучить теоретические принципы использования графических объектов GDI+ и получить практические навыки разработки программ, имитирующих движение графических объектов.

2. Краткая теория

2.1. Работа с перьями

Все методы класса **Graphics**, предназначенные для рисования фигур или текста, получают через один из параметров перо класса **Pen** или кисть класса **Brush**, с помощью которых и выполняется рисование.

Перья используются для рисования линий и простейших геометрических фигур и создаются как объекты класса **Pen**. Вот соответствующие конструкторы:

```
public Pen(Color);
public Pen(Color, float);
public Pen(Brush);
public Pen(Brush, float);
```

Первый из этих конструкторов создает перо заданного цвета. Цвет задается при помощи объекта класса **Color**. Второй конструктор позволяет дополнительно задать толщину пера. Третий и четвертый конструктор создают перо на основе кисти, причем в четвертом конструкторе можно указать толщину создаваемого пера.

После того как перо создано, программа может определить его атрибуты при помощи свойств класса **Pen**. Некоторые из этих свойств перечислены в таблице.

Свойство	Описание
Alignment	Выравнивание пера
Width	Ширина линии
Brush	Кисть, используемая пером
Color	Цвет пера
DashStyle	Стиль пунктирных и штрихпунктирных линий
DashCup	Вид точек и штрихов пунктирных и штрихпунктирных линий
DashOffset	Расстояние от начала линии до начала штриха
DashPattern	Массив шаблонов для создания произвольных штрихов и пробелов штриховых и штрихпунктирных линий
StartCup **EndCup**	Стиль концов линий
LineCap	Формы концов линий
LineJoin	Стиль соединения концов двух различных линий
MiterLimit	Предельная толщина в области соединения остроконечных

	линий

Устанавливая значение свойства **Color** и **Width**, приложение может изменить соответственно цвет и ширину линии, рисуемой пером.

Если надо нарисовать пунктирную или штрихпунктирную линию, приложение должно задать необходимое значение для свойства **DashStyle**. При этом допускается изменять вид точек и тире пунктирных и штрихпунктирных линий (свойство **DashCup**), задавать расстояние от начала линии до начала штриха (свойство **DashOffset**) или даже вовсе задать произвольный вид для штрихов и разделяющих эти штрихи пробелов (свойство **DashPattern**).

При необходимости изменить внешний вид концов линий используйте свойства **StartCup** и **EndCup**, задающие стиль концов линий. Свойство **LineCap** определяет форму концов линий.

Если нужно указать стык между двумя различными линиями, то стиль этого стыка задается свойством **LineJoin**, а предельная толщина стыка - стилем **MiterLimit.**

При помощи перьев можно рисовать не только прямые линии, но и любые геометрические фигуры.

2.2. Работа с кистью

Внутренняя область окна и замкнутых геометрических фигур может быть закрашена при помощи кисти. В приложениях Microsoft .NET Framework кисти создаются на базе классов, производных от абстрактного класса **Brush**. Это следующие классы:

-Brushes;
-SolidBrush;
-HatchBrush;
-TextureBrush;
-LinearGradientBrush;
-PathGradientBrush.

2.2.1. Кисть для сплошной закраски

Простейшие из кистей - это кисти **Brushes** и **SolidBrush**, предназначенные для сплошной закраски фигур. Эти кисти создаются при помощи конструктора с одним параметром, задающим цвет в виде объекта класса **Color.**

Ниже приведена часть приложения, в котором кисть класса **Brushes** применяется для создания перьев, с помощью которых приложение рисует прямоугольник и эллипс. Кроме этого, кисть черного цвета создается и для рисования текста:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.Clear(Color.White);
    g.DrawString(Text,new Font("Helvetica",12),Brushes.Black,0,0);
    g.DrawRectangle(new Pen(Brushes.Black, 2), 10, 30, 200,100);
```

```
    g.DrawEllipse(new Pen(Brushes.Black, 2), 150, 120, 100,130);
}
```

2.2.2. Кисти типа `HatchBrush`

При помощи класса **`HatchBrush`** можно создать прямоугольную кисть заданного стиля с заданным цветом изображения и фона.

Для создания кистей этого типа предусмотрено 2 конструктора:

```
public HatchBrush(HatchStyle, Color);
public HatchBrush(HatchStyle, Color, Color);
```

Первый из этих конструкторов позволяет создать кисть заданного стиля и цвета, а второй дополнительно позволяет указать цвет фона.

Далее в таблице перечислены различные стили кисти **`HatchBrush`**, представляющие собой константы перечисления **`HatchStyle`.**

Константы	Описание
`BackwardDiagonal`	Линии штриховки располагаются в обратном направлении (от верхнего правого угла к нижнему левому углу кисти)
`Cross`	Пересекающиеся горизонтальные и вертикальные линии
`DarkDownwardDiagonal`	Диагональные линии, идущие в направлении снизу вверх, и расположенные на 50 % плотнее, чем при использовании константы **`ForwardDiagonal`** (темная штриховка)
`DarkHorizontal`	Горизонтальные линии, которые на 50 % плотнее, чем при использовании константы **`Horizontal`** (темная штриховка)
`DarkUpwardDiagonal`	Диагональные линии, плотнее на 50 % чем при использовании константы **`BackwardDiagonal`** (темная штриховка)
`DarkVertical`	Вертикальные линии, которые на 50 % плотнее, чем при использовании константы **`Vertical`** (темная штриховка)
`DashedDownwardDiagonal`	Штриховые диагональные линии, идущие в обратном направлении
`DashedHorizontal`	Штриховые горизонтальные линии
`DashedUpwardDiagonal`	Штриховые диагональные линии, идущие в прямом направлении
`DashedVertical`	Штриховые вертикальные линии
`DiagonalBrick`	Диагональная «кирпичная» штриховка
`DiagonalCross`	Пересекающиеся прямые и обратные диагональные линии
`Divot`	Штриховка в виде дерна
`DottedDiamond`	Прямые и обратные диагональные пересекающиеся линии, состоящие из отдельных точек

DottedGrid	Горизонтальные и вертикальные пересекающиеся линии, состоящие из отдельных точек
ForwardDiagonal	Прямые диагональные линии, идущие в направлении от верхнего левого угла к нижнему правому углу кисти
Horizontal	Горизонтальные линии
HorizontalBrick	Горизонтальные «кирпичные» линии
LargeCheckerBoard	Штриховка в виде шахматной доски с крупными клетками
LargeConfetti	Штриховка в виде конфетти
LargeGrid	Пересекающиеся горизонтальные и вертикальные линии (то же, что и **Cross**)
LightDownwardDiagonal	Светлая обратная диагональная штриховка
LightHorizontal	Светлая горизонтальная штриховка
LightUpwardDiagonal	Светлая прямая диагональная штриховка
LightVertical	Светлая вертикальная штриховка
Max	То же, что и **SolidDiamond**
Min	То же, что и **Horizonal**
NarrowHorizontal	Средняя горизонтальная штриховка
NarrowVertical	Средняя вертикальная штриховка
OutlinedDiamond	Пересекающиеся прямые и обратные диагональные линии штриховки
Percent05, **Percent10,** **Percent20,** **Percent30** **...** **Percent90**	Эти константы задают процентное соотношение цвета штриховки и цвета фона кисти
Plaid	Штриховка в виде пледа
Shingle	«Кровельная» штриховка
SmallCheckerBoard	Штриховка в виде шахматной доски с мелкими клетками
SmallConfetti	Штриховка в виде мелкого конфетти
SmallGrid	Штриховка в виде мелкой сетки
SolidDiamond	Штриховка в виде шахматной доски, расположенная по диагонали
Sphere	Штриховка с использованием сферических фигур
Trellis	Штриховка в виде решетки
Vertical	Вертикальные линии
Wave	Волнообразные линии
Weave	Штриховка в виде ткани
WideDownwardDiagonal	Широкие обратные диагональные линии
WideUpwardDiagonal	Широкие прямые диагональные линии
ZigZag	Зигзагообразные горизонтальные линии

Для демонстрации использования кистей класса **HatchBrush** ниже представлен исходный текст обработчика событий **Form1_Paint**:

```
using System.Drawing.Drawing2D;
...
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    HatchBrush hb = new HatchBrush(HatchStyle.Cross, Col-
or.Black, Color.White);
    g.FillRectangle(hb, 10, 30, 200, 100);
    g.DrawRectangle(new Pen(Brushes.Black,1),10,30,200,100);
    HatchBrush hb1 = new HatchBrush(HatchStyle.DottedGrid, Col-
or.Black, Color.YellowGreen);
    g.FillEllipse(hb1, 150, 120, 100, 130);
    g.DrawEllipse(new Pen(Brushes.Black,1),150,120,100,130);
    HatchBrush hb2 = new HatchBrush(HatchStyle.Divot, Col-
or.Tomato, Color.Tan);
    g.FillEllipse(hb2, 60,160,60,60);
    g.DrawEllipse(new Pen(Brushes.Blue, 2), 60, 160, 60, 60);
}
```

Как видно, здесь последовательно создаются 3 различные кисти, а затем используются для закраски внутренних областей прямоугольника и эллипсов. Результат работы приложения показан на рисунке 2.1.

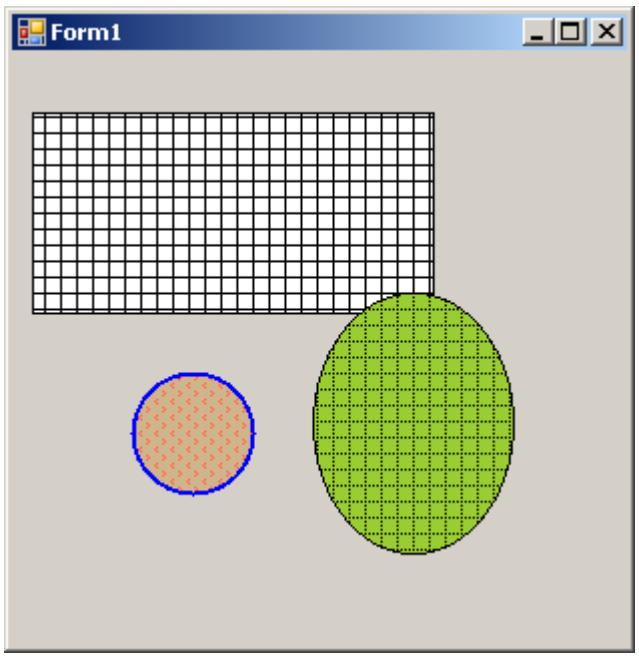


Рис. 2.1 – Использование кистей класса **HatchBrush**

2.2.3. Кисти типа TextureBrush

Если представленные выше кисти не подходят, то есть возможность создать собственную кисть на базе **TextureBrush** в виде произвольного изображения. Такая кисть, называемая текстурной, может иметь любой внешний вид и

любой цвет.
Для создания кисти класса **TextureBrush** приложение может воспользоваться одним из следующих конструкторов:

```
public TextureBrush(Image);
public TextureBrush(Image, Rectangle);
public TextureBrush(Image, RectangleF);
public TextureBrush(Image, WrapMode);
public TextureBrush(Image, Rectangle, ImageAttributes);
public TextureBrush(Image, WrapMode, Rectangle);
public TextureBrush(Image, WrapMode, RectangleF);
```

Самому простому из этих конструкторов нужно передать изображение, загруженное из ресурсов приложения или из внешнего файла. Структуры **Rectangle** и **RectangleF** позволяют задать границы прямоугольной области, ограничивающие изображение кисти. С помощью констант перечисления **WrapMode** программа может задать способ размещения текстуры по горизонтали и вертикали. Эти константы приведены в таблице.

Константа	Описание
Clamp	Текстура кисти «прикрепляется» к границе объекта
Tile	При закраске текстура кисти повторяется по вертикали и горизонтали
TileFlipX	Аналогично предыдущему, но изображение в соседних столбцах заменяется зеркальным, отражаясь по вертикали
TileFlipY	Аналогично **Tile**, но отражение происходит по горизонтали
TileFlipXY	Отражение и по вертикали и по горизонтали

И, наконец, параметр **ImageAttributes** позволяет задать различные атрибуты изображения, такие, как количество цветов и способ рисования. Описание этого параметра и класса **ImageAttributes** можно найти в электронной справочной документации системы Microsoft Visual Studio .NET.
Закраску с помощью текстурной продемонстрировано в приведенном ниже коде:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Image myBrushImage = new Bitmap(GetType(), "L_GREY.GIF");
    TextureBrush tb = new TextureBrush(myBrushImage);
    g.FillRectangle(tb, 10, 30, 200, 100);
    g.DrawRectangle(new Pen(Brushes.DarkBlue,2),10,30,200,100);
}
```

После получения контекста устройства создается новое изображение класса **Image**, загружаемое из ресурсов приложения:

```
Image myBrushImage = new Bitmap(GetType(), "L_GREY.GIF");
```

Обратите внимание, что мы создаем объект класса **Bitmap**, а полученную в результате ссылку присваиваем объекту класса **Image**. Предполагается, что

перед трансляцией приложения вы скопировали файл текстуры **L_GREY.GIF** в ресурсы приложения, а также установили значение свойства **Build Action** для файла изображения, равное **Embedded Resource**. Результат закраски прямоугольной области текстурной кистью показан на рисунке 2.2.

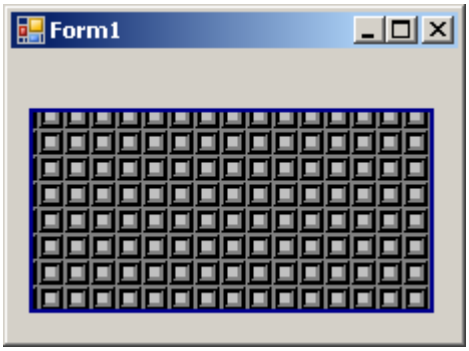


Рис. 2.2 – Закрашивание прямоугольника кистью типа **TextureBrush**

2.2.4. Градиентные кисти

Приложениям GDI+ доступен еще один вид кистей – так называемые градиентные кисти. Линейная градиентная кисть **LinearGradientBrush** позволяет задать в кисти переход от одного цвета к другому. Кисти с множественным градиентом **PathGradientBrush** позволяют задать внутри кисти область, которая будет закрашена с использованием цветового градиента.

Рассмотрим фрагмент кода, демонстрирующего возможности закрашивания при помощи градиентной кисти:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Rectangle rect = new Rectangle(10, 10, 50, 50);
    LinearGradientBrush gb = new LinearGradientBrush(rect, Col-
or.White, Color.Black, LinearGradientMode.BackwardDiagonal);
    g.FillRectangle(gb, 10, 30, 200, 100);
    g.DrawRectangle(new Pen(Brushes.DarkBlue, 2),10,30,200,100);
}
```

Прямоугольная область задает пространство, в котором происходит изменение цвета. Конструктору класса **LinearGradientBrush** передаются координаты этой области, значения двух цветов, а также режим градиентного закрашивания **LinearGradientMode.**

Результат работы данного приложения представлен на рисунке 2.3.

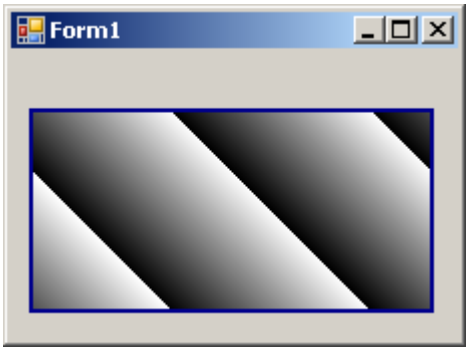


Рис. 2.3 – Закрашивание прямоугольника линейной градиентной кистью

В таблице приведены возможные значения перечисления **LinearGradientMode**.

Константа	Описание
Horizontal	Слева направо
Vertical	Сверху вниз
ForvardDiagonal	По диагонали из верхнего левого угла в нижний правый угол
BackwardDiagonal	По диагонали из верхнего правого угла в нижний левый угол

2.3. Проверка попадания в область изображения

Проверить попадание курсора мыши в какой-то компонент, производный от класса **Control** (например, типа **PictureBox**) очень просто, поскольку такой тип может сам отвечать на события мыши. Но что делать в том случае, когда нужно выполнить проверку в область геометрического шаблона, отображенного на поверхности формы?

Для иллюстрации соответствующего процесса рассмотрим приложение, целью которого является выявление того, что пользователь щелкнул на одном из трех изображений. Выяснив, на каком именно изображении был выполнен щелчок, мы с помощью изменения свойства **Text** формы выделяем это изображение рамкой шириной 5 пикселей. Первым шагом должно быть определение в классе формы новых переменных, представляющих объекты **Rectangle**, для которых будет выполняться регистрация события **MouseDown**. При наступлении такого события нужно программно выяснить, находятся ли поступающие координаты (**x, y**) в рамках границ объектов **Rectangle**, используемых для отображения соответствующих прямоугольных областей. Выяснив, что пользователь щелкнул на изображении, мы должны установить закрытую логическую переменную (**isImageClicked**) равной **true** (истина) и указать, какое изображение было выбрано, используя для этого другую переменную и соответствующее значение из пользовательского перечня **ClickedImage**, определенного следующим образом.

```
enum ClickedImage
```

```
{
    ImageA, ImageB, ImageC
}
```

С учетом вышесказанного, класс формы может выглядеть следующим образом.

