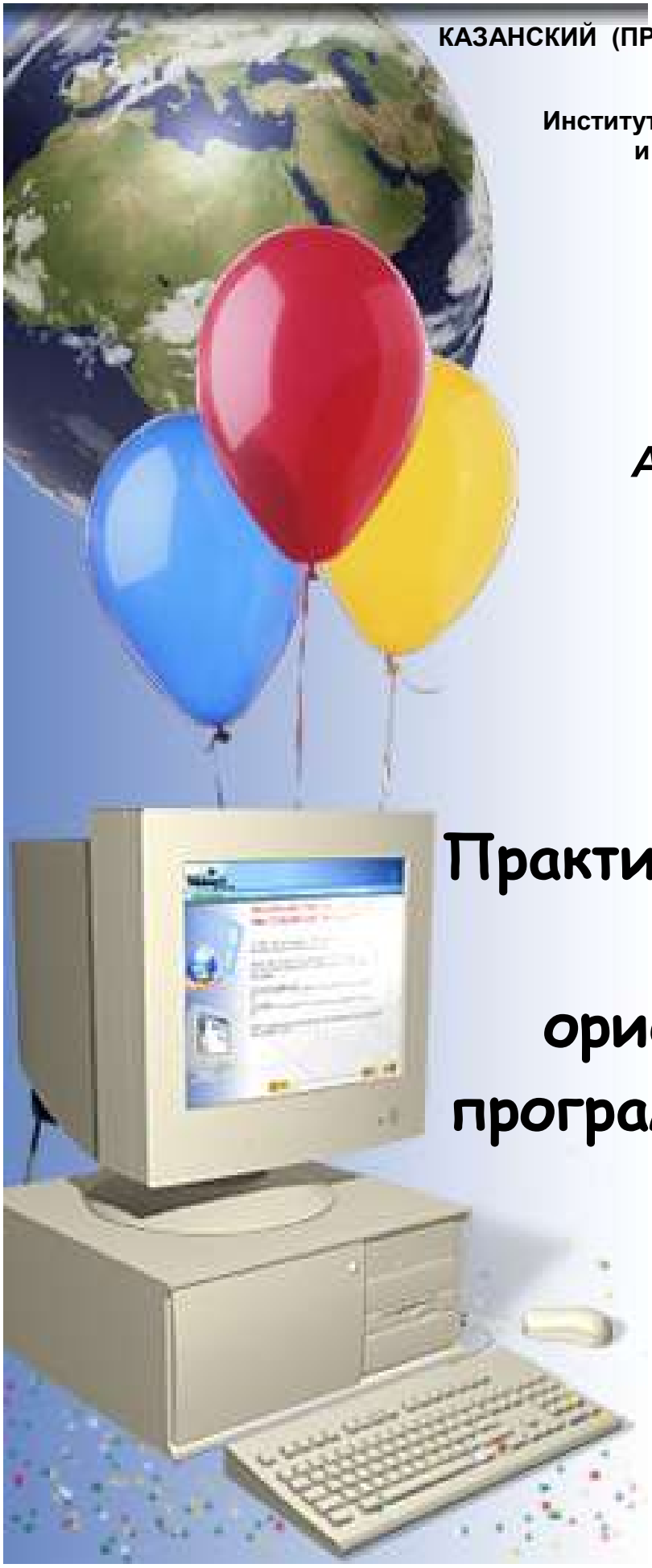
КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт Вычислительной Математики и Информационных Технологий



АНДРИАНОВА А.А.,
ИСМАГИЛОВ Л.Н.,
МУХТАРОВА Т.М.

Практикум по курсу «Объектно-ориентированное программирование» на языке C#

КАЗАНЬ - 2012

УДК 004.43

ББК 32.973.26 – 018.1

Печатается по постановлению редакционно-издательского совета Института вычислительной математики и информационных технологий Казанского (Приволжского) федерального университета

Рецензенты:

кандидат технических наук, доцент кафедры систем информационной безопасности КНИТУ им. А.Н. Туполева **Л.А. Александрова**

старший преподаватель кафедры системного анализа и информационных технологий Казанского (Приволжского) федерального университета **Р.Р. Тагиров**

Андрианова А.А., Исмагилов Л.Н., Мухтарова Т.М.
Практикум по курсу «Объектно-ориентированное программирование» на языке C#: Учебное пособие / А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. – Казань: Казанский (Приволжский) федеральный университет, 2012. – 115 с.

Учебное пособие предназначено для ведения практических занятий на 1, 2 курсах направлений «Прикладная информатика», «Прикладная математика и информатика», «Бизнес-информатика» в целях получения студентами практических навыков разработки объектно-ориентированных программ на языке программирования C#. Также пособие может использоваться студентами иных специальностей, интересующихся программированием.

Оглавление

Введение

Учебное пособие является руководством для ведения практических занятий при изучении студентами объектно-ориентированного программирования на языке C#. В пособии проиллюстрировано применение основных принципов объектно-ориентированного программирования (абстрагирования, инкапсуляции, наследования и полиморфизма) на разнообразных примерах. Помимо этого большое внимание уделено различным возможностям языка программирования C# в рамках объектно-ориентированного подхода. Этот язык программирования является сейчас одним из наиболее распространенных средств разработки приложений. Он был разработан в 1998—2001 годах группой инженеров под руководством Андерса Хейлсберга в компании Microsoft как основной язык разработки приложений для платформы Microsoft .NET Framework.

Основной акцент при создании объектно-ориентированных программ ставится на описании набора взаимодействующих друг с другом объектов, которые относятся к различным классам приложения (пользовательским и библиотечным). Поэтому разработка объектно-ориентированной программы– это определение основных объектов задачи, их структурных и поведенческих характеристик и принципов их взаимодействия друг с другом. Именно этот процесс на примерах различной сложности и предметной направленности подробно описывается в данном учебном пособии.

Данный практикум может использоваться совместно с учебным пособием тех же авторов «Объектно-ориентированное программирование на C#», являясь его практическим сопровождением.

Учебное пособие состоит из шести разделов, каждый из которых посвящен разбору одного или нескольких примеров.

В первом разделе рассматриваются принципы абстрагирования и инкапсуляции на примере создания класса работы с рациональными числами. Особое внимание в этом примере уделено переопределению операций с такими числами. Также демонстрируется создание метода-обобщения, который может работать как с целыми или вещественными числами, так и с рациональными числами, класс которых определен в этом разделе.

Второй раздел посвящен такой возможности языка C# как делегаты. Делегат – это объект, который может хранить информацию о методах. Применение делегатов демонстрируется на примере решения задачи поиска корня уравнения на заданном отрезке с помощью метода деления отрезка пополам, метода хорд и метода касательных.

В третьем разделе описывается получение решения системы линейных уравнений. На данном примере демонстрируется взаимодействие объектов различных классов (матрица, система уравнений) и разнообразные способы представления и решения системы уравнений.

В четвертом разделе впервые затрагивается принцип наследования. Использование этого принципа будет продемонстрировано на примере создания интерфейса математических объектов и раскрытия этого интерфейса на примерах классов «Матрица» и «Полином». С помощью обобщенной функции показывается, как единообразно можно осуществлять обработку объектов-потомков одного родителя, не анализируя их тип.

В пятом разделе рассматривается принцип полиморфизма позднего связывания на примере создания иерархии классов, определяющих различные кривые на плоскости, и применения этой иерархии для построения множества точек, удовлетворяющих некоторой системе ограничений.

Шестой раздел посвящен работе с коллекциями – классами, которые позволяют хранить различным образом организованные структуры данных и использовать их при решении прикладных задач. В данном разделе приводится как описание понятия коллекции, так и несколько примеров, которые основываются на использовании этих классов – стеков и очередей, списков, словарей. В одном из примеров демонстрируются возможности языка запросов LINQ, который предназначен для выборки данных из коллекций.

В конце разделов приведен список задач для самостоятельного решения, с помощью которых студенты могут закрепить навыки, приобретенные при изучении соответствующего раздела. Программы, приведенные в пособии, разработаны и отлажены в интегрированной оболочке проектирования Microsoft Visual Studio 2010.

1. Класс «Рациональное число»

Рациональное число (лат. *ratio* – отношение, деление, дробь) – число, представляемое обыкновенной дробью $\frac{m}{n}$, где *m, n* – целые числа.

Правильной называется дробь, у которой модуль числителя меньше модуля знаменателя. Правильные дроби представляют рациональные числа, принадлежащие интервалу (-1, 1). Дробь, не являющаяся правильной, называется неправильной. У нее модуль числителя больше или равен модулю знаменателя.

Неправильную дробь можно представить в виде суммы целого числа и правильной дроби. Такая запись числа называется смешанной дробью.

В качестве примера разберем создание класса «Рациональное число», который должен реализовывать стандартные операции над числами: сложение, вычитание, умножение, деление и операции сравнения. В классе также необходимо предусмотреть средства приведения дроби к смешанному виду.

1.1. Переменные и методы класса «Рациональное число»

Из определения следует, что любое рациональное число в смешанном виде определяется четырьмя составляющими:

- знаком числа (число положительное или отрицательное);
- целой частью;
- числителем;
- знаменателем.

Все составляющие дроби являются целыми числами. Знак дроби тоже будем представлять в виде целого числа (1 – положительная дробь, -1 – отрицательная дробь), поскольку это удобно при реализации арифметических операций:

```
// класс "Рациональное число"
class Fraction
{
   int sign;                  // знак дроби (+ или -)
   int intPart;               // целая часть дроби
   int numerator;             // числитель дроби
   int denominator;           // знаменатель дроби
    . . .
}
```

При описании операций с дробями предполагаем, что объекты класса `Fraction` находятся в смешанном виде. Результатом операции над дробями может быть неправильная дробь, которую, согласно предположению, необходимо перевести в смешанный вид. Для этого необходимы методы «преобразования в смешанный вид», «сокращения дроби» и «выделения целой части». Данные методы будут применяться при выполнении арифметических операций над дробями или при создании дроби, гарантируя, что дробь после завершения операции будет иметь смешанный вид. Таким образом, пользователю класса нет необходимости выполнять операции приведения дроби к смешанному виду, поскольку эта операция выполняется автоматически. Поэтому методы преобразования в смешанный вид, сокращения дроби и выделения целой части можно описать как закрытые элементы класса.

```
// класс "Рациональное число"
class Fraction
{
   int sign;                  // знак дроби (+ или -)
   int intPart;               // целая часть дроби
   int numerator;             // числитель дроби
   int denominator;           // знаменатель дроби


   // метод преобразование дроби в смешанный вид
   void GetMixedView()
   {
        . . .
   }

   // метод сокращения дроби
    void Cancellation()
   {
        . . .
   }
```

```
    // метод выделения целой части дроби
    void GetIntPart()
    {
        . . .
    }
    . . .
}
```

Деструктор также должен относиться к закрытым элементам класса – это определено правилами языка C#.

К доступным элементам класса `Fraction` относятся конструкторы, методы, реализующие арифметические операции, методы сравнения, метод преобразования в вещественное число. Для ввода дроби необходимо разработать статический метод `Parse` для выделения дроби из символьной строки. Для вывода дроби удобно переопределить неявную операцию преобразования в символьную строку, которая будет осуществлять получение символьного представления дроби.