```
public partial class Form1 : Form
{
    enum ClickedImage
    {
        ImageA, ImageB, ImageC
    }

    private Rectangle[] imageRects = new Rectangle[3];
    private bool isImageClicked = false;
    ClickedImage imageClicked = ClickedImage.ImageA;

    public Form1()
    {
        InitializeComponent();
        //установка прямоугольников
        imageRects[0] = new Rectangle(30, 30, 80, 80);
        imageRects[1] = new Rectangle(30, 135, 80, 80);
        imageRects[2] = new Rectangle(30, 240, 80, 80);
    }

    private void Form1_MouseDown(object sender,MouseEventArgs e)
    {
        //получение координат (x, y) щелчка
        Point mousePt = new Point(e.X, e.Y);

//проверка попадания указателя в любой из прямоугольников
        if (imageRects[0].Contains(mousePt))
        {
            isImageClicked = true;
            imageClicked = ClickedImage.ImageA;
            this.Text = "Вы щелкнули на изображении A";
        }
        else if (imageRects[1].Contains(mousePt))
        {
            isImageClicked = true;
            imageClicked = ClickedImage.ImageB;
            this.Text = "Вы щелкнули на изображении B";
        }
        else if (imageRects[2].Contains(mousePt))
        {
            isImageClicked = true;
            imageClicked = ClickedImage.ImageC;
            this.Text = "Вы щелкнули на изображении C";
        }
        else //попадения не обнаружено, использовать умолчания
        {
```

```
            isImageClicked = false;
            this.Text = "Проверка попадения в зону изображения";
        }
        Invalidate();
    }
}
```

Обратите внимание на то, что при последней проверке переменная **isImageClicked** устанавливается равной **false** (ложь), поскольку пользователь не выполнил щелчка ни одном из трех изображений. Это важно, если нужно удалить рамку у ранее выделенного изображения. После проверки всех элементов клиентская область обновляется. При этом нужно подключить пространство имен **System.Drawing.Drawing2D**. Обработчик события **Paint** выглядит следующим образом:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Rectangle rect = new Rectangle(10, 10, 50, 50);
    LinearGradientBrush gb1 = new LinearGradientBrush(rect, Col-
or.Yellow, Color.Green, LinearGradientMode.BackwardDiagonal);
    //визуализация изображений
    g.FillRectangle(gb1, imageRects[0]);
    LinearGradientBrush gb2 = new LinearGradientBrush(rect, Col-
or.White, Color.Black, LinearGradientMode.ForwardDiagonal);
    g.FillRectangle(gb2, imageRects[1]);
    LinearGradientBrush gb3 = new LinearGradientBrush(rect, Col-
or.Blue, Color.White, LinearGradientMode.Vertical);
    g.FillRectangle(gb3, imageRects[2]);
    //прорисовка контура (при щелчке в соответствующем месте)
    if (isImageClicked == true)
    {
        Pen outline = new Pen(Color.Tomato, 5);
        switch (imageClicked)
        {
            case ClickedImage.ImageA:
                g.DrawRectangle(outline, imageRects[0]);
                break;
            case ClickedImage.ImageB:
                g.DrawRectangle(outline, imageRects[1]);
                break;
            case ClickedImage.ImageC:
                g.DrawRectangle(outline, imageRects[2]);
                break;
            default:
                break;
        }
    }
}
```

Результат работы данного приложения показан на рисунке 2.4. Несложно убедиться, запустив это приложение, в том, что контур появляется вокруг

каждого изображения, на котором был выполнен щелчок, и что никакого контура не появляется, если щелчок выполнен за пределами этих изображений.

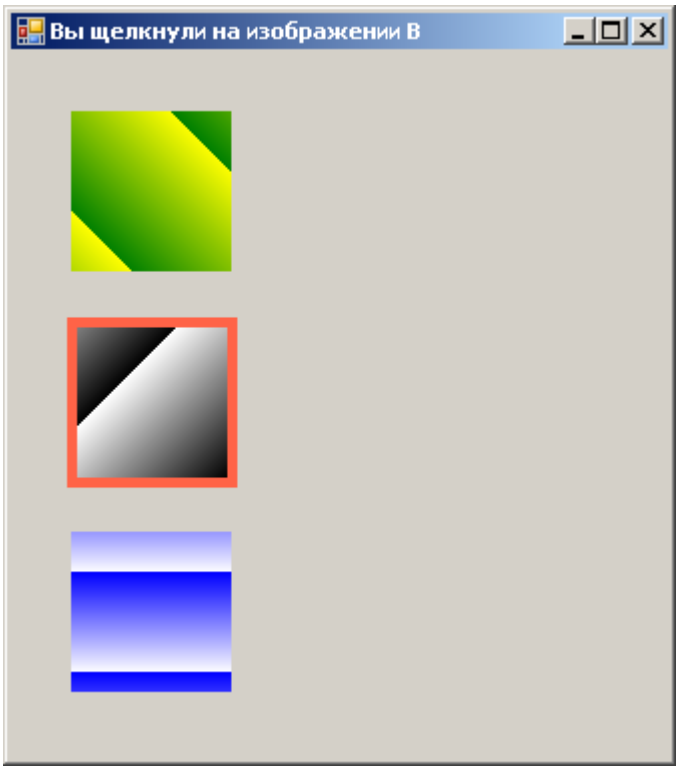


Рис. 2.4 – Проверка попадания в прямоугольник

2.4. Проверка попадания в область, отличную от прямоугольной

Далее нужно выяснить, как выполнить проверку попадания в область, форма которой отличается от прямоугольника? Предположим, что данное приложение обновлено так, что теперь в нем отображается геометрический шаблон неправильной формы, и при щелчке на этом шаблоне его тоже требуется выделить с помощью контура (рис. 2.5).

Этот геометрический образ был создан на форме с помощью метода **`FillPath`** типа **`Graphics`**. Указанный метод получает в качестве параметра экземпляр класса **`GraphicsPath`**, содержащий последовательность соединенных линий, кривых и строк. Добавление новых элементов в объект **`GraphicsPath`** осуществляется с помощью последовательности связанных методов **`Add()`**, как описывается в таблице.

Метод	Описание
`AddArc()`	Добавляет к имеющейся фигуре эллиптическую дугу
`AddBezier()` **`AddBeziers()`**	Добавляет к имеющейся фигуре кубическую кривую Безье (или множество кривых Безье)
`AddClosedCurve()`	Добавляет к имеющейся фигуре замкнутую кривую
`AddCurve()`	Добавляет к имеющейся фигуре кривую
`AddEllipse()`	Добавляет к имеющейся фигуре эллипс
`AddLine()`	Добавляет к имеющейся фигуре сегмент линии

`AddLines()`	
`AddPath()`	Добавляет к имеющейся фигуре указанный `GraphicsPath`
`AddPie()`	Добавляет к имеющейся фигуре сектор круга
`AddPolygon()`	Добавляет к имеющейся фигуре многоугольник
`AddRectangle()` `AddRectangles()`	Добавляет к имеющейся фигуре прямоугольник (или несколько прямоугольников)
`AddString()`	Добавляет к имеющейся фигуре текстовую строку

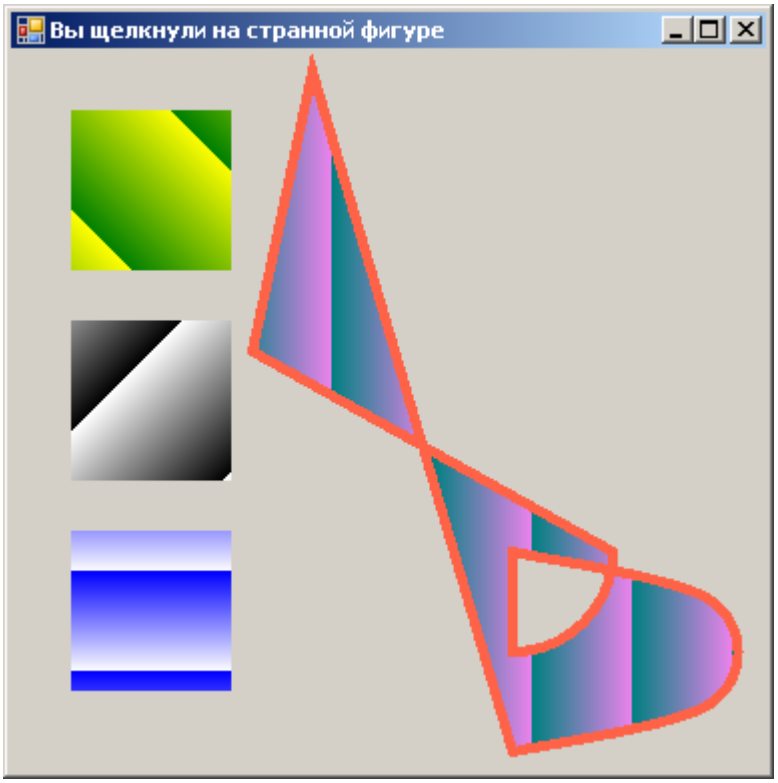


Рис. 2.5 – Проверка попадания в многоугольник

Для реализации этих действий нужно добавить новую переменную типа `GraphicsPath` в класс формы, в конструкторе формы построить множество элементов, представляющих соответствующую траекторию.

```
public partial class Form1 : Form
{
    GraphicsPath myPath = new GraphicsPath();
...
    public Form1()
    {
        myPath.StartFigure();
        myPath.AddLine(new Point(150, 10), new Point(120, 150));
        myPath.AddArc(200, 200, 100, 100, 0, 90);
        Point point1 = new Point(250, 250);
        Point point2 = new Point(350, 275);
        Point point3 = new Point(350, 325);
        Point point4 = new Point(250, 350);
        Point[] points = { point1, point2, point3, point4 };
```

```
        myPath.AddCurve(points);
        myPath.CloseFigure();
...
    }
    ...
}
```

Обратите внимание на вызовы методов **StartFigure()** и **CloseFigure()**. При вызове **StartFigure()** можно вставить новый элемент в строящуюся траекторию. Вызов **CloseFigure()** закрывает имеющуюся фигуру и начинает новую (если это требуется). Также следует знать, что в том случае, когда фигура содержит последовательность соединенных линий и кривых (как в случае с экземпляром **myPath**), цикл завершается путем соединения конечной и начальной точек с помощью линии.

В перечисление **ClickedImage** нужно добавить дополнительное имя **StrangePath**.

```
enum ClickedImage
{
    ImageA, ImageB, ImageC, StrangePath
}
```

Затем обновить имеющийся обработчик события **Form1_MouseDown**, чтобы проверить присутствие указателя мыши в границах **GraphicsPath**. Как и для типа **Region**, это можно сделать с помощью метода **IsVisible()**.

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    ...
    else if(myPath.IsVisible(mousePt))
    {
        isImageClicked=true;
        imageClicked = ClickedImage.StrangePath;
        this.Text = "Вы щелкнули на странной фигуре";
    }
...
}
```

Наконец, обработчик события **Form1_Paint** нужно изменить, как предлагается ниже.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    ...
    LinearGradientBrush gb4 = new LinearGradientBrush(rect, Col-
or.Teal, Color.Violet, LinearGradientMode.Horizontal);
    g.FillPath(gb4, myPath);
    //прорисовка контура (при щелчке в соответствующем месте)
    if (isImageClicked == true)
    {
        Pen outline = new Pen(Color.Tomato, 5);
        switch (imageClicked)
```

```
        {
            ...
            case ClickedImage.StrangePath:
                g.DrawPath(outline, myPath);
                break;
            default:
                break;
        }
    }
}
```

2.5. Определение цветовых значений

Многие методы графического вывода, определенные классом **Graphics**, требуют указания цвета, который должен использоваться в процессе рисования. Структура **System.Drawing.Color** представляет цветовую константу ARGB (от Alpha-Red-Green-Blue – альфа (прозрачность), красный, зеленый, синий). Функциональные возможности типа **Color** (цвет) представляются рядом статических доступных только для чтения свойств, возвращающих конкретный тип **Color**.

```
//Один из множества встроенных цветов
Color c = Color.PapayaWhip;
```

Если стандартные цветные значения не подойдут, можно создать новый тип **Color** и указать для него значения A, R, G и B, используя метод **FromArgb()**.

```
//Указание ARGB вручную.
Color myColor = Color.FromArgb (0, 255, 128, 64);
```

Используя метод **FromName()**, можно также сгенерировать тип **Color** по данному строковому значению. Строковый параметр должен при этом соответствовать одному из элементов перечисления **KnownColor** (который содержит значения для различных цветовых элементов Windows, например, таких как **KnownColor.WindowFrame** и **KnownColor.WindowText**).

```
//Получение Color по известному имени.
Color myColor = Color.FromName ("Red");
```

Независимо от метода получения типа **Color**, с этим типом можно взаимодействовать с помощью его компонентов.

GetBrighness() – возвращает значение яркости типа **Color** на основании измерения HSB (Hue-Saturation-Brightness – оттенок, насыщенность, яркость).

GetSaturation() – возвращает значение насыщенности типа **Color** на основании измерения HSB.

GetHue() – возвращает значение оттенка типа **Color** на основании измерения HSB.

IsSystemColor – индикатор того, что данный тип **Color** является зарегистрированным системным цветом.

A, R, G, B – возвращают значения, присвоенные для прозрачности, красной, зеленой и синей составляющей типа **Color**.

2.6. Стандартное диалоговое окно выбора цвета

Чтобы обеспечить пользователю приложения возможность выбирать цвет

изображения, пространство имен **System.Windows.Forms** предлагает встроенный класс диалогового окна с именем **ColorDialog** (рис. 2.6).

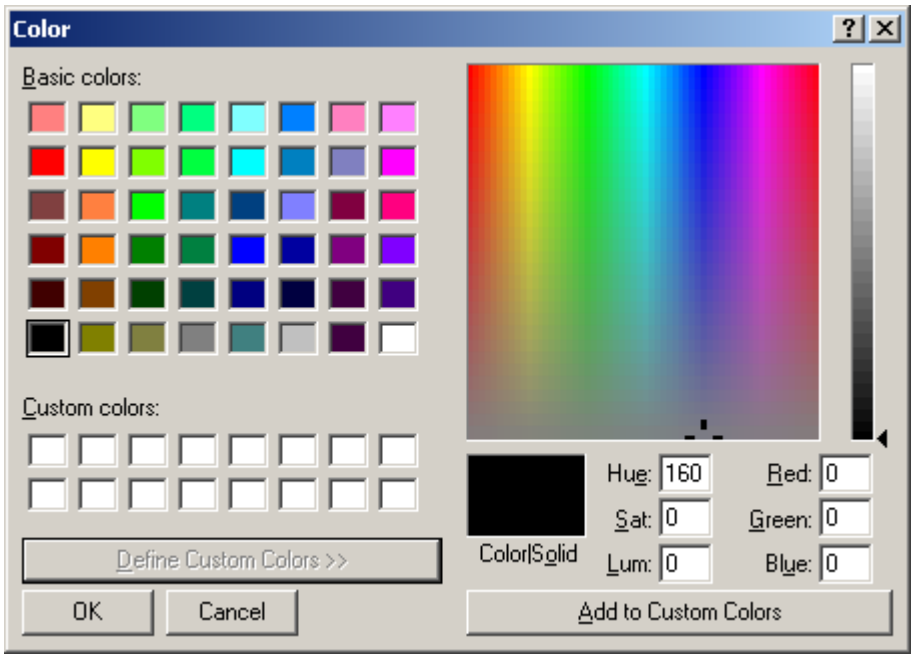


Рис. 2.6 – Стандартное диалоговое окно выбора цвета

Работать с этим диалоговым окном очень просто. Для созданного объекта класса **ColorDialog** нужно вызвать метод **ShowDialog()**, чтобы отобразить окно модально. После закрытия диалогового окна пользователем можно получить выбранный цвет, используя свойство **Color.**

Предположим, что надо с помощью **ColorDialog** предоставить пользователю возможность выбирать цвет фона для области клиента формы. Чтобы упростить ситуацию, будем отображать **ColorDialog** тогда, когда пользователь щелкнет в любом месте клиентской области окна.

```
public partial class MainForm: Form
{
     private ColorDialog colorDlg;
     private Color currColor = Color.DimGray;
public MainForm()
{
          InitializeComponent();
          colorDlg = new ColorDialog();
Text = "Для изменения цвета щелкните здесь";
this.MouseDown + = new MouseEventHandler (Main-
Form_MouseDown);
}
private    void    MainForm_MouseDown(object    sender,
MouseEventArgs e)
{
     if (colorDlg.ShowDialog() != DialogResult.Cancel)
     {
          currColor = colorDlg.Color;
          this.BackColor = currColor;
```

```
            string strARGB = colorDlg.Color.ToString();
            MessageBox.Show(strARGB, "Выбранный цвет:");
        }
}
}
```

3. Пример выполнения работы

Задание. Реализовать эффект движения простейшего геометрического объекта (круга) внутри области рисования с «отскакиванием от стенок». Предусмотреть возможность изменения его цвета и скорости движения. Изменение цвета круга происходит при щелчке в нем левой кнопкой мыши. Скорость регулируется с помощью компонента **`TrackBar`**.

3.1. Визуальное проектирование диалогового окна

Внешний вид главного окна приложения приведен на рисунке 2.7.

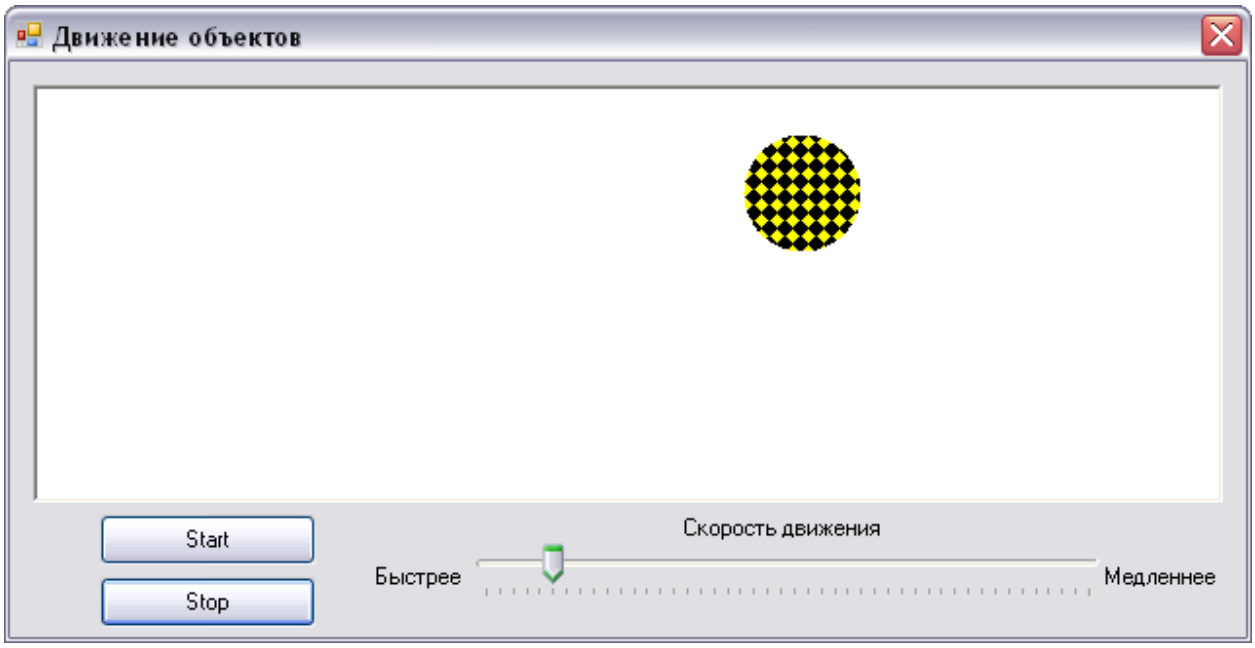


Рис. 2.7. Главное окно приложения

Для графической панели **`pictureBox1`** устанавливаются свойства:

Свойство	Значение	Описание
`BorderStyle`	**`Fixed3D`**	Стиль границы
`BackColor`	**`White`**	Цвет фона

Настройки компонента **`trackBar1`** будут рассматриваться далее.

3.2. Проектирование программного кода

3.2.1. Начальное рисование фигуры

Т.к. в данном приложении будет создаваться кисть со штриховой заливкой, прежде всего, необходимо в файл главной формы подключить пространство имен **`System.Drawing.Drawing2D`**.

Для рисования круга необходимо иметь координаты его центра и радиус. Для этого нужно объявить соответствующие переменные в классе формы:

```
int x0, y0;
int radius;
```

Начальные значения этим переменным можно установить в конструкторе формы или в обработчике события **Load**. Здесь начальное положение круга – левый нижний угол области рисования, радиус – 30 пикселей.

```
x0 = 30;
y0 =  pictureBox1.Height - 35;
radius = 30;
```

Рисование круга будет производиться в обработчике события **Paint** области рисования **pictureBox1**. Процесс рисования состоит из создания объекта **Graphics**, очистки области рисования, создания кисти и непосредственного рисования круга. Кисть создается как объект класса **HatchBrush** (штриховая кисть). При этом первым параметром задается стиль заливки (здесь – **HatchStyle.SolidDiamond** – «шахматка»), вторым параметром – основной цвет заливки (желтый). Можно указать и третий параметр – «неосновной» цвет или цвет фона (по умолчанию – черный). При рисовании круга (метод **FillEllipse()**) необходимо задать кисть для заливки, координаты левого верхнего угла прямоугольника, ограничивающего окружность, а также его стороны. Код обработчика приведен ниже:

```
private void pictureBox1_Paint(object sender, PaintEventArgs e)
{
    //создаем графический объект
    Graphics g = e.Graphics;
    //очищаем область рисования
    g.Clear(Color.White);
    //создаем штриховую кисть желтого цвета
    HatchBrush brush = new HatchBrush(HatchStyle.SolidDiamond,
Color.Yellow);
    //рисуем круг
    g.FillEllipse(brush, x0 - radius, y0 - radius, 2 * radius, 2
* radius);
}
```

3.2.2. Реализация движения фигуры

Реализация движения основана на использовании компонента-таймера (**Timer**). После помещения на форму его свойству **Interval** (интервал в миллисекундах, через который поступает сигнал от таймера) устанавливается значение 10.

По нажатию кнопки «Start» необходимо активизировать («включить») таймер, а кнопки «Stop» – остановить его.

```
//кнопка "Start"
private void button1_Click(object sender, EventArgs e)
{
    timer1.Start();
```

```
}

//кнопка "Stop"
private void button2_Click(object sender, EventArgs e)
{
    timer1.Stop();
}
```

Чтобы заставить фигуру «двигаться», достаточно изменять ее координаты по сигналу таймера, после чего перерисовать фигуру:

```
private void timer1_Tick(object sender, EventArgs e)
{
    //изменяем координаты круга
    x0++;
    y0--;
    //обновляем область рисования (перерисовываем круг)
    pictureBox1.Invalidate();
}
```

В данном случае после нажатия кнопки «Start» круг будет двигаться по диагонали вправо вверх (координата x увеличивается, y – уменьшается), пока не исчезнет за пределами области рисования или пока не будет нажата кнопка «Stop».

3.2.3. Программная реализация «отскакивания от стенок»

Для создания подобного эффекта нужно отслеживать координаты круга с учетом его радиуса. В этом случае удобно создать в классе формы две дополнительные переменные, задающие направление движения по каждой из осей координат:

```
int xDir, yDir;
```

Переменная **xDir** принимает значение 1, если круг двигается вправо, и –1, если влево. Переменная **yDir** равна 1, если идет движение вниз, и –1, если вверх. Поскольку изначально напрвление вижения задается вправо вверх, этим переменным нужно задать соответствующие исходные значения (в конструкторе или в обработчике события **Load**):

```
xDir = 1;
yDir = -1;
```

«Отскакивание от стенок» реализуется за счет изменения значений этих переменных. Если круг «подлетел» к левой стенке, то после этого он должен начать двигаться вправо, т.е. переменной **xDir** нужно присвоить значение 1, и т.д. Приближение к стенкам отслеживается по значениям переменных x0 и y0 с учетом радиуса круга. При этом при «касании» кругом правой и нижней стенок необходимо вносить корректировку в 5 пикселей из-за наличия трехмерной рамки у области рисования.

Таким образом, обработчик события **Tick** для таймера модифицируется следующим образом:

```
private void timer1_Tick(object sender, EventArgs e)
{
    //если круг "подлетел" к левой стенке,
    if (x0 - radius < 0)
        xDir = 1;   //то начинаем двигаться вправо
    //если к правой стенке,
    if (x0 + radius + 5 > pictureBox1.Width)
        xDir = -1;  //то влево
    //если к верхней стенке,
    if (y0 - radius < 0)
        yDir = 1;   //то вниз
    //если к нижней стенке,
    if (y0 + radius + 5 > pictureBox1.Height)
        yDir = -1;  //то вверх
    //изменяем координаты круга
    x0 += xDir;
    y0 += yDir;
    //обновляем область рисования (перерисовываем круг)
    pictureBox1.Invalidate();
}
```

3.2.4. Изменение скорости движения круга

Для изменения скорости движения нужно изменять интервал поступления сигнала от таймера, что, в свою очередь, будет регулироваться компонентом **TrackBar**. Интервал сигнала от таймера в данном приложении задается диапазоном от 5 до 50 миллисекунд. Соответствующие настройки задаются и для компонента **TrackBar**:

Свойство	Значение	Описание
Minimum	**5**	Минимальное значение
Maximum	50	Максимальное значение
Value	**10**	Текущее значение

При прокрутке движка компонента **TrackBar** программе поступает сообщение **Scroll**, в обработчике которого и нужно изменять значение интервала таймера:

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    //изменяем значение интервала таймера на выбранное
    timer1.Interval = trackBar1.Value;
}
```

3.2.5. Создание диалоговой панели выбора цвета заливки

Для выбора цвета заливки круга в данном приложении будет использоваться дополнительное диалоговое окно с группой радиокнопок. Внешний вид этого окна приведен на рисунке 2.8.

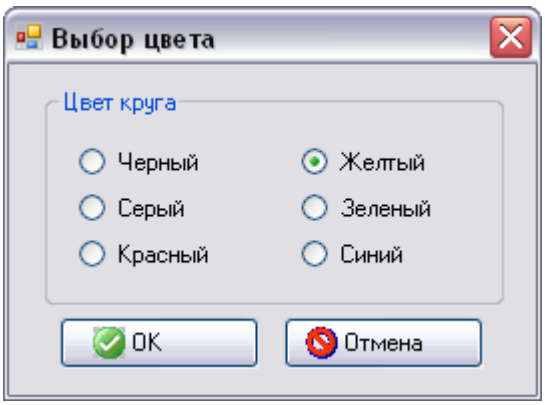


Рис. 2.8 – Диалоговая панель выбора цвета заливки

В этом окне в компоненте **GroupBox** расположены 6 радиокнопок (**RadioButton**)с названиями цветов, а также 2 кнопки: «OK» и «Отмена». На кнопки помимо надписей добавлены соответствующие рисунки.
Рисунки можно добавить на кнопку либо через ее свойство **Image**, либо, как и сделано в данной программе, через компонент **ImageList**. Этот компонент представляет собой список изображений, которые потом можно назначать обычным кнопкам, пунктам меню, кнопкам панели инструментов и другим компонентам. После помещения этого компонента на форму нужно настроить его свойство **Images**, где с помощью отдельного диалогового окна в проект добавляются файлы с изображениями. Каждое добавленное изображение имеет свой индекс (порядковый номер), начиная с 0. В данном приложении рисунок для кнопки «OK» имеет индекс 0, а для кнопки «Отмена» – 1.
Файлы с рисунками и иконками можно найти в папке: C:\Program Files\Microsoft Visual Studio 8\Common 7\VS2005ImageLibrary.
Настройки кнопок «OK» и «Отмена» приведены в таблице:

Свойство	Значение	Описание
ImageList	**imageList1**	Источник изображений
ImageIndex	0 для кнопки «OK», 1 для кнопки «Отмена»	Индекс изображения в источнике
ImageAlign	**MiddleCenter**	Выравнивание изображения
TextAlign	**MiddleCenter**	Выравнивание текста
TextImageRelation	**ImageBeforeText**	Взаимное расположение текста и изображения
DialogResult	**OK** для кнопки «OK», **Cancel** для кнопки «Отмена»	Возвращаемое значение для нажатия кнопки

Установка значения **OK** свойству **DialogResult** для кнопки «OK» означает, что при нажатии на эту кнопку форма закроется и вернет в главное окно значение **DialogResult.OK** (для кнопки «Cancel» – **DialogResult.Cancel**).

3.2.6. Создание свойства для получения цвета

После отображения на экране диалогового окна значение выбранного цвета необходимо передать в главную форму программы. Непосредственно значение выбранной радиокнопки передать не удастся, т.к. переменные, описывающие компоненты формы, закрыты (**private**). Поэтому для решения этой проблемы необходимо создать в классе **Form2** общедоступное свойство для получения выбранного цвета, а также для установки текущего цвета круга (соответствующая радиокнопка становится выбранной). Тип свойства – **Color**. Код свойства приведен далее:

```
public Color CircleColor
{
    get
    {
        //если выбрана первая радиокнопка,
        if (radioButton1.Checked)
            //то вернуть черный цвет
            return Color.Black;
        if (radioButton2.Checked)   //и т.д.
            return Color.Gray;
        if (radioButton3.Checked)
            return Color.Red;
        if (radioButton4.Checked)
            return Color.Yellow;
        if (radioButton5.Checked)
            return Color.Green;
        return Color.Blue;
    }
    set
    {
        //если текущий цвет круга - черный,
        if (value == Color.Black)
            //то сделать выбранной первую радиокнопку
            radioButton1.Checked = true;
        if (value == Color.Gray)    //и т.д.
            radioButton2.Checked = true;
        if (value == Color.Red)
            radioButton3.Checked = true;
        if (value == Color.Yellow)
            radioButton4.Checked = true;
        if (value == Color.Green)
            radioButton5.Checked = true;
        if (value == Color.Blue)
            radioButton6.Checked = true;
    }
}
```

3.2.7. Проверка попадания курсора мыши внутрь круга

Изменение цвета круга в программе должно происходить при щелчке левой кнопкой мыши внутри круга (событие **MouseClick** для компонента **pictureBox1**). Для этого необходимо отслеживать координаты курсора мыши: попал ли он в момент щелчка внутрь круга.

Если область изображения прямоугольная, то такая проверка осуществляется достаточно просто:

```
if (прямоугольник.Contains(точка_курсора))
    //выполнить нужные действия
```

Если же область изображения – не прямоугольная (круг, как в нашем случае, или более сложная фигура), нужно использовать компоненты класса **GraphicsPath** для формирования сложных фигур. Объект этого типа описывается в классе **Form1**

```
GraphicsPath path;
```

и в конструкторе класса формы ему выделяется память:

```
path = new GraphicsPath();
```

Когда на экране рисуется очередной новый круг, его изображение помещается в предварительно очищенный объект **GraphicsPath**. Таким образом, в конец метода **pictureBox1_Paint** надо добавить следующие строки:

```
//очищаем фигуру
path.Reset();
//начинаем формирование фигуры
path.StartFigure();
//добавляем круг в фигуру
path.AddEllipse(x0-radius, y0 - radius, 2 * radius, 2 * radius);
//завершаем формирование фигуры
path.CloseFigure();
```

Если изображение состоит из нескольких элементов (не только из одного круга, как в нашем случае), их все нужно добавить в объект **GraphicsPath.**

3.2.8. Изменение цвета круга

Прежде всего, в класс формы необходимо добавить переменную, задающую цвет круга:

```
Color myColor;
```

Начальное значение (желтый цвет) присваивается ей в конструкторе формы:

```
myColor = Color.Yellow;
```

Изменение цвета круга происходит при щелчке левой кнопки мыши внутри круга (возможно, движущегося) в обработчике события **MouseClick** (не просто **Click**!) компонента **pictureBox1**. Пояснения к коду приведены в комментариях:

```
private void pictureBox1_MouseClick(object sender, MouseEven-
tArgs e)
{
    //проверяем, был ли щелчок именно левой кнопкой мыши
    if (e.Button == MouseButtons.Left)
    {
        //получаем точку с координатами места щелчка
        Point pt = new Point(e.X, e.Y);
        //если эта точка - внутри фигуры (круга)
        if (path.IsVisible(pt))
```

```
        {
            //создаем объект формы выбора цвета
            Form2 form = new Form2();
            //передаем в форму текущий цвет круга
            form.CircleColor = myColor;
            //отображаем форму на экране
            if (form.ShowDialog() == DialogResult.OK)
            {
                //получаем значение выбранного цвета
                myColor = form.CircleColor;
                //перерисовываем круг
                pictureBox1.Invalidate();
            }
        }
    }
}
```

После этого остается лишь указание значение выбранного цвета при создании кисти: в методе **pictureBox1_Paint** строку

```
HatchBrush brush = new HatchBrush(HatchStyle.SolidDiamond, Col-
or.Yellow);
```

заменить на строку