Таким образом, полный состав класса `Fraction` может выглядеть так:

```
class Fraction
{
    int sign;                   // знак дроби (+ или -)
    int intPart;                // целая часть дроби
    int numerator;              // числитель дроби
    int denominator;            // знаменатель дроби

    // метод преобразования дроби в смешанный вид
    void GetMixedView()
    {
        . . .
    }

    // метод сокращения дроби
    void Cancellation()
    {
        . . .
    }

    // метод выделения целой части дроби
    void GetIntPart()
    {
        . . .
    }
```

```
// конструктор без параметров
public Fraction()
{
    . . .
}

// конструктор с параметрами
public Fraction(int n, int d, int i = 0, int s = 1)
{
    . . .
}
// деструктор
~Fraction()
{
    . . .
}

// метод сложения двух дробей
static public Fraction operator + (Fraction ob1, Fraction ob2)
{
    . . .
}

// метод сложения дроби с целым числом
static public Fraction operator + (Fraction ob1, int a)
{
    . . .
}

// метод сложения целого числа и дроби
static public Fraction operator + (int a, Fraction ob1)
{
    . . .
}

// метод изменение знака дроби на противоположный
static public Fraction operator - (Fraction ob)
{
    . . .
}

// метод вычитания двух дробей
static public Fraction operator - (Fraction ob1, Fraction ob2)
{
    . . .
}

// метод вычитания из дроби целого числа
static public Fraction operator - (Fraction ob1, int a)
```

```
{
    . . .
}

// метод вычитания дроби из целого числа
static public Fraction operator - (int a, Fraction ob1)
{
    . . .
}


// метод умножения двух дробей
static public Fraction operator * (Fraction ob1, Fraction ob2)
{
    . . .
}

// метод умножения дроби на целое число
static public Fraction operator * (Fraction ob1, int a)
{
    . . .
}

// метод умножения целого числа и дроби
static public Fraction  operator * (int a, Fraction ob1)
{
    . . .
}

// метод деления двух дробей
static public Fraction operator / (Fraction ob1, Fraction ob2)
{
    . . .
}

// метод деления дроби на целое число
static public Fraction operator / (Fraction ob1, int a)
{
    . . .
}

// метод деления целого числа на дробь
static public Fraction operator / (int a, Fraction ob1)
{
    . . .
}

// метод преобразования дроби в тип double
static public explicit operator double(Fraction ob)
{
    . . .
```

```
    }

    // методы сравнения двух дробей
    public static bool operator > (Fraction ob1, Fraction ob2)
    {
        . . .
    }



    public static bool operator < (Fraction ob1, Fraction ob2)
    {
         . . .
    }

    public static bool operator >= (Fraction ob1, Fraction ob2)
    {
       . . .
    }

    public static bool operator <= (Fraction ob1, Fraction ob2)
    {
         . . .
    }

    public static bool operator != (Fraction ob1, Fraction ob2)
    {
        . . .
    }

    public static bool operator == (Fraction ob1, Fraction ob2)
    {
        . . .
    }

    // статический метод преобразования строки в дробь
    public static Fraction Parse(string str)
    {
        . . .
    }

    // метод получения строкового представления дроби – оператор
    // преобразования в символьную строку
    public static implicit operator string(Fraction ob)
    {
       . . .
    }
}
```

1.2. Конструкторы и деструктор класса «Рациональное число»

Для создания объекта определим конструктор с четырьмя параметрами, соответствующими четырем структурным элементам класса:

- значение числителя;
- значение знаменателя;
- значение целой части;
- знак числа.

Прототип конструктора имеет следующий вид:

```
// конструктор с параметрами
public Fraction(int n, int d, int i = 0, int s = 1)
```

Если при создании объекта не указываются значения целой части и знака, то по умолчанию считается, что целая часть числа равна нулю и число является положительным. Это определяется заданием значений по умолчанию соответствующих параметров конструктора в его объявлении (прототипе). Параметры, для которых указываются значения по умолчанию, должны располагаться в конце списка формальных параметров.

```
//конструктор класса «Рациональное число»
public Fraction(int n, int d, int i = 0, int s = 1)
{
   intPart = i;
    numerator = n;
    denominator = d;
    sign = s;
    GetMixedView();
}
```

При создании объекта конструктору могут быть переданы значения числителя и знаменателя, образующие неправильную или сократимую дробь. В этом случае в теле конструктора после инициализации свойств нужно преобразовать дробь в смешанный вид. Это можно сделать путем вызова метода преобразования `GetMixedView()`.

Также определим в классе конструктор без параметров, который может использоваться при создании дроби, равной нулю. В

конструкторе без параметров структурным свойствам присваиваются конкретные значения:

```
// конструктор без параметров класса «Рациональное число»
public Fraction()
{
   intPart = 0;
   numerator = 0;
    denominator = 1;
    sign = 1;
}
```

Отдельно рассмотрим метод преобразования дроби в смешанную и несократимую форму. В случаях, если значения числителя и знаменателя задают неправильную или сократимую дробь, в методе происходит выделение целой части, а затем осуществляется сокращение.

```
// метод преобразования дроби в смешанный вид
void GetMixedView()
{
   GetIntPart();        // выделение целой части числа
   Cancellation();      // сокращение дроби
}
```

Если числитель дроби больше знаменателя, то выделяется целая часть:

```
// метод выделения целой части рационального числа
void GetIntPart()
{
   if(numerator >= denominator)
    {
       intPart += (numerator / denominator);
       numerator %= denominator;
    }
}
```

Сокращение дроби осуществляется путем деления числителя и знаменателя дроби на их наибольший общий делитель, который вычисляется с помощью алгоритма Евклида.

```
// метод сокращения рациональной дроби
void Cancellation()
{
   if(numerator != 0)
    {
```

```
            int m = denominator,
                n = numerator,
                ost = m%n;
            // вычисление НОД(числителя, знаменателя)
            // алгоритмом Евклида
            while(ost != 0)
            {
               m = n;
                n = ost;
                ost = m % n;
            }
            int nod = n;
            if(nod != 1)
            {
               numerator /= nod;
                denominator /= nod;
            }
        }
    }
```

Деструктор класса выводит сообщение о том, что уничтожен объект класса `Fraction`.

```
    // деструктор
    ~Fraction()
    {
        Console.WriteLine("Дробь " + this + " уничтожена.");
    }
```

Далее в функции `Main()` приведены различные способы создания объектов класса `Fraction` с помощью конструкторов.

```
    static void Main(string[] args)
    {
        // создание дроби 2/3
        Fraction d1 = new Fraction(2, 3, 0, 1);
        // создание дроби -2 4/5
        Fraction d2 = new Fraction(4, 5, 2, -1);
        // создание дроби 2 1/3
        Fraction d3 = new Fraction(4, 3, 1, 1);
        // создание дроби 1 2/3
        Fraction d4 = new Fraction(10, 6);
        // создание дроби 3/7
        Fraction d5 = new Fraction(3, 7);
        // создание дроби 2 3/8
        Fraction d6 = new Fraction(3, 8, 2);
        // создание рационального числа 0
        Fraction d7 = new Fraction();
        . . .
    }
```

1.3. Перегрузка операций для класса «Рациональное число»

Для использования знаков арифметических операций и операций сравнения перегрузим соответствующие операторы.