```
HatchBrush brush = new HatchBrush(HatchStyle.SolidDiamond, my-
Color);
```

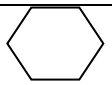
4. Варианты заданий для самостоятельной работы

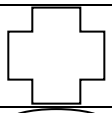
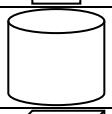
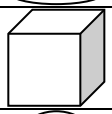
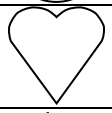
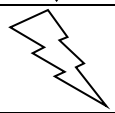
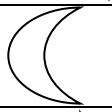
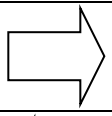
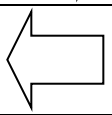
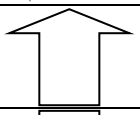
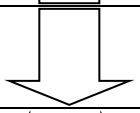
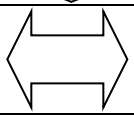
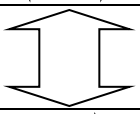
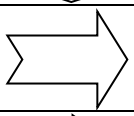
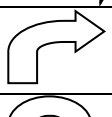
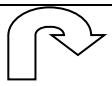
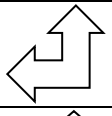
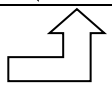
В каждом варианте необходимо реализовать движение требуемого изображения по заданной траектории, не допуская при этом выход фигуры за пределы области рисования. Каждое изображение состоит из нескольких (двух-трех) элементарных фигур. Предусмотреть:

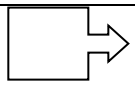
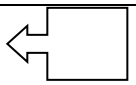
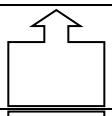
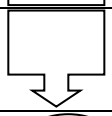
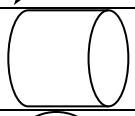
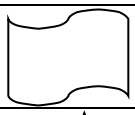
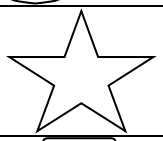
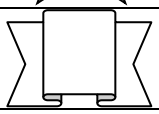
– возможность изменения скорости движения с помощью компонента TrackBar;

– выбор цвета заливки фигуры нажатием левой кнопки мыши внутри изображения. Для выбора используется стандартное диалоговое окно выбора цвета (компонент ColorDialog);

– выбор стиля заливки фигуры нажатием правой кнопки мыши внутри изображения. Для выбора используется пользовательское диалоговое окно с радиокнопками.

№ варианта	Вид изображения	Траектория движения
1.		Вдоль границ области рисования по часовой стрелке

2.		Вдоль границ области рисования против часовой стрелки
3.		От центра области рисования: вверх, вниз, влево, вправо
4.		От центра области рисования: влево, вверх, вправо, вниз. От каждой стенки – к центру
5.		От центра области рисования поочередно к углам. От угла – назад в центр
6.		Вдоль границ области рисования по часовой стрелке
7.		Вдоль границ области рисования против часовой стрелки
8.		От центра области рисования: вверх, вниз, влево, вправо
9.		От центра области рисования: влево, вверх, вправо, вниз. От каждой стенки – к центру
10.		От центра области рисования поочередно к углам. От угла – назад в центр
11.		Вдоль границ области рисования по часовой стрелке
12.		Вдоль границ области рисования против часовой стрелки
13.		От центра области рисования: вверх, вниз, влево, вправо
14.		От центра области рисования: влево, вверх, вправо, вниз. От каждой стенки – к центру
15.		От центра области рисования поочередно к углам. От угла – назад в центр
16.		Вдоль границ области рисования по часовой стрелке
17.		Вдоль границ области рисования против часовой стрелки
18.		От центра области рисования: вверх, вниз, влево, вправо
19.		От центра области рисования: влево, вверх, вправо, вниз. От каждой стенки – к центру
20.		От центра области рисования поочередно к углам. От угла – назад в центр

21.		Вдоль границ области рисования по часовой стрелке
22.		Вдоль границ области рисования против часовой стрелки
23.		От центра области рисования: вверх, вниз, влево, вправо
24.		От центра области рисования: влево, вверх, вправо, вниз. От каждой стенки – к центру
25.		От центра области рисования поочередно к углам. От угла – назад в центр
26.		Вдоль границ области рисования по часовой стрелке
27.		Вдоль границ области рисования против часовой стрелки
28.		От центра области рисования: вверх, вниз, влево, вправо
29.		От центра области рисования: влево, вверх, вправо, вниз. От каждой стенки – к центру
30.		От центра области рисования поочередно к углам. От угла – назад в центр

Лабораторная работа №12. Разработка приложений с многодокументным интерфейсом

1. Цель работы

Изучить теоретические принципы и получить практические навыки разработки программ на основе многооконного (многодокументного) интерфейса.

2. Краткая теория

Аббревиатура MDI расшифровывается как Multiple-document interface – многооконный (многодокументный) интерфейс. То есть MDI-приложения позволяют отображать сразу несколько документов одновременно. При этом каждый документ будет отображаться в своем собственном окне. Обычно MDI-приложения имеют в основном меню подпункты для переключения между окнами и документами.

Механизм работы MDI-приложений немного сложнее, чем обычных приложений, базирующихся на диалогах. Основным окном MDI-приложения является *родительская форма*. Она может содержать несколько *дочерних окон*. Только одно из дочерних окон может быть активно в один момент времени.

2.1. Создание родительской формы

После создания нового проекта (можно использовать уже существующий проект) необходимо выбрать форму, которая будет играть роль главной (родительской) и присвоить ее свойству **`IsMdiContainer`** значение **`true`**.

Родительская форма, как правило, имеет главное меню, в котором предусмотрены пункты для управления дочерними окнами.

2.2. Создание дочерних окон

Любое окно, существующее в приложении, может быть дочерним. Разработчик сам создает *шаблон формы*, которая будет являться дочерним окном.

Для создания шаблона дочернего окна, необходимо добавить в приложение новую форму: Project → Add Windows Form → Windows Form.

Создание дочерних окон в приложении обычно происходит при выборе пункта меню «&Создать». Для этого необходимо создать обработчик этого пункта меню – обработчик события **`Click`**.

Код обработчика:

```
Form2 newMDIChild = new Form2();
newMDIChild.MdiParent = this;
newMDIChild.Show();
```

В функции создается экземпляр класса **`Form2`** с именем **`newMDIChild`**.

Объект **newMDIChild** – это обычное окно. Для того чтобы создаваемое окно отображалось как дочерняя форма приложения, необходимо установить его свойство **MdiParent** равным **this**. Таким образом указывается, что родительской формой создаваемого окна является главная форма приложения. Метод **Show()** позволяет отобразить форму на экране.

Ссылка на текущее дочернее окно может быть получена с помощью свойства **ActiveMdiChild** родительской формы.

3. Пример создания MDI-приложения

Для создания тестового примера необходимо выполнить следующие пошаговые инструкции:

1) Создайте новый проект.

2) Проект содержит только одну форму **Form1**. Сделайте ее главной (родительской).

3) Установите свойству **WindowState** формы **Form1** значение **Maximized**. При запуске программы главное окно будет принимать максимальный размер (раскрываться на весь экран). Аналогичного эффекта можно добиться, если в конструкторе формы **Form1** дописать следующий код:

```
this.WindowState=FormWindowState.Maximized;
```

4) Создайте главное меню.

5) Добавьте пункты меню «&Файл» и «&Окно». Измените их свойство **name** на **menuItemFile** и **menuItemWindow** соответственно.

6) Добавьте в меню «&Файл» пункт «&Создать». Свойство **name** установите в **menuItemNew**. Этот пункт меню будет предназначен для создания дочерних окон.

7) Добавьте в проект еще одну форму, которая будет шаблоном дочерней формы.

8) Создайте обработчик события **Click** для пункта меню «&Создать» и добавьте код из п.4.2. краткой теории лабораторной работы.

9) Добавьте в меню «&Окно» пункты «&Горизонтально», «&Вертикально» и «&Каскадом». Свойство **name** установите в **menuItemTileHorizontal**, **menuItemTileVertical** и **menuItemCascade** соответственно. Эти пункты меню предназначены для вариантов упорядочения дочерних окон.

10) Выберите элемент главного меню и установите его свойству **MdiWindowListItem** значение **menuItemWindow**. Это означает, что в пункте меню «&Окно» будет отображаться список всех открытых дочерних окон.

11) Создайте один общий обработчик для подпунктов меню пункта «&Окно». Для этого выделите все три пункта меню, удерживая клавишу Ctrl, и щелкните два раза указателем мыши по событию **Click** в окне свойств. Замените имя события (свойство **name**), созданное по умолчанию, на имя **menuItemTool_Click**. Функция **menuItemTool_Click** будет устанавливать вариант упорядочения дочерних окон.

12) Добавьте следующий код в функцию **menuItemTool_Click**:

```
ToolStripMenuItem item = sender as ToolStripMenuItem;
switch (item.Text)
{
    case "&Горизонтально":
        this.LayoutMdi(MdiLayout.TileHorizontal);
        break;
    case "&Вертикально":
        this.LayoutMdi(MdiLayout.TileVertical);
        break;
    case "Каскадом":
         this.LayoutMdi(MdiLayout.Cascade);
         break;
}
```

13) Запустите программу. Выберите из меню пункт «Файл»/ «Создать». На экране появится дочернее окно с именем «**Form2**». Вы можете создать еще несколько дочерних окон, воспользовавшись все тем же пунктом меню. При этом пункт меню «Окно» будет содержать список всех дочерних окон, открытых на текущий момент.

14) Попробуйте различные варианты упорядочения дочерних окон, воспользовавшись пунктом меню «Окно».

15) Заголовки всех дочерних окон одинаковые – «**Form2**». Модифицируем приложение таким образом, чтобы при создании нового дочернего окна в заголовке появлялся порядковый номер этого окна. Для этого необходимо изменить код обработчика события **Click** пункта меню «&Создать» следующим образом:

```
Form2 newMDIChild = new Form2();
newMDIChild.MdiParent = this;

int wnd_num = this.MdiChildren.Length-1;
newMDIChild.Text = "Doc"+wnd_num.ToString();

newMDIChild.Show();
```

Свойство **MdiChildren** содержит массив дочерних форм.

16) Запустите программу. Создайте несколько дочерних окон. Посмотрите как изменяются заголовки.

17) Добавьте на форму панель инструментов (компонент **ToolStrip**) с кнопками, соответствующими пунктам меню. Рисунки на кнопки можно добавить через свойство **Image.**

18) В качестве обработчика события Click каждой кнопки панели инструментов укажите функцию-обработчик соответствующего пункта меню. После этого реакция на выбор пункта меню или нажатие соответствующей кнопки панели инструментов будет одинаковой.

4. Варианты заданий для самостоятельного решения

В каждом варианте в главном окне необходимо создать меню и панель инструментов (компонент ToolStrip) с кнопками, дублирующими пункты меню.

1. Дан список фамилий (загружается из файла при создании дочернего окна). Организовать отбор во всех списках (во всех дочерних окнах) по первой букве фамилий. Результаты отображать в виде списка в отдельном окне (не дочернем).

2. Создать простейший текстовый редактор с возможностью открытия и сохранения файлов.

3. Создайте приложение «Секундомер». При создании дочернего окна должен запускаться секундомер. В окне должна быть кнопка останова, паузы и нового запуска.

4. Создать простейшее приложение для просмотра графических файлов с возможностью открытия и сохранения изображений.

5. Создать следующее приложение. В каждом дочернем окне – текстовая метка или поле ввода, которые работают как числовой счетчик. Скорость увеличения задается таймером. Скорость таймера можно менять одновременно для всех дочерних окон в отдельном диалоговом окне (не дочернем).

6. Создать приложение с двумя видами шаблонов дочерних окон. В первом случае должен осуществляться контроль ввода только цифровых данных, во втором – только текстовых. Заголовок должен содержать слово «Цифры» или «Символы».

7. Создайте приложение «Конверт». Шаблон дочернего окна должен представлять собой образец заполнения полей на почтовом конверте: в левом верхнем углу адрес отправителя, внизу справа адрес получателя. В заголовке окна должен отображаться получатель.

8. Создайте приложение «Статистика текста». При наборе текста с клавиатуры должен проводиться подсчет и осуществляться вывод количества знаков препинания, букв и цифр.

9. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения цвета линии этого объекта.

10. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения цвета линии этого объекта с помощью задания интенсивности красного, зеленого и синего цветов (модель RGB) через компоненты **`TrackBar`.**

11. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения стиля линии этого объекта (для этого у пера нужно изменять свойство **`DashStyle`**).

12. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения толщины линии этого объекта.

13. Создать приложение, где в каждом дочернем окне изображается ка-

кой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения цвета заливки этого объекта.

14. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения цвета заливки этого объекта с помощью задания интенсивности красного, зеленого и синего цветов (модель RGB) через компоненты **TrackBar.**

15. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения стиля заливки этого объекта.

16. Создать приложение, где в каждом дочернем окне изображается движущийся графический объект (круг, прямоугольник и т.п.).

17. Разработать приложение для ввода результатов сессии. Организовать табличный ввод, используя компонент **DataGridView**. Предусмотреть запись результатов в текстовый файл.

18. Разработать приложение «Табулирование функций». Программа позволяет получить значения аргумента и функции в заданном интервале с заданным шагом. Вид функции можно задать выбором в компоненте **ListBox** или в компонентах **RadioButton.**

19. Разработать приложение, где в каждом дочернем окне должен вводиться пароль. Если пароль введен верно, то отображается какое-нибудь графическое изображение.

20. Разработать следующее приложение. В каждом дочернем окне – компонент-счетчик. В главном меню есть пункт, по нажатию которого открывается диалоговое окно, где отображается сумма чисел из счетчиков во всех открытых дочерних окнах.

21. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения координат этого объекта.

22. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность изменения размера этого объекта.

23. Создать приложение, где в каждом дочернем окне изображается какой-то графический объект (круг, прямоугольник и т.п.). Предусмотреть возможность поворота этого объекта (влево на 90°, вправо на 90°, на 180°).

24. Создать простейший текстовый редактор с возможностью изменения шрифта и цвета текста.

25. Разработать следующее приложение. В каждом дочернем окне выбирается дата с помощью компонента **DateTimePicker**. В главном меню есть пункт, по нажатию которого открывается диалоговое окно, где выводится самая ранняя из всех дат, заданных в открытых дочерних окнах.

26. Разработать следующее приложение. В диалоговом окне (не дочернем) вводится текстовая строка. Программа должна разбить эту строку на слова и каждое слово передать в отдельное дочернее окно, например, в ком-

понент **Label** или **TextBox** (должно открыться столько окон, сколько слов в строке).

27. Разработать следующее приложение. В диалоговом окне (не дочернем) вводится матрица чисел (в компонент **DataGridView**). Программа должна разбить эту матрицу на строки и каждую строку отобразить в отдельном дочернем окне (должно открыться столько окон, сколько строк в матрице).

28. Разработать следующее приложение. В диалоговом окне (не дочернем) вводится матрица чисел (в компонент **DataGridView**). Программа должна разбить эту матрицу на столбцы и каждый столбец отобразить в отдельном дочернем окне (должно открыться столько окон, сколько столбцов в матрице).

29. Разработать следующее приложение. В каждом дочернем окне вводится одно и то же количество чисел (например, в компонент **DataGridView** или **ListBox**). Необходимо все введенные числа из каждого дочернего окна объединить по строкам в матрицу, матрицу отобразить в отдельном диалоговом окне (не дочернем).

30. Разработать следующее приложение. В каждом дочернем окне вводится одно и то же количество чисел (например, в компонент **DataGridView** или **ListBox**). Необходимо все введенные числа из каждого дочернего окна объединить по столбцам в матрицу, матрицу отобразить в отдельном диалоговом окне (не дочернем).

Лабораторная работа №13. Разработка многопоточных приложений

1. Цель работы

Изучить методы и средства, пространства имен и классы, применяемые при работе с потоками в среде .NET Framework. Получить практические навыки разработки многопоточных приложений на языке C#.

2. Сведения из теории

2.1. Общие сведения по работе с потоками

Многопоточная программа состоит из двух или более частей, которые могут выполняться одновременно. Каждая часть такой программы называется *потоком*, и каждый поток определяет собственный путь выполнения инструкций. Таким образом, многопоточность представляет собой специальную форму многозадачности.

Различают два вида многозадачности: с ориентацией на процессы и с ориентацией на потоки. *Процесс* по сути представляет собой выполняемую программу. Следовательно, многозадачность, ориентированная на процессы, – это средство, позволяющее компьютеру выполнять две или больше программ одновременно.

Поток – это управляемая единица выполняемого кода. В многозадачной среде, ориентированной на потоки, все процессы имеют по крайней мере один поток, но возможно и большее их количество. Это означает, что одна программа может выполнять сразу две или более задач.

Поток может находиться в одном из нескольких состояний. Он может *выполняться*. Он может быть *готовым к выполнению* (как только получит время ЦП). Выполняющийся поток может быть *приостановлен*, т.е. его выполнение временно прекращается. Позже оно может быть *возобновлено*. Поток может быть *заблокирован* в ожидании необходимого ресурса. Наконец, поток может *завершиться*, и уж в этом случае его выполнение окончено и продолжению (возобновлению) не подлежит.

В среде .NET Framework определено два типа потоков: *высокоприоритетный* (foreground) и *низкоприоритетный*, или *фоновый* (background). По умолчанию поток создается высокоприоритетным, но его тип можно изменить, т.е. сделать фоновым. Единственное различие между высоко- и низкоприоритетным потоками состоит в том, что последний будет автоматически завершен, если все высокоприоритетные потоки в его процессе остановились.

Все процессы имеют по крайней мере один поток управления, который обычно называется *основным*, поскольку именно с этого потока начинается выполнение программы. Из основного можно создать и другие потоки. Желательно (но не обязательно), чтобы в многопоточной программе основной поток выполнялся последним.

Классы, которые поддерживают многопоточное программирование,

определены в пространстве имен **System.Threading.**

Многопоточная система C# встроена в класс **Thread**, который инкапсулирует поток управления. Класс Thread является **sealed**-классом, т.е. он не может иметь наследников. В классе **Thread** определен ряд методов и свойств для управления потоками.

Чтобы создать поток, необходимо создать объект типа **Thread**. В классе **Thread** определен следующий конструктор:

```
public Thread(ThreadStart entryPoint)
```

Здесь параметр **entryPoint** содержит имя метода, который будет вызван, чтобы начать выполнение потока. Тип **ThreadStart** – это делегат (ссылка на метод), определенный в среде .NET Framework:

```
public delegate void ThreadStart()
```

Итак, начальный метод должен иметь тип возвращаемого значения **void** и не принимать никаких аргументов.

Выполнение созданного потока не начнется до тех пор, пока не будет вызван метод **Start()**. Начавшись, выполнение потока будет продолжаться до тех пор, пока не завершится метод, заданный параметром **entryPoint.** Поэтому после выхода из этого метода выполнение потока автоматически завершится. Если попытаться вызвать метод **Start()** для потока, запущенного на выполнение, будет сгенерировано исключение типа **ThreadStateException.**

В классе **Thread** предусмотрено значительное число компонентов (как статических, так и обычных) для создания текущим потоком новых потоков, а также для приостановки, полной остановки и удаления указанного потока. Наиболее важные статические компоненты представлены в таблице:

Статический компонент	Назначение
CurrentThread	Это свойство только для чтения возвращает ссылку на поток, выполняемый в настоящее время
Sleep()	Приостанавливает выполнение текущего потока на указанное пользователем время

Обычные (нестатические) компоненты класса **Thread** приведены в следующей таблице:

Компонент	Назначение
IsAlive	Возвращает **true** или **false** в зависимости от того, запущен поток или нет
IsBackground	Определяет, является ли поток фоновым
Name	Текстовое имя потока
Priority	Позволяет получить/установить приоритет потока (используются значения из перечисления **ThreadPriority**)
ThreadState	Возвращает информацию о состоянии потока (используют-

	ся значения из перечисления **ThreadState**)
Interrupt()	Прерывает работу текущего потока
Join()	Ждет появления другого потока (или указанный промежуток времени) и завершается
Resume()	Продолжает работу после приостановки работы потока
Start()	Начинает выполнение потока, определенного делегатом **ThreadStart**
Suspend()	Приостанавливает выполнение потока. Если выполнение потока уже приостановлено, то игнорируется

2.2. Синхронизация потоков

При использовании в программе нескольких потоков иногда необходимо координировать их выполнение. Процесс координации потоков называется *синхронизацией*. К синхронизации прибегают в тех случаях, когда двум или большему числу потоков необходимо получить к общему ресурсу, который в каждый момент времени может использовать только один поток. Например, когда один поток выводит данные в файл, в это самое время второй файл должен быть лишен возможности выполнять аналогичные действия. Синхронизация необходима и в других ситуациях. Например, один поток ожидает, пока не произойдет событие, «судьба» которого зависит от другого потока. В этом случае необходимо иметь средство, которое бы удерживало первый поток в состоянии приостановки до тех пор, пока не произойдет ожидаемое им событие. После этого «спящий» поток должен возобновить выполнение.

В основе синхронизации лежит понятие *блокировки*, т.е. управление доступом к некоторому блоку кода в объекте. На то время, когда объект заблокирован одним потоком, никакой другой поток не может получить доступ к заблокированному объекту. Когда поток снимет блокировку, объект станет доступным для использования другим потоком.

Синхронизация поддерживается ключевым словом **lock**. Формат использования:

```
lock(object)
{
  //инструкции, подлежащие синхронизации
}
```

Здесь параметр **object** представляет собой ссылку на синхронизируемый объект. Инструкция **lock** гарантирует, что указанный блок кода, защищенный блокировкой для данного объекта, может быть использован только потоком, который получает эту блокировку. Все другие потоки остаются заблокированными до тех пор, пока блокировка не будет снята. А снята она будет лишь при выходе из этого блока.

2.3. Взаимодействие потоков

Рассмотрим следующую ситуацию. Поток (назовем его T) выполняет содержимое **lock**-блока и требует доступа к ресурсу (назовем его R), который временно недоступен. Что делать потоку T? Если поток T войдет в цикл

опроса в ожидании доступности ресурса R, он свяжет объект, блокируя доступ к нему другим потокам. Это решение трудно назвать оптимальным, поскольку оно аннулирует преимущества программирования в многопоточной среде. Будет лучше, если поток T временно откажется от «претензий» на объект, позволив другому потоку выполнить свою работу. Когда же ресурс R станет доступным, поток T можно уведомить об этом, и он возобновит выполнение. Такой подход опирается на межпоточные средства общения, которые позволяют одному потоку уведомить другой о том, что он блокируется, а затем первого поставить в известность о том, что он может возобновить выполнение. C# поддерживает межпоточное взаимодействие с помощью методов **`Wait()`**, **`Pulse()`** и **`PulseAll()`**, определенных в классе **`Monitor`**. Эти методы можно вызывать только внутри **`lock`**-блока.

Когда выполнение потока временно блокируется, вызывается метод **`Wait()`**, т.е. он переходит в режим ожидания («засыпает») и снимает блокировку с объекта, позволяя другому потоку использовать этот объект. Позже, когда другой поток входит в аналогичное состояние блокирования и вызывает метод **`Pulse()`** или **`PulseAll()`**, «спящий» поток «просыпается». Обращение к методу **`Pulse()`** возобновляет выполнение потока, стоящего первым в очереди потоков, пребывающих в режиме ожидания. Обращение к методу **`PulseAll()`** сообщает о снятии блокировки всем ожидающим потокам.

При разработке многопоточных программ необходимо позаботиться о том, чтобы во время их выполнения не создалась тупиковая ситуация, вызванная взаимоблокировкой. При *взаимоблокировке* один поток ожидает, пока другой не выполнит некоторое действие, но в то же время второй поток ожидает действия первого. Таким образом, оба потока приостановлены, ожидая друг друга, и ни один из них не выполняется. Как правило, если многопоточная программа вдруг «виснет», то наиболее вероятная причина этого – взаимоблокировка.

3. Пример выполнения работы

Задание. Необходимо разработать многопоточное приложение с синхронизацией двух потоков и управлением доступом к общему ресурсу. В каждом потоке осуществляется движение графического объекта (в данном случае, круга) от верхнего края формы к нижнему, после чего потоки завершаются. В качестве общего ресурса выступает прямоугольная область в середине формы. Необходимо скоординировать потоки таким образом, чтобы в каждый момент времени внутри этой области находился только один круг.

Внешний вид работающего приложения приведен на рисунке 8.1.

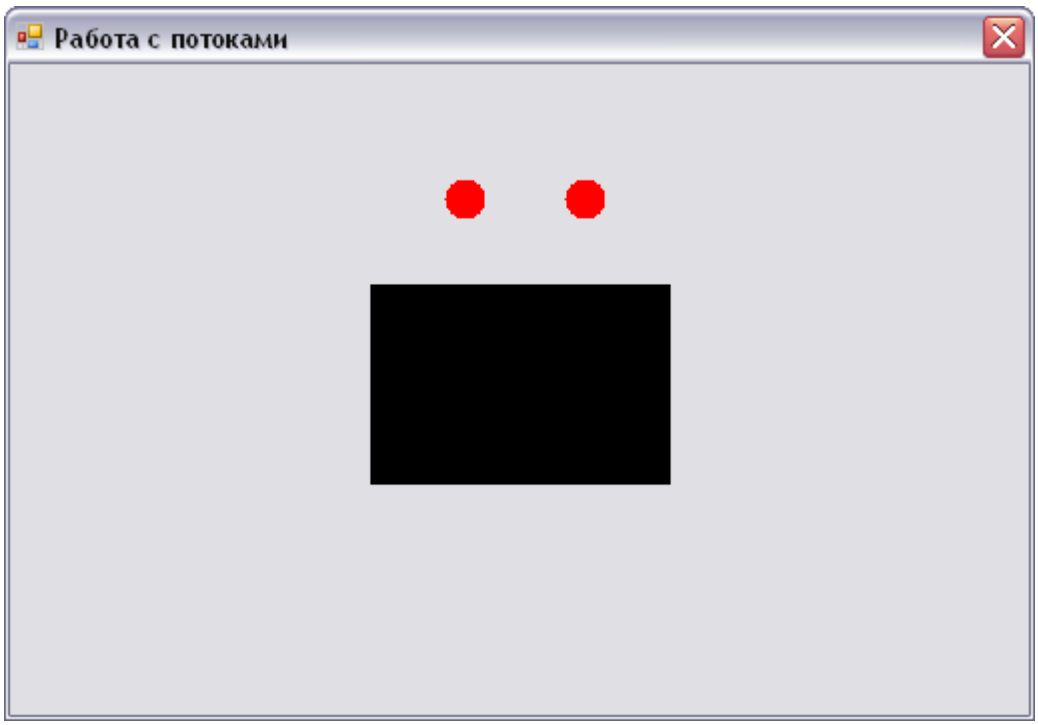


Рис. 6.1. Пример работающего приложения

3.1. Создание потоков

Прежде всего, для работы с потоками необходимо подключить пространство имен **`System.Threading`**.

Для того чтобы создавать в приложении новые потоки, необходимо задать метод, который будет являться точкой входа в поток. Этот метод должен быть типа **`void`** и без параметров. В нашем случае именно в этом методе (который здесь назван **`MoveCircle()`**) и будет осуществляться движение круга.

В этом методе сначала создается контекст графического устройства и две кисти: одна (красного цвета) для круга, вторая (цвета формы) – для стирания предыдущего изображения круга.

Чтобы два круга в разных потоках не сливались друг с другом, для них указываются разные координаты **`x`**. Значения координаты устанавливаются в зависимости от имени потока (свойство **`Name`**).

После этого в цикле, пока не достигнут нижний край формы, рисуется очередной круг, поток на некоторое время приостанавливается (иначе движение будет происходить настолько быстро, что увидеть его просто не получится), а затем круг стирается.

После окончания цикла для корректности выводится сообщение о завершении текущего потока. Листинг метода приведен ниже:

```
private void MoveCircle()
{
    //создаем контекст устройства
    Graphics g = this.CreateGraphics();
    //создаем красную кисть - для круга
    Brush b1 = Brushes.Red;
    //и кисть цвета формы - для стирания круга
```

```
    Brush b2 = SystemBrushes.Control;
    int x;
    //координата x круга будет зависеть от имени потока
    if (Thread.CurrentThread.Name == "First")
        x = Width / 2 - 30;
    else
        x = Width / 2 + 30;
    //цикл от верхнего до нижнего края формы
    for (int y = 10; y < Height - 40; y++)
    {
        //рисуем круг
        g.FillEllipse(b1, x - 10, y - 10, 20, 20);
        //"усыпляем" поток на 30 миллисекунд
        Thread.Sleep(30);
        //стираем круг
        g.FillEllipse(b2, x - 10, y - 10, 20, 20);
    }
    //проверяем корректность завершения потока
    MessageBox.Show("Поток " + Thread.CurrentThread.Name + "
                    завершен!");
}
```

Непосредственное создание потоков можно выполнять в обработчике события **Load** главной формы. При этом для каждого потока нужно создать объект класса **Thread**, указав при этом имя метода, который будет точкой входа в поток. Далее следует присвоить потоку имя и запустить его с помощью метода **Start()**. Код обработчика приведен ниже:

```
private void Form1_Load(object sender, EventArgs e)
{
    //создаем первый поток
    //(точка входа в него - метод MoveCircle())
    Thread thread1 = new Thread(new ThreadStart(MoveCircle));
    //присваиваем потоку имя
    thread1.Name = "First";
    //аналогично для второго потока
    Thread thread2 = new Thread(new ThreadStart(MoveCircle));
    thread2.Name = "Second";
    //запускаем оба потока
    thread1.Start();
    thread2.Start();
}
```

3.2. Работа с прямоугольной областью

Для задания координат и размера прямоугольника необходимо добавить в класс формы соответствующую переменную типа **Rectangle**:

```
Rectangle rect = new Rectangle(180, 110, 150, 100);
```

Рисование прямоугольника производится в обработчике события **Paint**

формы:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    //получаем контекст устройства
    Graphics g = e.Graphics;
    //создаем черную кисть
    Brush b = Brushes.Black;
    //рисуем прямоугольник
    g.FillRectangle(b, rect);
}
```

Если на этом этапе запустить программу, то можно увидеть, как круги, проходя сквозь прямоугольник, «стирают» на своем пути часть его. Это происходит из-за того, что при реализации движения круг «стирается» путем заливки его цветом формы. Устранить данный эффект можно путем перерисовки прямоугольника при попадании круга внутрь него. Таким образом, в методе **MoveCircle()** после строчки **Thread.Sleep(30);** следует добавить следующее:

```
//если круг или его часть - внутри прямоугольника,
if (y + 10 > rect.Y && y - 10 < rect.Y + rect.Height)
    //то перерисовываем прямоугольник
    Invalidate();
```

3.3. Синхронизация потоков

Так как в данной программе необходимо обеспечить управление доступом потоков (кругов) к общему ресурсу (прямоугольнику), то, прежде всего, нужно обеспечить блокировку объекта с помощью метода **lock()**. Эта блокировка должна действовать, только если один из кругов находится внутри прямоугольника, иначе блокировку нужно снять (метод **Wait()** класса **Monitor**) и уведомить об этом второй поток (метод **Pulse()** класса **Monitor**).

Также для корректного завершения работы потоков необходимо вызвать метод **Pulse()** при завершении работы потока.

Таким образом, окончательный вариант метода **MoveCircle()** будет выглядеть следующим образом:

```
private void MoveCircle()
{
    //устанавливаем блокировку метода
    lock (this)
    {
        //создаем контекст устройства
        Graphics g = this.CreateGraphics();
        //создаем красную кисть - для круга
        Brush b1 = Brushes.Red;
        //и кисть цвета формы - для стирания круга
        Brush b2 = SystemBrushes.Control;
```

```
            int x;
            //координата x круга будет зависеть от имени потока
            if (Thread.CurrentThread.Name == "First")
                x = Width / 2 - 30;
            else
                x = Width / 2 + 30;
            //цикл от верхнего до нижнего края формы
            for (int y = 10; y < Height - 40; y++)
            {
                //рисуем круг
                g.FillEllipse(b1, x - 10, y - 10, 20, 20);
                //"усыпляем" поток на 30 миллисекунд
                Thread.Sleep(30);
                //если круг или его часть - внутри прямоугольника,
                if (y+10 > rect.Y && y - 10 < rect.Y + rect.Height)
                    //то перерисовываем прямоугольник
                    Invalidate(rect);
                else
                {
                    //разрешаем выполнение другого потока
                    Monitor.Pulse(this);
                    //ждем приостановки/завершения другого потока
                    Monitor.Wait(this);
                }
                //стираем круг
                g.FillEllipse(b2, x - 10, y - 10, 20, 20);
            }
            //выводим поток из режима ожидания
            Monitor.Pulse(this);
        }
        //проверяем корректность завершения потока
        MessageBox.Show("Поток " + Thread.CurrentThread.Name + "
                        завершен!");
}
```

4. Задание для самостоятельной работы

Модифицировать данную программу следующим образом:

– приложение должно быть основано на многодокументном интерфейсе;

– вся работа с потоками (движение круга) должна производиться в дочернем окне;

– увеличить количество потоков в каждом дочернем окне до трех;

– в каждом дочернем окне должно быть два прямоугольника, круги (потоки) должны «проходить сквозь них» последовательно;

– круги должны двигаться горизонтально.

Лабораторная работа №14 Разработка сетевых приложений

1. Цель работы

Изучить основы работы компонентов, связанных с обработкой данных, полученных после использования HTTP-запросов и создать приложение, иллюстрирующее работу этих компонентов и возможности получения информации.

2. Сведения из теории

2.1 Начальные знания о протоколе TCP/IP

Протокол определяет правила, используемые для передачи информации по сети. Стеком протоколов TCP/IP называют набор сетевых протоколов, используемых в интернет. В этом стеке различают несколько уровней, и протоколы высокого уровня всегда базируются на протоколах более низких уровней. В самом низу находятся физический уровень и канальный уровень. Пример протокола — Ethernet, описывающий передачу данных по коаксиальному кабелю или витой паре. Протоколы этих уровней обычно реализуются на уровне железа, например в сетевой карте компьютера.

Выше идёт сетевой уровень, где находится протокол IP, описывающий структуру сети и доставку пакетов.

Ещё выше — транспортный уровень, где находится протокол TCP, использующийся для передачи данных. Эти протоколы обычно реализуются на уровне операционной системы.

На самом верху находится множество протоколов прикладного уровня, выполняющих конкретные прикладные задачи. Обычно они программируются в отдельных приложениях.

2.1.1 Определение IP

IP — протокол, лежащий в основе интернета, его название так и расшифровывается: Internet Protocol. Согласно протоколу, каждый узел в сети имеет свой IP адрес, состоящий из 4х байт и обычно записываемый как *n.n.n.n*

Каждый узел напрямую «видит» только узлы в своей подсети, с «похожими» адресами (с помощью маски подсети). А другим узлам он передает пакеты через промежуточные узлы — Маршрутизаторы.

2.1.2 Определение TCP

TCP протокол базируется на IP для доставки пакетов, но добавляет две важные вещи:

- установление соединения — это позволяет ему, в отличие от IP, гарантировать доставку пакетов;
- порты — для обмена пакетами между приложениями, а не просто узлами

2.1.3 TCP-соединение

Соединение начинается с handshake (рукопожатия):
Узел *A* посылает узлу *B* специальный пакет SYN — приглашение к соединению. *B* отвечает пакетом SYN-ACK — согласием об установлении соединения. *A* посылает пакет ACK — подтверждение, что согласие получено. После этого TCP соединение считается установленным, и приложения, работающие в этих узлах, могут посылать друг другу пакеты с данными. «Соединение» означает, что узлы помнят друг о друге, нумеруют все пакеты, идущие в обе стороны, посылают подтверждения о получении каждого пакета и перепосылают потерявшиеся по дороге пакеты. Для *узла A* это соединение называется исходящим, а для *узла B* — входящим. Эти термины показывают инициатора соединения, то есть направление самого первого пакета (SYN). Любое установленное TCP соединение симметрично, и пакеты с данными по нему всегда идут в обе стороны.

2.1.4 Порт

Сетевой порт — условное число от 1 до 65535, указывающее, какому приложению предназначается пакет.

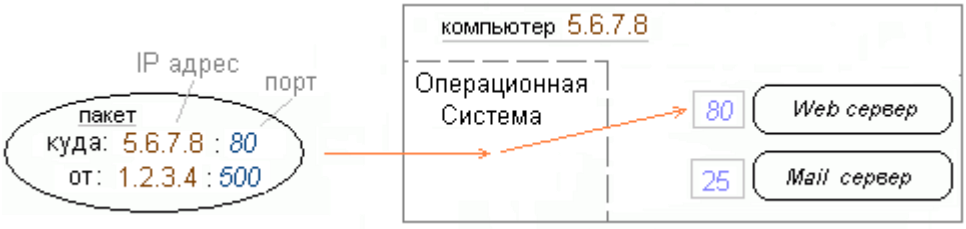


Рисунок 7.1. Пример обращения на порт компьютера

Согласно IP, в каждом пакете присутствуют IP адрес узла-источника и IP адрес узла-назначения. В TCP пакетах дополнительно указываются порт источника и порт назначения. Узел назначения, получив пакет, смотрит на порт назначения и передает пакет соответствующему у себя приложению. Использование портов позволяет независимо использовать TCP протокол сразу многим приложениям на одном и том же компьютере.