Поскольку любая дробь является вещественным числом, переопределим оператор явного преобразования объекта класса `Fraction` к вещественному типу данных `double`:

```
// операция преобразования дроби в тип double
public static explicit operator double(Fraction ob)
{
   double res = (double)ob.sign*(ob.intPart * ob.denominator +
                         ob.numerator) / ob.denominator;
   return res;
}
```

Данное преобразование удобно будет использовать при сравнении дробей.

Перегрузку операций сравнения двух дробей (больше, больше или равно, меньше, меньше или равно, равно, не равно) осуществим с помощью статических методов класса `Fraction`. Эти операторы должны возвращать значение типа `bool`. Заметим, что эти операторы следует перегружать «парами». Например, если класс содержит перегруженную операцию ”==”, то обязательно в нем должна быть перегружена и операция ”!=”.

Операторы ”==” и ”!=” осуществляют поэлементное сравнение двух дробей, которые представлены в смешанном виде. Остальные операторы сравнения используют преобразование дроби к вещественному числу и сравнивают полученные значения.

```
// операции сравнения двух дробей
public static bool operator == (Fraction ob1, Fraction ob2)
{
   if (ob1.sign != ob2.sign || ob1.intPart != ob2.intPart ||
             ob1.numerator * ob2.denominator !=
                              ob1.denominator * ob2.numerator)
      return false;
    return true;
}
```

```
public static bool operator != (Fraction ob1, Fraction ob2)
{
   if (ob1.sign == ob2.sign && ob1.intPart == ob2.intPart &&
            ob1.numerator * ob2.denominator ==
                              ob1.denominator * ob2.numerator)
      return false;
   return true;
}

public static bool operator > (Fraction ob1, Fraction ob2)
{
   if ((double)ob1 <= (double)ob2)
      return false;
    return true;
}

public static bool operator < (Fraction ob1, Fraction ob2)
{
   if ((double)ob1 >= (double)ob2)
      return false;
    return true;
}

public static bool operator >= (Fraction ob1, Fraction ob2)
{
   if ((double)ob1 < (double)ob2)
      return false;
    return true;
}

public static bool operator <= (Fraction ob1, Fraction ob2)
{
   if ((double)ob1 > (double)ob2)
      return false;
    return true;
}
```

Каждая арифметическая операция (“+”, “–“, “*”, ”/”) перегружена тремя методами-операторами для случаев, когда:

- операндами операции являются объекты класса `Fraction`;
- первый операнд – дробь, второй – целое число;
- первый операнд – целое число, второй – объект-дробь.

Результатом выполнения этих операторов является новая дробь. Рассмотрим подробнее реализацию операторов сложения.

Оператор сложения двух дробей после формирования результата осуществляет преобразование к смешанному виду.

```
// операция сложения двух дробей
static public Fraction operator + (Fraction ob1, Fraction ob2)
{
   Fraction res=new Fraction();
   res.numerator = ob1.sign * (ob1.intPart * ob1.denominator +
        ob1.numerator) * ob2.denominator + ob2.sign *(ob2.intPart
  *
        ob2.denominator + ob2.numerator) * ob1.denominator;
   res.denominator = ob1.denominator * ob2.denominator;
   if (res.numerator < 0)
   {
      res.numerator *= -1;
      res.sign = -1;
   }
   res.GetMixedView();
   return res;
}
```

В определении функции сложения дроби с целым числом осуществляется преобразование этого целого числа в дробь (создается новый объект класса `Fraction`, значение которого равно целому числу) и вызов оператора сложения двух дробей.

```
// метод сложения дроби с целым числом
static public Fraction operator + (Fraction ob1, int a)
{
   // если к дроби прибавляется число, равное 0,
   // результат совпадает с операндом-дробью
   if (a == 0)
      return new Fraction(ob1.numerator, ob1.denominator,
                                    ob1.intPart, ob1.sign);
   // создание новой дроби ob2 = a
   Fraction ob2=new Fraction(0, 1, Math.Abs(a), a/Math.Abs(a));

   Fraction res = ob1 + ob2;       //сложение двух дробей
   return res;
}

// метод сложения целого числа и дроби
static public Fraction operator + (int a, Fraction ob1)
{
   // если к дроби прибавляется число, равное 0,
   // результат совпадает с операндом-дробью
   if (a == 0)
      return new Fraction(ob1.numerator, ob1.denominator,
                                    ob1.intPart, ob1.sign);
   // создание новой дроби ob2 = a
   Fraction ob2=new Fraction(0, 1, Math.Abs(a), a/Math.Abs(a));

   Fraction res = ob1 + ob2;       //сложение двух дробей
```

```
        return res;
    }
```

Аналогичным образом определяют и другие арифметические операции.

Обсудим особенности ввода и вывода рациональных дробей. Поскольку при вводе пользователь задает символьную строку, представляющую дробь, то задача ввода дроби сводится к созданию метода преобразования символьной строки в дробь. При выводе происходит противоположная операция. Поэтому для организации вывода в класс `Fraction` введем операцию неявного преобразования дроби в тип `string`, а для ввода – статический метод `Parse()` формирования объекта-дроби из строки.

Для представления дроби пользователю удобно использовать ее привычный математический вид с учетом существования целой и/или дробной части. Именно такой вид формируется в операции преобразования дроби в строку.

```
// операция получения строкового представления дроби
static public implicit operator string(Fraction ob)
{
   string res = "";
   // знак числа выводится, только если число отрицательно
   if (ob.sign < 0)
      res = res + "-";
   // если целая часть не равна 0, выводим ее
   if (ob.intPart != 0)
      res = res + ob.intPart;
   // дробная часть печатается, если числитель не равен 0
   if (ob.numerator != 0)
      res = res + " " + ob.numerator + "/" + ob.denominator;
   // если и целая часть и дробная часть равны 0,
   // то число равно 0
   if (ob.intPart == 0 && ob.numerator == 0)
      res = "0";
   return res;
}
```