2.1.5 Использование портов

Клиентом называют приложение, которое пользуется каким-то сервисом, предоставляемым другим приложением — Сервером, обычно на удаленном компьютере. Практически всегда клиент начинает исходящие соединения, а сервер ожидает входящих соединений (от клиентов), хотя бывают и исключения. Сервер при запуске сообщает операционной системе, что хотел бы "занять" определенный порт (или несколько портов). После этого все пакеты, приходящие на компьютер к этому порту, ОС будет передавать этому серверу. Говорят, что сервер "слушает" этот порт. Клиент, начиная соединение, выпрашивает у своей ОС какой-нибудь незанятый порт во временное пользование, и указывает его в посланных пакетах как порт источника. Затем на этот порт он получит ответные пакеты от сервера.

Таким образом, сервер:

- слушает на определённом порту, заранее известном клиенту;
- занимает этот порт всё время, пока не завершит работу;
- об IP адресе и номере порта клиента узнаёт из приглашения, посланного клиентом.

Клиент:

- заранее знает IP адрес и порт сервера;
- выбирает у себя произвольный порт, который освобождает после окончания соединения;
- посылает приглашение к соединению.

Разделение портов на группы по функциональному назначению можно увидеть в следующей таблице:

Таблица 7.1. Назначение портов TCP

Номера портов	Категория	Описание
0-1023	Общеизвестные порты	Номера портов назначены IANA и на большинстве систем могут быть использованы исключительно процессами системы или прикладными программами, запущенными привелигированными пользователями. p.s. IANA (от англ. Internet Assigned Numbers Authority – "Администрация адресного пространства Интернет") – американская организация, управляющая пространствами IP-адресов, доменов верхнего уровня, а также регистрирующая типы данных MIME и параметры прочих протоколов Интернета.
1024-49151	Зарегистрированные порты	Номера портов включены в каталог IANA и на большинстве систем могут быть использованы процессами обычных пользователей или программами, запущенными обычными пользователями.
49152-65535	Динамически используемые порты и/или порты, используемые внутри закрытых (private) сетей	Предназначены для временного использования — в качестве клиентских портов, портов, используемых по согласованию для приватных сервисов, а также для тестирования приложений до регистрации выделенных портов. Эти порты не могут быть зарегистрированы.

2.1.6 UDP

UDP — это ещё один протокол транспортного уровня. Он тоже базируется на IP и тоже использует порты, но в отличие от TCP он не устанавливает соединений и не требует подтверждения получения каждого пакета. Именно поэтому пакеты могут теряться или приходить в неправильном порядке. Зато этот протокол быстрее и использует меньше ресурсов. На UDP обычно базируют прикладные протоколы, которым скорость доставки данных важнее надежности, например для передачи потокового видео, общения голосом или онлайновых игр.

2.1.7 Прикладные протоколы

Большинство прикладных протоколов базируется на TCP.

У многих протоколов прикладного уровня для серверов определены стандартные порты, используемые по умолчанию. Самые известные прикладные протоколы и их стандартные порты:

HTTP — основой протокол всемирной паутины - WWW (TCP порт 80);

SMTP — протокол пересылки почты (TCP порт 25);

FTP — протокол для обращения к FTP серверу (TCP порт 21);

DNS — протокол сопоставления доменных имен IP адресам (UDP порт 53).

Благодаря использованию стандартных портов мы можем набирать в браузере адреса веб серверов и не указывать порт — наши браузеры сами добавляют стандартный номер порта. Например, адрес http://www.mytestwebsite.org/ на самом деле полностью выглядит так: http://www.mytestwebsite.org:80/

Разумеется, стандартный — не значит обязательный. Практически во всех прикладных протоколах можно указать серверу слушать произвольный номер порта. Правда, тогда этот номер уже указывать обязательно, например http://www.mytestwebsite.org:8080/

2.2 Использование Internet в приложениях на C#

Язык C# ориентирован на использование в современной вычислительной среде, в которой Internet играет немаловажную роль. Поэтому неудивительно то, что одним из основных критериев разработки C#-средств являлось обеспечение возможностей использования Internet. Несмотря на то что и такие языки программирования, как C и C++, позволяют подключаться к Internet, загружать файлы и получать желаемые ресурсы, этот процесс нельзя назвать простым и удобным, каким хотели бы его видеть многие программисты. Стандартные средства C# и библиотека .NET значительно улучшили эту ситуацию, существенно облегчив программистам создание Internet-приложений.

Поддержка сетевых возможностей реализована в двух пространствах имен. В первом, *System.Net*, определено множество высокоуровневых и удобных для использования классов, в которых предусмотрены различные операции для работы с Internet. Второе пространство имен, *System.Net.Sockets*, предназначено для поддержки сокетов, которые обеспечивают низкоуровневое управ-

ление сетевыми операциями. Поддержка серверных ASP.NET-ориентированных приложений реализована в пространстве имен *System.Web*.

2.2.1 Члены пространства имен *System.Net*

System.Net — обширное пространство имен, содержащее множество членов. И хотя здесь будут рассмотрены только некоторые из них, вы должны иметь представление о том, что вообще доступно программисту в этой области. Итак, вот классы, определенные в пространстве имен *System.Net*:

AuthenticationManager	*Authorization*	*Cookie*
CookieCollection	*CookieContainer*	*CookieException*
CredentialCache	*Dns*	*DnsPerraission*
DnsPermissionAttribute	*EndPoint*	*EndpointPermission*
FileWebRequest	*FileMebResponse*	*GlobalProxySelection*
HttpVersion	*HttpWebRequest*	*HttpWebResponse*
IPAddress	*IPEndPoint*	*IPHostEntry*
NetworkCredential	*ProtocolViolationException*	*ServicePoint*
ServicePointManager	*SocketAddress*	*SocketPermission*
SocketPermissionAttribute	*WebClient*	*WebException*
WebHeaderCollection	*WebPermission*	*MebPermissionAttribute*
WebProxy	*WebRequest*	

В пространстве имен System.Net определены также следующие интерфейсы:

IAuthenticationModule	*ICertificatePolicy*	*ICredentials*
IWebProxy	*IWebRequestCreate*	

и четыре перечисления:

HttpStatusCode	*NetworkAccess*
TransportType	*WebExceptionStatus*

Наконец, в пространстве имен *System.Net* определен один делегат: *HttpContinueDelegate*.

Несмотря на то что в пространстве имен *System.Net* определено так много членов, для решения большинства задач Internet-программирования достаточно ограничиться лишь некоторыми из них. Ядро сетевых программных средств составляют два абстрактных класса *WebRequest* и *webResponse*. Эти классы наследуются классами, которые поддерживают конкретные сетевые протоколы. Например, производные классы, предназначенные для поддержки стандартного протокола HTTP, имеют имена *HttpWebRequest* и *HttpWebResponse*.

Несмотря на то что классы *HttpWebRequest* и *HttpWebResponse* просты в применении, для некоторых задач можно воспользоваться даже еще более простым средством, а именно классом *WebClient*. Например, если вам нужно лишь загрузить в удаленный компьютер (или из него) некоторый файл, то в большинстве случаев лучше всего для этого использовать именно класс *WebClient*.

2.2.2 Универсальные идентификаторы ресурсов (URL, URI)

В основе Internet-программирования лежит универсальный идентификатор ресурса (Uniform Resource Identifier — URI). URI-идентификатор описывает местоположение в сети некоторого ресурса. URI-идентификатор часто называют также унифицированным указателем информационного ресурса (Uniform Resource Locator — URL). Поскольку при описании членов пространства имен *System.Net* компания Microsoft использует термин URI, то и в данной лабораторной работе используется именно этот термин. Вероятно, Вы уже хорошо знакомы с URI-идентификаторами. Ведь это тот адрес, который ни раз набирали в любом браузере чтобы отобразить требуемую страницу.
URI-идентификатор записывается в следующей обшей форме:

Protocol://ServerID/FilePath?Query

Элемент *Protocol* означает используемый протокол (например, HTTP). Элемент *ServerID* идентифицирует конкретный сервер (например, *VKontakte.ru* или *Weather.com*). Элемент *FilePath* задает маршрут к нужному файлу. Если элемент *FilePath* не задан, открывается страница, действующая по умолчанию для указанного значения *ServerID*. Наконец, элемент *Query* содержит информацию, которая должна быть послана серверу. Элемент *Query* необязателен. В языке C# URI-идентификаторы инкапсулированы в классе *Uri*.

2.2.3 Основы Internet-доступа в приложениях на C#

Классы, содержащиеся в пространстве имен *System.Net*, при взаимодействии с Internet поддерживают модель "запрос-ответ". Это означает, что программа-клиент запрашивает информацию с сервера, а затем ожидает ответ. Например, ваша программа может отправить серверу конкретный URI-идентификатор или адрес Web-сайта. В ответ вы получите гипертекст, связанный с указанным URI-идентификатором. Этот метод запроса-ответа удобен и прост для использования, поскольку большинство деталей обрабатываются автоматически.
Иерархии классов, в вершине которых стоят классы *WebRequest* и *WebResponse*, реализуют то, что Microsoft называет *сменными протоколами* (pluggable protocols). Как Вам уже известно, существуют различные типы протоколов сетевой связи. Наиболее распространенным для Internet-взаимодействия является протокол передачи гипертекстовых файлов (HyperText Transfer Protocol — HTTP). Часто используется и протокол передачи файлов (File Transfer Protocol — FTP), Префикс URI-идентификатора задает именно протокол. Например, в таком URI-идентификаторе, как *HTTP://MyWebSite.com*, используется префикс HTTP, который означает протокол передачи гипертекстовых файлов.
Как упоминалось выше, *WebRequest* и *WebResponse* — абстрактные классы, в которых определяются операции запроса-ответа, общие для всех протоколов. Из этих абстрактных классов выведены классы, которые реализуют конкретные протоколы. Производные классы регистрируются самостоятельно с помощью статического метода *RegisterPrefix()*, который определен в классе

WebRequest. При создании объекта класса *WebRequest* автоматически используется протокол (при условии его доступности), заданный префиксом URI. Достоинство использования настраиваемого адреса состоит в том, что большая часть программы остается неизменной при изменении типа протокола.
.NET-среда автоматически определяет протокол HTTP. Следовательно, если вы указываете URI-идентификатор с префиксом HTTP, то автоматически получите HTTP-совместимый класс, поддерживающий этот протокол. К классам, которые поддерживают протокол HTTP, относятся *HttpWebRequest* и *HttpWebResponse*. Эти классы — производные от классов *WebRequest* и *WebResponse*, и вполне естественно, что в них дополнительно определены собственные члены, которые применимы исключительно к протоколу HTTP.
В пространстве имен *System.Net* предусмотрена поддержка как синхронного, так и асинхронного взаимодействия. Для многих Internet-приложений предпочтительнее использовать синхронные транзакции, поскольку их проще реализовать. При синхронном взаимодействии программа посылает запрос, а затем ожидает ответа. Для некоторых типов высокоэффективных приложений лучше реализовать асинхронное взаимодействие. В этом случае ваша программа, ожидая информацию, может продолжать работу. Однако реализовать асинхронное взаимодействие гораздо труднее. Более того, не всем программам это "на руку". Например, нередки ситуации, когда без ожидаемой из Internet информации программе просто нечего дальше делать. В таких случаях потенциальный выигрыш, ожидаемый от применения асинхронного способа организации связи, не достигается. Синхронный доступ к Internet проще реализуем и более универсален.
Итак, начнем с самого главного: классов *WebRequest* и *WebResponse*, которые составляют ядро пространства имен *System.Net*.

2.2.4 Класс WebRequest

Класс *WebRequest* предназначен для управления сетевыми запросами. Этот класс абстрактный, поскольку не реализует конкретный протокол. Однако в нем определены методы и свойства, общие для всех запросов. Методы класса *WebRequest*, которые поддерживают синхронную передачу данных, представлены в табл. 7.2, а свойства — в табл. 7.3. Стандартные значения для свойств определяются производными классами. Класс *WebRequest* не определяет ни одного public-конструктора.
Чтобы отправить запрос по URI-адресу, необходимо сначала создать объект класса (выведенного из класса *WebRequest*), который реализует желаемый протокол. Это можно сделать с помощью статического метода *Create()*, определенного в классе *WebRequest*. Метод *Create()* возвращает объект класса, который является производным от класса *WebRequest* и реализует нужный протокол.

Таблица 7.2. Методы поддержки синхронной передачи данных, определенные в классе *WebRequest*

Метод	Описание

public static WebRequest Create(string *uri*);	Создает *WebRequest*-объект для URI-идентификатора, заданного строкой, переданной параметром *uri*. Возвращаемый объект реализует протокол, заданный префиксом URI-идентификатора, Следовательно, возвращаемый объект будет объектом класса, производного от *WebRequest*. Если затребованный протокол недоступен, генерируется исключение типа *NotSupportedException*. Если же недействителен указанный формат URI-идентификатора, генерируется исключение типа *UriFormatException*
public static WebRequest Create(Url *uri*);	Создает *WebRequest*-объект для URI-идентификатора, заданного строкой, переданной параметром *uri*. Возвращаемый объект реализует протокол, заданный префиксом URI-идентификатора. Следовательно, возвращаемый объект будет объектом класса, производного от *WebRequest*. Если затребованный протокол недоступен, генерируется исключение типа *NotSupportedException*
public virtual Stream GetRequestStream()	Возвращает выходной поток, связанный с предварительно зарегистрированным URI-идентификатором
public virtual WebResponse GetResponse()	Отсылает предварительно созданный запрос и ожидает ответ. После получения ответа возвращает его в виде объекта класса *WebReponse*. Программа должна использовать этот объект для получения информации от заданного URI. При возникновении ошибки в процессе получения ответа генерируется исключение типа *WebException*

Таблица 7.3. Свойства, определенные в классе *WebRequest*

Свойство	Описание
public virtual string ConnectionGroupName { get; set; }	Получает или устанавливает групповое имя подключения. Группы подключения представляют собой способ создания набора запросов. Они не нужны для простых Internet-транзакций
public virtual long ContentLength { get; set; }	Получает или устанавливает длину пакета переданных данных
public virtual string ContentType { get; set; }	Получает или устанавливает описание переданной информации
public virtual ICredentials Credentials { get; set; }	Получает или устанавливает уровень доступа. Уровни доступа необходимы для тех сайтов, которые требуют аутентификации (служба контроля доступа, проверяющая регистрационную информацию пользователя)
public virtual WebHeaderCollection Headers{ get; set; }	Получает или устанавливает коллекцию заголовков
public virtual string Method { get; set; }	Получает или устанавливает протокол
public virtual bool PreAuthenticate { get; set;)	Если равно значению true, в отправляемый запрос включается регистрационная информация. Если равно значению false, регистрационная информация предоставляется только по URI-требованию

окончание таблицы 7.3

public virtual IWebProxy Proxy { get; set; }	Получает или устанавливает путь к proxy-серверу. Применимо только в средах, в которых используется proxy-сервер

public virtual Uri RequestTjri { get; }	Получает URI запроса
public virtual int Timeout { get; set; }	Получает или устанавливает количество миллисекунд, в течение которых будет ожидаться ответ на запрос. Для установки бесконечного ожидания используйте значение *Timeout.infinite*

2.2.4 Класс WebResponse

Класс *WebResponse* инкапсулирует ответ, который принимается в качестве результата запроса. Класс *WebResponse* – абстрактный. Производные классы создают конкретные версии, ориентированные на поддержку того или иного протокола. Объект класса *WebResponse* обычно создается посредством вызова метода *GetResponse()*, определенного в классе *WebRequest*. Этот объект является экземпляром класса (выведенного из класса *WebReponse*), который реализует конкретный протокол. Методы, определенные в классе *WebResponse*, представлены в таблице 7.4, а свойства – в таблице 7.5. Значения этих свойств устанавливаются на основе каждого ответа. Класс *WebResponse* не определяет ни одного *public*-конструктора.

Таблица 7.4. Методы, определенные в классе *WebResponse*

Метод	Описание
public virtual void Close ()	Закрывает поток, содержащий ответ. Закрывает также поток ответа, который был возвращен методом *GetResponseStream()*
public virtual Stream GetResponse stream()	Возвращает входной поток, связанный с затребованным URI. С по этого потока данные могут быть считаны с URI-источника

Таблица 7.5. Свойства, определенные в классе *WebResponse*

Свойство	Описание
public virtual long ContentLength { get; set; }	Получает или устанавливает длину пакета принятых данных. Устанавливается равным -1, если данные о длине недоступны
public virtual string ContentType { get; set; }	Получает описание принимаемой информации
public virtual WebHeaderCollection Headers{ get; }	Получает коллекцию заголовков, связанных с URI
public virtual Uri ReponseUri { get; }	Получает URI, который сгенерировал ответ. Этот адрес может отличаться от запрашиваемого, если ответ был перенаправлен на другой URI-адрес

2.2.5 Классы HttpWebRequest и HttpWebResponse

Классы *HttpWebRequest* и *HttpWebResponse* (производные от классов *WebRequest* и *WebResponse*, соответственно) реализуют протокол HTTP. В обоих определен ряд собственных свойств, которые позволяют детализировать информацию о HTTP-транзакциях. Однако при выполнении простых Internet-операций вам вряд ли придется прибегать к этим дополнительным средствам.