В том же формате будет осуществляться и ввод данных из символьной строки. В методе `Parse()` производится выделение составных частей числа – знака, целой части, числителя и знаменателя.

```
// метод получения дроби из строки
public static Fraction Parse(string str)
```

```
{
    int intPart, numerator, denominator, sign;
    // разделение строки на подстроки
    // с помощью разделителя-пробела
    string [] strs=str.Split(' ');
    string[] strs1;
    Fraction res;
    if (strs.Length == 1)
    {
        // в строке не было найдено пробелов
        // производим разделение строки по символу ‘/’
        strs1 = str.Split('/');
        if (strs1.Length == 1)
        {
            // число задано в виде только целой части
            // выделяем целую часть
            intPart = int.Parse(strs1[0]);
            // в зависимости от значения целой части,
            // формируем новую дробь
            if (intPart!=0)
                res = new Fraction(0, 1, Math.Abs(intPart),
                                   intPart / Math.Abs(intPart));
            else
                res = new Fraction(0, 1, Math.Abs(intPart), 1);
            return res;
        }
        else
        {
            // число задано в виде только дробной части
            // выделяем отдельно числитель и знаменатель
            numerator = int.Parse(strs1[0]);
            denominator = int.Parse(strs1[1]);
            sign = 1;
            // определяем знак числа по знаку числителя
            if (numerator < 0)
            {
                numerator = -numerator;
                sign = -1;
            }
            // формируем новую дробь и приводим ее
            // к несократимому виду
            res = new Fraction(numerator, denominator, 0, sign);
            res.GetMixedView();
            return res;
        }
    }
    // дробь задана в смешанном виде
    // отделяем дробную часть по разделителю ‘/’
    strs1 = strs[1].Split('/');
    intPart = int.Parse(strs[0]);
    // определяем знак числа по знаку целой части
```

```
        if (intPart < 0)
        {
           intPart = -intPart;
           sign = -1;
        }
        else
           sign = 1;
        numerator = int.Parse(strs1[0]);
        denominator = int.Parse(strs1[1]);
        // формируем новую дробь и приводим ее
        // к несократимому виду
        res = new Fraction(numerator, denominator, intPart, sign);
        res.GetMixedView();
        return res;
    }
```

Приведем пример использования объектов класса `Fraction` и операций работы с ними (Рис. 1.1).

```
    static void Main(string[] args)
    {
        // создание дроби 2/3
        Fraction r1 = new Fraction(2, 3, 0, 1);
        Console.WriteLine("r1 = " + r1);
        // создание дроби 5/7
        Fraction r2 = new Fraction(5, 7, 0, 1);
        Console.WriteLine("r2 = " + r2);
        Console.WriteLine("-r2 = " + -r2);
        Console.WriteLine("r2 = " + (double)r2);
        // вызов оператора "==" для двух дробей
        if (r1 == r2)
          Console.WriteLine("r1 == r2");
        else
          Console.WriteLine("r1 != r2");
        // вызов оператора ">" для двух дробей
        if (r1 > r2)
          Console.WriteLine("r1 > r2");
        else
          Console.WriteLine("r1 <= r2");
        // вызов оператора "<=" для двух дробей
        if (r1 > r2)
          Console.WriteLine("r1 <= r2");
        else
          Console.WriteLine("r1 > r2");
        // вызов оператора "+" для двух дробей
        Fraction d = r1 + r2;
        Console.WriteLine("r1 + r2 = " + d);
        // вызов оператора "+" для дроби и числа
        d = r1 + (-11);
        Console.WriteLine("r1 + (-11) = " + d);
        // вызов оператора "+" для числа и дроби
```

```
        d = 5 + r1;
        Console.WriteLine("5 + r1 = " + d);
        . . .
}
```

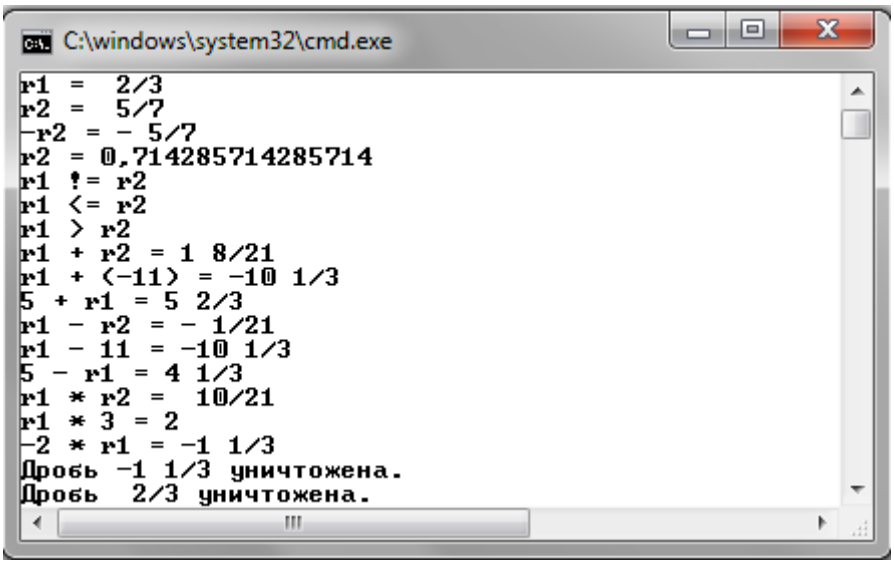


Рис.1.1. Демонстрация операций с классом `Fraction`.

1.4. Быстрая сортировка массива дробей

Тип `Fraction` может использоваться в качестве подставляемого вместо обобщенного типа данных при генерации функций и классов на основе обобщений. Приведем пример использования обобщения функции быстрой сортировки для массива дробей. В сгенерированном методе с типом `Fraction` требуется организовать сравнение двух дробей. Для этого класс `Fraction` должен раскрывать интерфейс `IComparable` и иметь метод сравнения дробей `CompareTo()`.

```
class Fraction : IComparable
{
    int sign;             // знак дроби (+ или -)
    int intPart;          // целая часть дроби
    int numerator;        // числитель дроби
    int denominator;      // знаменатель дроби

    // метод сравнения двух дробей
    public int CompareTo(object ob)
    {
        if (this < (ob as Fraction)) return -1;
        if (this > (ob as Fraction)) return 1;
        return 0;
    }
```

```
    . . .
}
```

Определим обобщенный метод быстрой сортировки, указав ограничение на использование в качестве обобщенных типов только тех, которые раскрывают интерфейс `IComparable`. Пусть этот метод является статическим для главного класса приложения `Program`.

```
class Program
{
   static void QuickSort<T>(T [] m, int l, int r)
                                    where T: IComparable
   {
       if(l == r)
          return;
       int i = l, j = r;
       // выбирается элемент, делящий массив на две части
       T selected = m[l];
       // осуществляется поиск и перестановка элементов, меньших
       // выбранного, с конца массива и больших выбранного
      // с начала массива.
       while(i != j)
       {
          while(m[j].CompareTo(selected) >= 0 && j > i)
            j--;
          if(j > i)
          {
             m[i] = m[j];
             while(m[i].CompareTo(selected) <= 0 && i < j)
                i++;
             m[j] = m[i];
          }
       }
       // выбранный элемент устанавливается на надлежащее место
       m[i] = selected;
       // если существуют элементы слева от выбранного,
       // сортируем эту часть массива
       if(l <= i-1)
          QuickSort(m, l, i-1);
       // то же проводится с правой частью массива,
       // если она существует
       if( i+1 < r)
          QuickSort(m , i+1, r);
}

static void Main(string[] args)
{
   Fraction[] a = { new Fraction(4,2,3,1),
                    new Fraction(1,6,5,-1),
```

```
                    new Fraction(0,2,7,1),
                    new Fraction(10,2,3,-1),
                    new Fraction(2,13,20,1),
                    new Fraction(0,1),
                    new Fraction(1,1,3,1),
                    new Fraction(2,5,7,-1),
                    new Fraction(4,8,2,-1),
                    new Fraction(4,1,3,-1)};
        for (int i = 0; i < a.Length; i++)
           Console.Write(a[i]+" ");
        Console.WriteLine();
        // генерация и вызов функции быстрой сортировки
        // для массива дробей
        QuickSort(a, 0, 9);
        for (int i = 0; i < a.Length; i++)
           Console.Write(""+a[i]+" ");
        Console.WriteLine();
       . . .
  }
```

Это же обобщение можно применять и для массива чисел типов `int`, `double`, а также для массивов, элементами которых являются объекты классов (пользовательских или библиотечных), которые раскрывают интерфейс `IComparable`.

Задания для самостоятельной работы

1. Дополнить класс `Fraction` перегруженными арифметическими операциями, в которых один из операндов является вещественным числом.
2. Дополнить класс `Fraction` перегруженными операциями сравнения для дробей и вещественных чисел.
3. Дополнить класс `Fraction` перегруженным конструктором, осуществляющим преобразование вещественного числа к типу `Fraction`. Предполагается, что дробная часть вещественного числа содержит до `10` знаков после запятой.
4. Разработать класс «Комплексное число». Определить в нем конструктор, перегрузить арифметические операции, операции сравнения, операцию преобразования в строку и статический метод получения комплексного числа из строки.
5. Разработать класс «Комплексное число в тригонометрической форме». Определить в нем конструктор,

перегрузить арифметические операции, операции сравнения, операцию преобразования в строку и статический метод получения комплексного числа из строки.

6. Разработать класс «Комплексное число», в котором данные хранятся в двух видах: алгебраической и тригонометрической формах. Определить в нем конструкторы и деструктор, перегрузить арифметические операции, операции сравнения, операцию преобразования в строку и статический метод получения комплексного числа из строки, написать методы преобразования числа из одной формы в другую. Протестировать все возможности класса.

7. Разработать класс «Дата». Определить в нем конструкторы и деструктор, перегрузить операцию добавления к дате заданного количества дней, операцию вычитания двух дат, операции сравнения и операцию преобразования в символьную строку, а также статический метод получения даты из строки.

8. Разработать класс «Время». Определить в нем конструкторы и деструктор, перегрузить операцию добавления к времени заданного количества минут, операцию вычитания двух моментов времени, операцию преобразования в символьную строку и метод получения момента времени из строки.

9. Разработать класс «Прямоугольник». Определить в нем конструкторы и деструктор, перегрузить операцию пересечения прямоугольников (операция “*”), операцию вычисления площади прямоугольника операции сравнения (по площади), операцию преобразования в символьную строку и метод получения объекта-прямоугольника из строки.

10. Разработать класс «Треугольник». Определить в нем конструкторы и деструктор, перегрузить операцию преобразования в вещественное число (площадь треугольника), операцию проверки включения точки в треугольник, операции сравнения треугольников (по площади), операцию преобразования в символьную строку и метод получения объекта-треугольника из строки.

2. Методы решения уравнений

Решим следующую математическую задачу. Требуется найти корень уравнения вида $f(x) = 0$ на отрезке $[a, b]$ с точностью ε (точность по функционалу), т.е. нужно найти точку $x \in [a, b]$, для которой выполняется неравенство $|f(x)| \leq \varepsilon$.

Для решения этой задачи существует целый ряд различных методов, самыми известными из которых являются метод деления отрезка пополам, метод хорд и метод касательных.

Для применения этих методов необходимо, чтобы:

1) функция $f(x)$ являлась непрерывной,

2) значения функции $f(x)$ на концах отрезка имели противоположные знаки (тогда гарантируется, что корень на отрезке существует);

3) для метода касательных обязательно требуется, чтобы функция $f(x)$ была выпуклой или вогнутой на отрезке.

Все три метода нахождения корня основываются на единой процедуре.

1. Некоторым образом выбирается точка $c \in [a, b]$.

2. Выбирается один из отрезков $[a, c]$ или $[c, b]$, на котором далее будет производиться поиск корня. Должен быть выбран тот из отрезков, на котором в точках-концах функция принимает значения различных знаков:

если $f(a) * f(c) < 0$, то $b = c$, иначе $a = c$.

3. Если на новом отрезке $|f(a)| \leq \varepsilon$, то a – приближенное значение корня. Если же $|f(b)| \leq \varepsilon$, то b – приближенное значение корня.

Отличия метода деления отрезка пополам, метода хорд и метода касательных заключаются в реализации пункта 1 указанной процедуры. Так, метод деления отрезка пополам выбирает в качестве точки c середину отрезка $[a, b]$, метод хорд – точку пересечения отрезка, соединяющего точки $(a, f(a))$ и $(b, f(b))$, с осью абсцисс, метод касательных – точку пересечения одной из касательных к графику функции $f(x)$, построенных в точках a и b, с осью абсцисс (выбирается та из точек пересечения, которая принадлежит отрезку $[a, b]$).