2.3 Обработка сетевых ошибок

Несмотря на то что программа может быть вполне корректна, она совершенно беззащитна перед потенциальными "ударами судьбы", которые могут внезапно оборвать ее "жизнь". В приложениях необходимо предусмотреть все возможные неприятности и обеспечить полную обработку сетевых исключений, которые может сгенерировать программа. Для этого нужно проконтролировать обращения к методам *Create(), GetResponse()* и *GetResponseStream()*. Все типы потенциальных ошибок будут рассмотрены далее.

2.3.1 Исключения, генерируемые методом Create()

Метод *Create()*, определенный в классе *WebRequest*, может сгенерировать три исключения. Если протокол, заданный URI-префиксом, не поддерживается, генерируется исключение типа *NotSupportedException*. Если неверно задан формат URI-идентификатора, генерируется исключение типа *UriFormatException*. Метод *Create()*, вызванный с использованием нулевой ссылки, может сгенерировать также исключение типа *ArgumentNullException*, хотя эта ошибка и не относится к тем, которые генерируются сетевыми средствами.

2.3.2 Исключения, генерируемые методом GetReponse()

Если ошибка обнаружится в процессе получения ответа, то есть при вызове метода *GetReponse()*, генерируется исключение типа *WebException*. Помимо членов, определенных для всех исключений, в классе *WebException* определены также два дополнительных свойства, которые связаны с сетевыми ошибками: *Response* и *Status*.

Внутри обработчика исключения с помощью свойства Response можно получить ссылку на объект класса *WebResponse*. Этот объект будет возвращен методом *GetReponse()*, если исключение не генерируется. Определение свойства *Response* выглядит так:

public WebResponse Response {get;}

Если ошибка все-таки возникла, то чтобы понять, что именно произошло, можно использовать свойство *status*. Его определение имеет следующий вид:

public WebExceptionStatus Status {get;}

WebExceptionStatus – это перечисление, которое содержит следующие значения:

ConnectFailure	*ConnectionClosed*	*KeepAliveFailure*

NameResolutionFailure
Pending
PipelineFailure
ProtocolError
ProxyNameResolutionFailure
ReceiveFailure
RequestCanceled
SecureChannelFailure
SendFailure
ServerProtocolViolation
Success
Timeout
TrustFailure

После выяснения причины ошибки программа может предпринять соответствующие действия.

2.3.3 Исключения, генерируемые методом GetResponseStream()

Метод *GetResponseStream()* класса *GetResponse* может генерировать исключение типа *ProtocolViolationException*, которое в общем случае означает, что произошла ошибка, связанная с протоколом. А поскольку ее возникновение имеет отношение к методу *GetResponseStream()*, значит нет ни одного потока, содержащего ответную информацию. В процессе чтения потока может также быть сгенерировано исключение типа *IOException*.

2.4 Класс URI

Возможно, вы обратили внимание на то, что в табл. 7.2 метод *WebRequest.Create()* представлен в двух различных версиях. Одна из них принимает URI в виде строки. Вторая принимает URI как экземпляр класса *Uri*. Класс *Uri* инкапсулирует URI-идентификатор. Используя класс *Uri*, можно создать такой URI-адрес, который будет принят второй версией метода Create(). При этом можно разбить URI-идентификатор на части. И хотя в случае простых Internet-операций можно обойтись без класса *Uri*, все же в более сложных случаях он окажется ценным.

В классе *Uri* определено несколько конструкторов. Два наиболее употребимые из них определяются так:

public Uri(string uri)

public Uri(string base, string rel)

Первая форма позволяет создать *Uri*-объект на основе URI-адреса, заданного в виде строки. Вторая служит для создания *Uri*-объекта путем сложения относительного URI-идентификатора, заданного параметром *rel*, с базовым URI-идентификатором, заданным параметром *base*. Базовый URI-идентификатор определяет сам URI-адрес, а относительный – только сетевой маршрут.

В классе *Uri* определено множество полей, свойств и методов, которые позволяют управлять URI-идентификаторами или предоставляют доступ к различным его частям. Чаще всего используются следующие свойства.

Таблица 7.6. Свойства, определенные в классе *Uri*

Свойство	Описание
public string Host { get; }	Получает имя сервера
public string LocalPath { get; }	Получает маршрут, определяющий местоположение файла
public string PathAndQuery { get; }	Получает маршрут и строку запроса
public int Port { get; }	Получает номер порта для заданного протокола. Для HTTP номер порта равен 80
public string Query { get; }	Получает строку запроса
public string Scheme { get; }	Получает протокол

Эти свойства полезно применять при разбиении URI-идентификатора на составные части. Их использование демонстрирует следующая программа:

```
//Пример 1. Использование класса Uri.
using System;
using System.Net;

class UriDemo
{
   public static void Main()
   {
      Uri sample = new Uri("http://MySite.com:8080/somefile.txt?SomeSampleQuery");

      Console.WriteLine("Host:\t\t" + sample.Host);
      Console.WriteLine("Port:\t\t" + sample.Port);
      Console.WriteLine("Protocol:\t" + sample.Scheme);
      Console.WriteLine("Local Path:\t" + sample.LocalPath);
      Console.WriteLine("Query:\t\t" + sample.Query);
      Console.WriteLine("Path & Query:\t" + sample.PathAndQuery + "\n");
   }
}
```

Результат выполнения программы будет следующим:

```
Host:           mysite.com
Port:           8080
Protocol:       http
Local Path:     /somefile.txt
Query:          ?SomeSampleQuery
```

Path & Query: /somefile.txt?SomeSampleQuery

2.5 Доступ к дополнительной HTTP-информации

Используя класс *HttpWebResponse*, можно получить доступ не только к содержимому заданного ресурса, но и к информации, которая включает, например, время последней URI-модификации и имя сервера. Эту информацию можно получить с помощью различных свойств, перечисленных в таблице7.7 (четыре из них определены в классе *WebResponse*). Об их использовании пойдет далее речь.

Таблица 7.7. Свойства, определенные в классе *HttpWebResponse*

Свойство	Описание
public string CharacterSet { get; }	Получает название используемого символьного набора
public string ContentEncoding { get; }	Получает название схемы кодирования
public long ContentLength { get; }	Получает длину принимаемого содержимого. Если она недоступна, свойство содержит -1
public string ContentType { get; }	Получает описание содержимого
public CookieCollection Cookies { get; set; }	Получает или устанавливает список cookie-данных, присоединенных к ответу
public WebHeaderCollection Headers{ get; }	Получает коллекцию заголовков, присоединенных к ответу
public DateTime LastModified { get; }	Получает время последней URI-модификации
public string Method { get; }	Получает строку, которая задает способ ответа
public Version ProtocolVersion { get; }	Получает объект класса *Version*, который описывает версию протокола HTTP, используемую в транзакции
public Uri ReponseUri { get; }	Получает URI-идентификатор, по которому сгенерирован ответ. Он может отличаться от запрошенного, если ответ был перенаправлен по другому URI-адресу
public string Server { get; }	Получает строку, которая представляет имя сервера
public HttpStatusCode StatusCode { get; }	Получает объект класса *HttpStatusCode*, который описывает состояние транзакции
public string StatusDescription { get; }	Получает строку, которая представляет состояние транзакции в форме, удобной для восприятия человеком

2.5.1 Доступ к заголовку

С помощью свойства *Headers*, которое определено в классе *HttpWebResponse*, можно получить доступ к заголовочной информации HTTP-ответа:

public WebHeaderCollection Headers {get;}

HTTP-заголовок состоит из пар имя/значение, представленных в виде строк. Каждая такая пара хранится в объекте класса *WebHeaderCollection*, Это – специализированная коллекция, предназначенная для хранения пар ключ/значение, которую можно использовать подобно любой другой коллекции. Строковый массив имен можно получить из свойства *AllKeys*. Значение, связанное с конкретным именем, можно получить с помощью индексатора. Индексатор здесь перегружен на прием либо числового индекса, либо имени.

В следующей программе отображаются все заголовки, относящиеся к Web-сайту *www.vkontakte.ru*:

```
//Пример 2. Отображение заголовков Web-сайта.
using System;
using System.Net;

class HeaderDemo
{
   public static void Main()
   {
      //Создаем WebRequest-запрос по URI-адресу.
      HttpWebRequest req = (HttpWebRequest)
        WebRequest.Create("http://www.vkontakte.ru");

      //Отправляем этот запрос и получаем ответ.
      HttpWebResponse resp = (HttpWebResponse)
        req.GetResponse();

      //Получаем список имен.
      string[] names = resp.Headers.AllKeys;

      //Отображаем заголовок в виде пар имя/значение.
      Console.WriteLine("{0,-20}{1}\n", "Name", "Value");
      foreach (string n in names)
         Console.WriteLine("{0,-20}{1}", n, resp.Headers[n]);

      //Закрываем поток, содержащий ответ.
      resp.Close();
   }
}
```

Результат выполнения программы:

```
Name                Value

Connection          keep-alive
Pragma              no-cache
```

Cache-Control	private, must-revalidate
Content-Type	text/html; charset=windows-1251
Date	Sat, 10 May 2008 19:34:50 GMT
Set-Cookie	remixchk=5; expires=Sun, 10-May-2009 19:34:50 GMT; path=/; domain=.vkontakte.ru,remixchk=5; expires=Sun, 10-May-2009 19:34:50 GMT; path=/; domain=.vkontakte.ru
Server	nginx/0.5.35
X-Powered-By	PHP/5.2.0-8+etch10
Content-Length	6827

2.5.2 Доступ к данным Cookie

Доступ к cookie-данным, связанным с HTTP-ответом, можно получить с помощью свойства Cookies, которое определено в классе *HttpWebResponse*. Cookie-данные, которые хранятся браузером, состоят из пар имя/значение. Они способны упростить определенные типы операций Web-доступа. Вот как выглядит определение свойства

Cookies:
public CookieCollection Cookies (get; set; }

Класс CookieCollection реализует интерфейсы *ICollection* и *IEnumerable*, и его можно использовать подобно любой другой коллекции. Он имеет индексатор, который позволяет получить cookie-данные по заданному индексу или его имени.
Коллекция типа *CookieCollection* предназначена для хранения объектов типа *Cookie*. В классе *Cookie* определено несколько свойств, которые предоставляют доступ к различным частям cookie-данных. Нас интересуют свойства *Name* и *Value*:

public string Name { get; set; }
public string value { get; set; }

Нетрудно догадаться, что cookie-имя содержится в свойстве *Name*, а значение – в свойстве *Value*.
Чтобы получить список cookie-составляюших, связанных с ответом, необходимо воспользоваться cookie-контейнером. Для этого в классе *HttpWebRequest* определено свойство *CookieContainer*:

public CookieContainer CookieContainer { get; set; }

В классе *CookieContainer* определены различные поля, свойства и методы, которые позволяют хранить cookie-данные. Однако при создании многих приложений нет необходимости в непосредственном использовании свойства *CookieContainer*. Обычно используют коллекцию типа

CookieCollection, получаемую из ответа. Назначение класса *CookieContainer* – обеспечить механизм хранения cookie-данных.

2.5.4 Использование свойства LastModified

Иногда нужно узнать, когда в последний раз обновлялся Web-сайт с заданным URI-адресом. С помощью класса *HttpWebResponse* это не составляет труда, поскольку в нем определено свойство *LastModified*:

public DateTime LastModif ied { get;}

Свойство *LastModified* получает время последней URI-модификаиии. При выполнении следующей программы отображается время и дата последней модификации Web-сайта *http://is.kubagro.ru*:

```
//Пример 3. Использование свойства LastModified.

using System;
using System.Net;

class HeaderDemo
{
   public static void Main()
   {
      HttpWebRequest request = (HttpWebRequest)
         WebRequest.Create("http://is.kubagro.ru");
      HttpWebResponse response = (HttpWebResponse)
         request.GetResponse();
      Console.WriteLine(request.Address + " was last modified: " +
response.LastModified);
      response.Close();
   }
}
```

2.6 Использование класса WebClient

Как упоминалось ранее в данной лабораторной работе, если вам нужно загрузить на удаленный компьютер (или из него) файл, то вместо классов *WebRequest* и *WebResponse* можно для использовать класс *WebClient*. Достоинство класса *WebClient* состоит в том, что он выполняет многие действия сам, освобождая таким образом вас от написания сложных обработчиков.

В классе *WebClient* определен один конструктор:

public WebClient()

В нем также определены весьма полезные свойства (см. таблицу 7.8) и методы (см. таблицу 7.9). Все методы генерируют исключение типа *UriFormatException*, если заданный URI-адрес окажется недействительным, и исключение типа *WebException*, если во время передачи данных возникнет ошибка.

Таблица 7.8. Свойства, определенные в классе *WebClient*

Свойство	Описание
public string BaseAddress { get; set; }	Получает или устанавливает базовый адрес нужного URI. По умолчанию это свойство имеет *null*-значение. Если это свойство установлено, то адреса, заданные методами класса *WebClient*, должны интерпретироваться относительно этого базового адреса
public ICredentials Credentials { get; set; }	Получает или устанавливает регистрационную информацию. Это свойство по умолчанию имеет значение *null*
public WebHeaderCollection Headers{ get; set; }	Получает или устанавливает коллекцию заголовков запроса
public NameValueCollection QueryString { get; set; }	Получает или устанавливает строку запроса, состоящую из пар имя/значение, которые могут быть присоединены к запросу. Строка запроса отделяется от URI символом "?". Если таких пар больше одной, то каждая из них отделяется символом "@"
public WebHeaderCollection ResponseHeaders{ get; }	Получает коллекцию заголовков ответа

Таблица 7.9. Методы, определенные в классе *WebClient*

Метод	Описание
public byte[] DownloadData(string *uri*)	Загружает информацию с Web-страницы, URI-адрес которой задается параметром *uri*. Возвращает результат в виде массива байтов
public void DownloadFile(string *uri*, string *fname*)	Загружает информацию с Web-страницы, URI-адрес которой задается параметром *uri*, и сохраняет результат в файле, имя которого задается параметром
public Stream OpenRead(string *uri*)	Возвращает входной поток, информацию из которого можно прочитать с использованием URI-адреса, заданного параметром *uri*. После завершения считывания этот поток необходимо закрыть
public Stream OpenWrite(string *uri*)	Возвращает выходной поток, в который можно записать информацию с помощью URI-адреса, заданного параметром *uri*. После завершения записи этот поток необходимо закрыть
public Stream OpenWrite(string *uri*, string *how*)	Возвращает выходной поток, в который можно записать информацию с использованием URI-адреса, заданного параметром *uri*. После завершения записи этот поток необхо-

	димо закрыть. Строка, переданная в параметре *how*, задает способ записи этой информации

окончание таблицы 7.9

public byte[] UploadData(string *uri*, byte[] *info*)	Записывает информацию, заданную параметром *info*, по URI-адресу, заданному параметром *uri*. Возвращает ответ
public byte[] UploadData[string *uri*, string *how*, byte[] *info*)	Записывает информацию, заданную параметром *info*, по URI-адресу, заданному параметром *uri*. Возвращает ответ. Строка, переданная в параметре *how*, задает способ записи этой информации
public byte[] UploadFile(string *uri*, string *fname*)	Записывает информацию из файла, имя которого задается параметром *fname*, по URI-адресу, заданному параметром *uri*. Возвращает ответ
public byte[] UploadFile(string *uri*, string *how*, string *fname*)	Записывает информацию из файла, имя которого задается параметром *fname*, по URI-адресу, заданному параметром *uri*. Возвращает ответ. Строка, переданная в параметре *how*, задает способ записи этой информации
public byte[] UploadValues(string *uri*, NameValueCollection *vals*)	Записывает значения, хранимые в коллекции, заданной параметром *vals*, по URI-адресу, заданному параметром *uri*. Возвращает ответ
public byte[] UploadValues(string *uri*, string *how*, NameValueCollection *vals*)	Записывает значения, хранимые в коллекции, заданной параметром *vals*, по URI-адресу, заданному параметром *uri*. Возвращает ответ Строка, переданная в параметре *how*, задает способ записи этой информации

Использование класса WebClient для загрузки данных в файл демонстрируется в следующей программе:

```
//Пример 4. Использование класса WebClient для загрузки
//информации в файл.

using System;
using System.Net;
using System.IO;

class WebClientDemo
{
  public static void Main()
  {
    WebClient user = new WebClient();
    string uri = "http://krasnodar.50webs.org/";
    string fname = "data.out";

    try
    {
```

```
            Console.WriteLine(
               "Downloading from Web-site " +
               uri + " in file " + fname);
            user.DownloadFile(uri, fname);
         }
         catch (WebException exc)
          { Console.WriteLine(exc); }
         catch (UriFormatException exc)
          { Console.WriteLine(exc); }
         Console.WriteLine("Download Completed.");
      }
   }
```

Результат работы программы:

```
Downloading from Web-site http://krasnodar.50webs.org/ in file data.out
Download Completed.
```

Эта программа загружает информацию с Web-сайта http://krasnodar.50webs.org/ и помещает ее в файл с именем *data.out*. Обратите внимание на то, что все это реализуется очень небольшим количеством строк кода, в число которых включены и те, которые обеспечивают обработку двух возможных исключений. Приведенная программа позволяет загрузить информацию с любого URI-адреса: для этого достаточно изменить соответствующим образом строку, заданную переменной *uri*.