На рис. 2.1 изображены варианты выбора следующего отрезка, на котором будет осуществлен поиск корня, когда точка c определена

методом хорд. На рис. 2.1 **а** приведено изменение левого конца отрезка (на следующей итерации точку *a* перенесем в точку *x*), на рис. 2.1 **б** – правого (точку *b* перенесем в точку *x*).

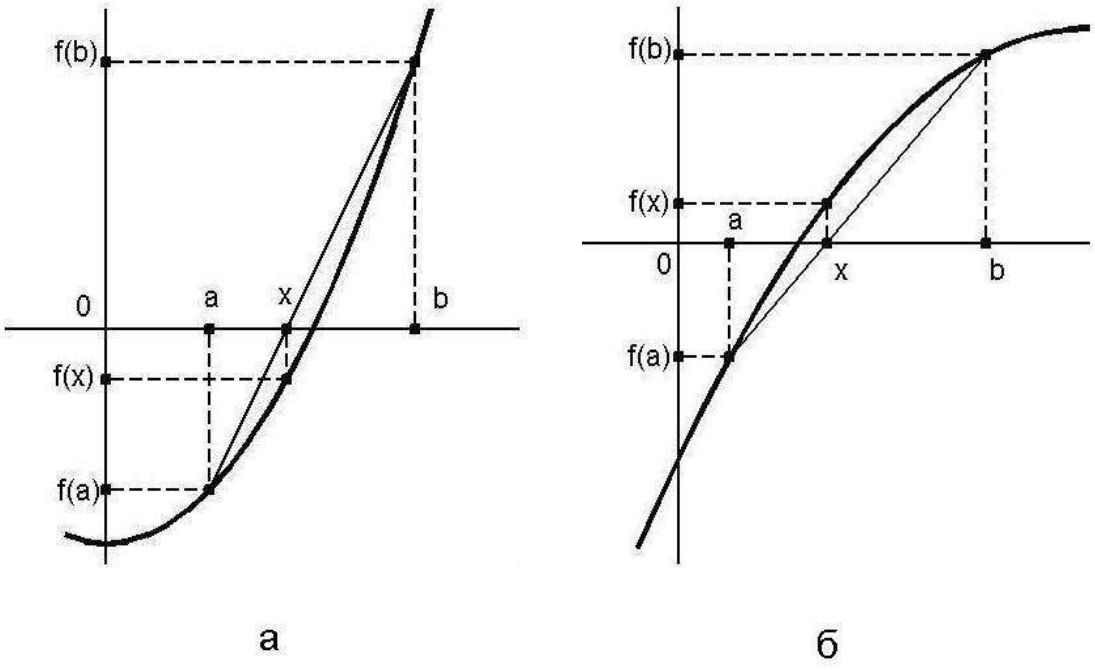


Рис. 2.1. Демонстрация одной итерации метода хорд.

При решении этой задачи будем использовать делегаты. Напомним, что делегаты – это классы, которые предназначены для хранения информации о каких-либо методах с одинаковым прототипом. Объект делегата становится псевдонимом хранимого метода (ссылкой на него). Как и любой объект, делегат можно передавать в функцию как параметр, он может являться полем класса или возвращаемым значением функции. Делегат также можно использовать для вызова хранимого метода в тот момент, когда это станет необходимо.

Создадим класс `Solver`, который будет решать уравнение. Пусть этот класс будет содержать только один открытый статический метод `RootEquation()` для решения уравнения. Остальные методы – методы определения точки-середины отрезка (`GetX1()`), точки пересечения хорды (`GetX2()`), и точки пересечения касательной с осью абсцисс (`GetX3()`), будут закрытыми, т.е. будут играть служебную роль. Выбор метода решения будет осуществляться случайным образом в статическом конструкторе.

Такая структура класса предусматривает использование делегатов в двух ситуациях:

1) Делегаты часто используются в качестве параметров других методов. В нашем случае уравнение будет задано посредством определения функции $f(x)$. Для хранения информации о том, как ее вычислять, создадим делегат:

```
// делегат для представления математической функции
delegate double Function(double x);
```

Экземпляр данного делегата может хранить информацию о любом методе, который получает в качестве параметра и возвращает вещественное число, например, таким методом может быть статический метод `Sin()` класса `Math`.

Таким образом, метод `RootEquation()` может получить указание как вычислять функцию $f(x)$ с помощью параметра типа указанного делегата.

2) Второй случай использования делегатов – настройка алгоритма решения задачи. Как уже было сказано, все три метода поиска корня основываются на единой процедуре, которая отличается только реализацией одного шага – выбора точки, разбивающей отрезок на два. Для реализации этого шага создадим в классе `Solver` три различных метода, информацию о которых будет хранить экземпляр следующего делегата:

```
// делегат для определения точки деления исходного отрезка
delegate double PointIn (Function f, double a, double b);
```

В классе `Solver` данный экземпляр будем хранить в виде статического поля `method` и инициализируем его в статическом конструкторе:

```
class Solver
{
   // делегат для определения точки деления отрезка
   static PointIn method;

   // статический конструктор
   static  Solver()
   {
      // инициализация генератора случайных чисел
      Random r = new Random();
      int q = r.Next(300)%3;
```

```
        switch (q)
        {
            case 0:
                // инициализация делегата функцией GetX1
                method = GetX1;
                Console.WriteLine("Выбран метод деления
                                            отрезка пополам");
                break;
            case 1:
                // инициализация делегата функцией GetX2
                method = GetX2;
                Console.WriteLine("Выбран метод хорд");
                break;
            case 2:
                // инициализация делегата функцией GetX3
                method = GetX3;
                Console.WriteLine("Выбран метод касательных");
                break;
        }
    }

    // получение середины отрезка
    static double GetX1(Function f, double a, double b)
    {
        return (a + b) / 2;
    }

    // получение точки пересечения хорды с осью OX
    static double GetX2(Function f, double a, double b)
    {
        return a - f(a) * (b - a) / (f(b) - f(a));
    }

    // получение точки пересечения касательной с осью OX
    static double GetX3(Function f, double a, double b)
    {
        // приближенное вычисление производной в точке a
        double fa = (f(a + 0.001) - f(a)) / 0.001;
        if (fa != 0)
        {
            // получение точки пересечения касательной
            // с осью абсцисс
            double c = a - f(a) / fa;
            // если точка c принадлежит отрезку,
            // выбираем ее
            if (a <= c && c <= b)
                return c;
        }

        // приближенное вычисление производной в точке b
        double fb = (f(b + 0.001) - f(b)) / 0.001;
```

```
            if (fb != 0)
            {
               // получение точки пересечения касательной
               // с осью абсцисс
               double c = b - f(b) / fb;
               if (a <= c && c <= b)
                  return c;
            }
            // функция не удовлетворяет тем свойствам,
            // которые гарантируют существование корня
            throw new Exception("Возможно, на этом
                                                отрезке корней нет");
         }
         . . .
}
```