Несмотря на то что классы *WebRequest* и *WebResponse* дают программисту большие возможности для управления и доступ к более обширной информации, средств, инкапсулированных классом *WebClient*, вполне достаточно для потребностей многих приложений. Он особенно полезен в том случае, если приложение должно обеспечить лишь загрузку информации из Web-пространства. Например, с помощью класса *WebClient* вы могли бы организовать получение обновленной документации по интересующим вас продуктам.

3 Примеры программ

3.1 Пример консольного приложения

Задание: Разработать программу на C# в которой будет использована обработка функций классов *WebRequest* и *WebResponse*. Консольное приложение должно работать в синхронном режиме, то есть после HTTP запроса ждать отклика. Текст должен выводиться порциями по 400 символов и ожидать нажатия кнопки <ENTER> для продолжения пролистывания. Также необходимо описать обработчики всех возможных сетевых исключений.

Выполнение:

```
//Пример 5. Прием гипертекстового документа.
//Обработка сетевых исключений.
using System;
using System.Net;
using System.IO;

class NetExcDemo
{
  public static void Main()
  {
    int ch;

    try
    {
      //Сначала создаем WebRequest-запрос по URI-адресу.
      HttpWebRequest req = (HttpWebRequest)
        WebRequest.Create("http://www.vkontakte.ru");

      //Затем отправляем запрос и получаем ответ.
      HttpWebResponse resp = (HttpWebResponse)
        req.GetResponse();

      //Из ответа получаем входной поток.
      Stream istrm = resp.GetResponseStream();

      /*Теперь считываем и отображаем html-документ,
       полученный от заданного URI. Текст "не улетит"
       с экрана, поскольку данные отображаются порциями
       объемом в 400 символов. Просмотрев очередные
       400 символов, нажмите клавишу <ENTER>
       для получения следующей часть документа.*/

      for (int i = 1; ; i++)
      {
        ch = istrm.ReadByte();
        if (ch == -1) break;
        Console.Write((char)ch);
        if ((i % 400) == 0)
        {
          Console.Write("Press any key");
          Console.Read();
        }
      }

      /*Закрываем поток, содержащий ответ.
        При этом автоматически закроется и входной поток istrm.*/
```

```
            resp.Close();
        }
        catch (WebException exc)
        {
            Console.WriteLine("Network Error: " + exc.Message + "\nStatus Code: " +
exc.Status);
        }
        catch (ProtocolViolationException exc)
        {
            Console.WriteLine("Protocol Error: " + exc.Message);
        }
        catch (UriFormatException exc)
        {
            Console.WriteLine("Error in URI format: " + exc.Message);
        }
        catch (NotSupportedException exc)
        {
            Console.WriteLine("Unhandled Protocol: " + exc.Message);
        }
        catch (IOException exc)
        {
            Console.WriteLine("I/O Error: " + exc.Message);
        }
    }
}
```

В этом варианте программы будут перехвачены исключения, потенциально генерируемые Internet-методами. Например, если используемое в программе обращение к методу *Create()* заменить следующим:

```
WebRequest.Create("http://www.vkontakte.ru/none");
```

и перекомпилировать программу, а затем ее выполнить, вы непременно получите такое сообщение:

```
Network Error: The remote server returned an error: (404) Not Found.
Status Code: ProtocolError
```

Поскольку сайт www.vkontakte.ru не имеет каталога с именем "none", вполне естественно, что этот URI-адрес не обнаружен.

3.2 Пример приложения на C# с использованием форм

Задание: Разработать Windows-приложение, основанное на форме. Функционально приложение будет таким же, как и предыдущий консольный пример. Обеспечить возможность ввода адреса, номера порта. Описать обработчики исключений и посчитать количество символов, которое обработала программа.

Выполнение: Сначала нужно разработать форму и расставить на ней компоненты. После этого Вы можете воспользоваться таблицей 7.10 для того чтобы задать всем компонентам необходимые свойства.

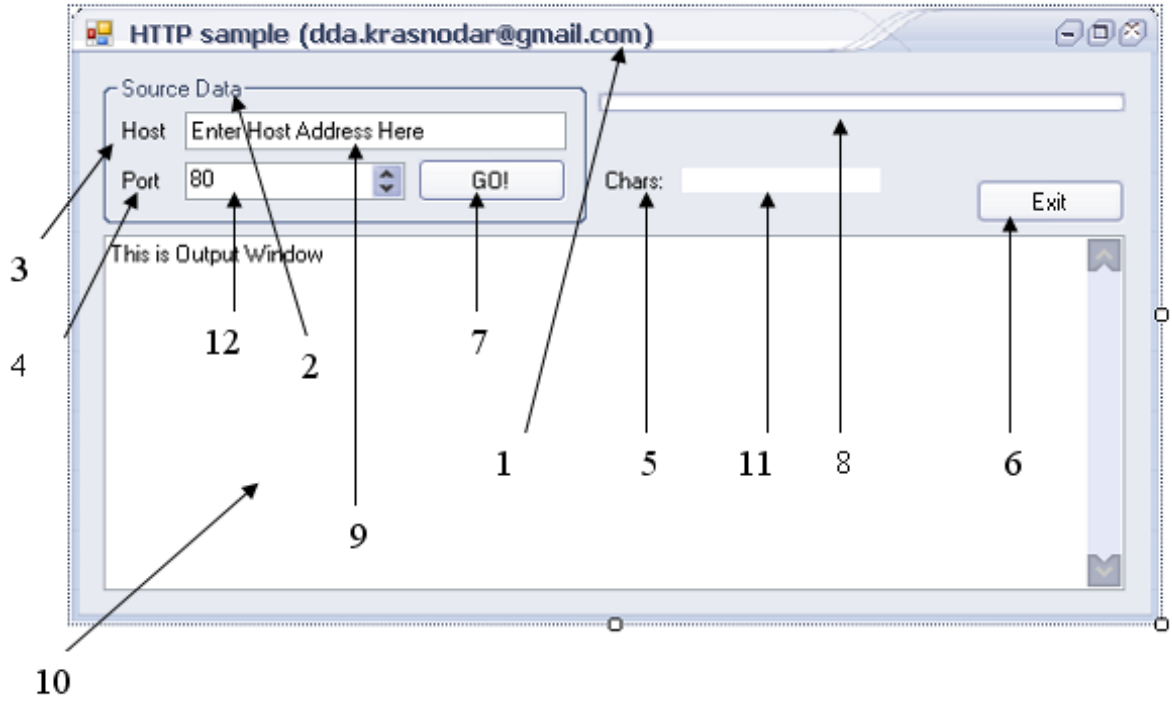


Рисунок 7.2. Вид окна разрабатываемого приложения

Таблица 7.10. Свойства компонентов программы

№	Компонент	Свойство	Значение
1	Form1	Text	HTTP sample (dda.krasnodar@gmail.com)
		StartPosition	CenterScreen
2	groupBox1	Text	Source Data
3	label1	Text	Host
4	label2	Text	Port
5	label3	Text	Chars:
		Visible	False
6	button1	Text	Exit

7	button2	Text	GO!
8	progressBar1		
9	textBox1	Text	Enter Host Address Here
10	textBox2	Multiline	True
		ScrollBars	Vertical
		Text	This is Output Window
		False	Enabled
окончание таблицы 7.10			
11	textBox3	Enabled	False
12	numericUpDown1	Name	numericUpDown
		Maximum	65535
		Value	80

Если Вы запустите программу без внесения кода, то увидите что компоненты *label3* и *textBox3* не появляются на форме. Это сделано для того, чтобы можно было проверить правильность набора кода программы и проследить в каком месте происходит неверное выполнение кода.
Чтобы программа начала реагировать на действия пользователя надо добавить обработку на компоненты *button1*, *button2*. Первое действие – это выходи из программы. Ее код выглядит так:

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

О действии второй кнопки будет описано позже. В программе описывается 2 реакции на события. Первое событие сгенерирует строку "http://" в компоненте *textBox1* в том случае, если пользователь собирается вводить адрес хоста. Второе событие создано для того чтобы проследить за правильностью ввода данных в *numericUpDown*. Как Вы помните, каждое соединение может быть осуществлено с подключением к определенному порту, если только этот порт не является "портом по умолчанию" в интерпретации протокола HTTP. Согласно таблице 1, мы можем использовать порты из пространства 0-49151. Код следит за корректностью ввода этого числа и если оно выпадает из диапазона – ставит 80-й порт автоматически.
Код для реакции на событие *Enter* компонента *textBox1*

```
private void textBox1_Enter(object sender, EventArgs e)
    {
      textBox1.Text = "http://";
    }
```

Код для реакции на событие *ValueChanged* компонента *numericUpDown*

```
private void numericUpDown_ValueChanged(object sender, EventArgs e)
    {
      int a = (int)numericUpDown.Value;
      if ((a < 0) || (a > 49151))
      numericUpDown.Value = 80;
    }
```

Теперь для корректной работы программы и выполнения поставленного задания нужно сделать обработку кнопки *button2*, которая как раз и будет инициировать соединение программы с хостом (внешним сервером) по указанному адресу (как в формате ***n.n.n.n***, так и в формате имен), подсоединяться к указанному порту и "вытягивать" необходимые данные. В данном случае нам нужно прочитать содержимое корневого каталога сайта. Но как это сделать в компонент *textBox2*, если учесть что мы "достаем" по одному байту? Можно организовать массив, помещать все "байты" в него, а затем преобразовывать их в *Char*, затем в *String* и отправлять в *textBox2*. В этом примере был выбран другой способ: создавалась временная переменная, которая хранила текущее значение *textBox2.text*, затем "вытягивался" следующий байт и он прибавлялся к уже существующему значению. Затем значение снова отправлялось во временную переменную и к нему на следующем шаге прибавлялся новый знак (стоит отметить тот факт, что он сразу конвертировался из *byte* в *char*, а затем в *string* чтобы сразу получить символ, а не байтовое представление). В этом же коде помещена обработка исключительных ситуаций с ошибками протокола HTTP и подсчет количества символов, обработанных программой.

p.s. Конечно же работа через массив была бы намного быстрее, ведь можно сразу добавлять элемент в его конец, а потом отобразить его полностью. Но так как скорость работы программы не была очень важна, а важно было показать последовательность действий, которая производится при запросе данных с веб-узла, был выбран вариант с записью во временную переменную.

Код обработки кнопки *button2*

```
private void button2_Click(object sender, EventArgs e)
    {
      long ch;
      try
      {
        HttpWebRequest req = (HttpWebRequest)
          WebRequest.Create(textBox1.Text + ":" + numericUpDown.Value);
        HttpWebResponse resp = (HttpWebResponse)
          req.GetResponse();
```

```
            Stream istrm = resp.GetResponseStream();
            textBox2.Text = "";
            textBox2.Enabled = true;
            textBox3.Visible = true;
            label3.Visible = true;
            for (long i = 1; ; i++)
            {
               ch = istrm.ReadByte();
               if (ch == -1) break;
               //Создается временная переменная чтобы была возможность
               //поместить более одного символа в TextBox
               string temp = textBox2.Text;
               string data = Convert.ToString((char)ch);
               //Тут временная переменная складывается с текущим символом
               //и получается непрерывная последовательность символов
               textBox2.Text = temp + data;
               //Вывод количества символов, обработанных программой
               textBox3.Text = Convert.ToString(i);
               progressBar1.Increment(1);
            }
            resp.Close();
         }
         catch (WebException exc)
         {MessageBox.Show(exc.Message + "\n" + exc.Status);}
         catch (ProtocolViolationException exc)
         {MessageBox.Show(exc.Message); }
         catch (UriFormatException exc)
         {MessageBox.Show(exc.Message); }
         catch (NotSupportedException exc)
         {MessageBox.Show(exc.Message); }
         catch (IOException exc)
         {MessageBox.Show(exc.Message); }
      }
```

4. Варианты заданий для самостоятельной работы

Примечание. Все программы должны быть созданы в качестве Windows-приложений.

1-2. Дается список адресов, который содержится в компоненте *DataGridView* в одном столбце. Необходимо создать программу, которая во втором столбце отобразит версию программного обеспечения сервера. Использовать адреса "192.168.20.111" и "192.168.20.200" (порт 80).

3-4. Дается список адресов, который содержится в компоненте *DataGridView* в одном столбце (адреса "192.168.20.111" и "192.168.20.200"

[порт 80]). Обеспечить вывод информации о дате и времени на хостах в компонент *TextBox.*

5-6. Пользователь вводит адреса для проверки времени последней модификации ПО на сервере через *TextBox*. После обработки программой все даты записать парами "хост/дата" в две колонки *DataGridView*.

7-8. Отобразить в неформатированном виде все ссылки (строки, которые содержат значение "<a href=" в корневом каталоге сервера http://192.168.20.111.

9-10. Дается список адресов, который содержится в компоненте *DataGridView* в одном столбце (адреса "192.168.20.111" и "192.168.20.200" [порт 80]). Обеспечить вывод информации о дате и времени на хостах во второй столбец компонента *DataGridView.*

11-12. Загрузить данные из указанного местоположения в файл. Местоположение задается через компонент *TextBox*. Предусмотреть возможность выбора альтернативного номера порта пользователем.

13-14.Загрузить данные из указанного местоположения в файл. Местоположение задается через столбец компонента *DataGridView*. Предусмотреть возможность выбора альтернативного номера порта пользователем.

15-17. Дается список серверов (192.168.20.111 и 192.168.20.200) через *DataGridView*. Осуществить просмотр всех доступных о них сведений из заголовков и сохранить их в файл.

18-21. Дается список серверов (192.168.20.111 и 192.168.20.200) через два поля *TextBox*. Осуществить просмотр всех доступных о них сведений из заголовков и отобразить их последовательно в *TextBox.*

22-24. Отобразить всё мета-содержимое (строки, начинающиеся с "<*meta* " заглавной страницы сервера 192.168.20.111:80. Обеспечить его ввод через *TextBox*, а вывод в файл.

25-29. Отобразить всё мета-содержимое (строки, начинающиеся с "<*meta* "заглавной страницы сервера 192.168.20.111:80. Обеспечить ввод адреса через *TextBox* (но предусмотреть чтобы он уже был в поле в момент запуска программы). Вывод осуществить в другой *TextBox*.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Агапов В.П. Основы программирования на языке C# [Электронный ресурс]: учебное пособие/ Агапов В.П.— Электрон. текстовые данные.— М.: Московский государственный строительный университет, ЭБС АСВ, 2012.— 128 с.— Режим доступа: http://www.iprbookshop.ru/16366.— ЭБС «IPRbooks», по паролю

2. Васильев А.Н. Объектно-ориентированное программирование на C++ [Электронный ресурс]/ Васильев А.Н.— Электрон. текстовые данные.— СПб.: Наука и Техника, 2016.— 544 с.— Режим доступа: http://www.iprbookshop.ru/60648.html.— ЭБС «IPRbooks», по паролю

3. Котов О.М. Язык C#. Краткое описание и введение в технологии программирования [Электронный ресурс] : учебное пособие / О.М. Котов. — Электрон. текстовые данные. — Екатеринбург: Уральский федеральный университет, 2014. — 208 с. — 978-5-7996-1094-4. — Режим доступа: http://www.iprbookshop.ru/68524.html.— ЭБС «IPRbooks», по паролю

4. Логинова, Ф.С. Объектно-ориентированные методы программирования [Электронный ресурс] : учебное пособие. — Электрон.дан. — СПб. : ИЭО СПбУУиЭ (Институт электронного обучения Санкт-Петербургского университета управления и экономики), 2012. — 208 с. — Режим доступа: http://e.lanbook.com/books/element.php?pl1_id=64040, по паролю

2. Мейер Б. Объектно-ориентированное программирование и программная инженерия [Электронный ресурс]/ Мейер Б.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Эр Медиа, 2019.— 285 с.— Режим доступа: http://www.iprbookshop.ru/79706.html.— ЭБС «IPRbooks», по паролю

3. Бабушкина, И.А. Практикум по объектно-ориентированному программированию [Электронный ресурс] : учебное пособие / И.А. Бабушкина, С.М. Окулов. — Электрон.дан. — М. : "Лаборатория знаний" (ранее "БИНОМ. Лаборатория знаний"), 2015. — 369 с. — Режим доступа: http://e.lanbook.com/books/element.php?pl1_id=66121, по паролю.

4. Павлова Е.А. Технологии разработки современных информационных систем на платформе Microsoft .NET [Электронный ресурс]/

Павлова Е.А.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 128 c.— Режим доступа: http://www.iprbookshop.ru/16101.— ЭБС «IPRbooks», по паролю

5. Борисенко В.В. Основы программирования [Электронный ресурс]/ Борисенко В.В.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 323 c.— Режим доступа: http://www.iprbookshop.ru/22427.— ЭБС «IPRbooks», по паролю

6. Туральчук К.А. Параллельное программирование с помощью языка C# [Электронный ресурс]/ Туральчук К.А.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Эр Медиа, 2019.— 189 c.— Режим доступа: http://www.iprbookshop.ru/79714.html.— ЭБС «IPRbooks», по паролю

7. Золотарёв О.В. Технология внедрения корпоративных информационных систем [Электронный ресурс]: методические указания к лабораторным работам/ Золотарёв О.В.— Электрон. текстовые данные.— М.: Российский новый университет, 2013.— 40 c.— Режим доступа: http://www.iprbookshop.ru/21325.— ЭБС «IPRbooks», по паролю

8. Фарафонов А.С. Программирование на языке высокого уровня [Электронный ресурс]: методические указания к проведению лабораторных работ по курсу «Программирование»/ Фарафонов А.С.— Электрон. текстовые данные.— Липецк: Липецкий государственный технический университет, ЭБС АСВ, 2013.— 32 c.— Режим доступа: http://www.iprbookshop.ru/22912.— ЭБС «IPRbooks», по паролю