Для решения уравнения достаточно иметь только одну функцию, которая реализует общую процедуру всех методов решения уравнений. Реализация пункта 1 процедуры в этой функции осуществляется посредством вызова через делегат `method` именно того метода деления отрезка, который был выбран при инициализации класса `Solver`. В указанном далее коде использование делегатов выделено.

```
   // метод решения уравнения
   // 1 параметр – делегат функции, которая определяет уравнение,
   // 2, 3 параметры - концы отрезка,
   // 4 параметр - точность решения
   static public double RootEquation(Function f, double a,
                                                double b, double eps)
   {
      // корень гарантировано существует,
      // если на концах отрезка
      // функция принимает значения различных знаков.
      if(f(a) * f(b) > 0)
         throw new Exception("Возможно, на этом отрезке корней нет");

      // цикл метода - вычисления продолжается до тех пор,
      // пока на одном из концов отрезка не будет получено
      // значение функции с заданной точностью
      while(Math.Abs(f(a))>eps && Math.Abs(f(b))>eps)
      {
         // поиск точки деления отрезка - вызов делегата
         double c = method(f,a,b);
         // если найденная точка - корень, это решение
         if(f(c) == 0)
            return c;
         // выбор следующего отрезка
         if(f(a)*f(c) < 0)
```

```
            b = c;
         else
            a = c;
      }
      // выбор приближенного значения корня
      if(Math.Abs(f(a))<=eps )
         return a;
      else
         return b;
   }
```

Использование разработанного класса для приближенного решения квадратного уравнения $x^2 - 2x - 5 = 0$ может быть таким:

```
class Program
{
   // функция определения уравнения
   static double func(double x)
   {
      return x * x - 2 * x - 5;
   }

   static void Main(string[] args)
   {
      Console.WriteLine("Введите отрезок");
      Console.WriteLine("Введите a");
      double a = double.Parse(Console.ReadLine());
      Console.WriteLine("Введите b");
      double b = double.Parse(Console.ReadLine());
      Console.WriteLine("Введите точность решения");
      double eps = double.Parse(Console.ReadLine());
      try
      {
         // вызов метода решения уравнения
         Console.WriteLine("Корень = {0}",
                     Solver.RootEquation(func, a, b, eps));
      }
      catch (Exception e)
      {
         Console.WriteLine(e.Message);
      }
   }
}
```

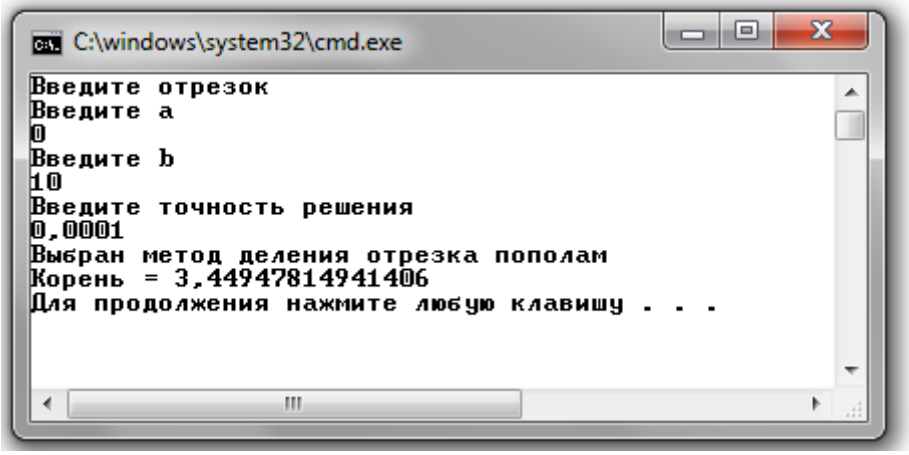


Рис 2.2. Демонстрация работы программы при выборе метода деления отрезка пополам

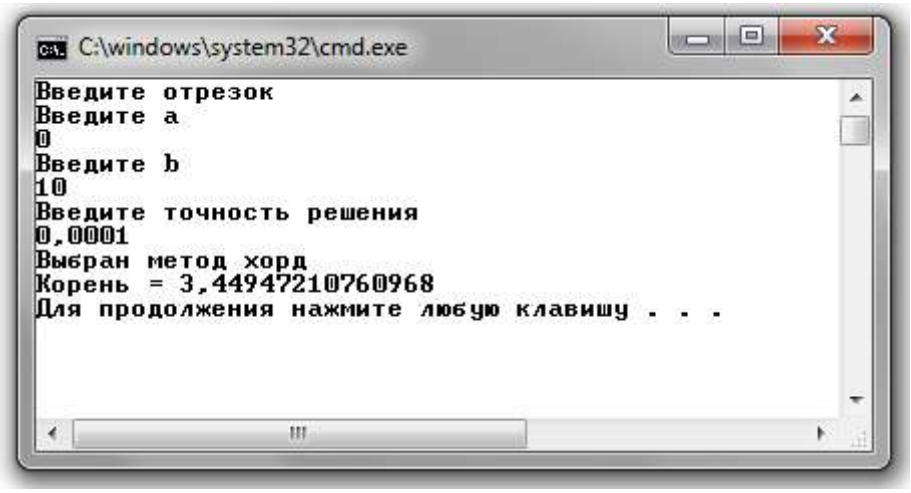


Рис 2.3. Демонстрация работы программы при выборе метода хорд

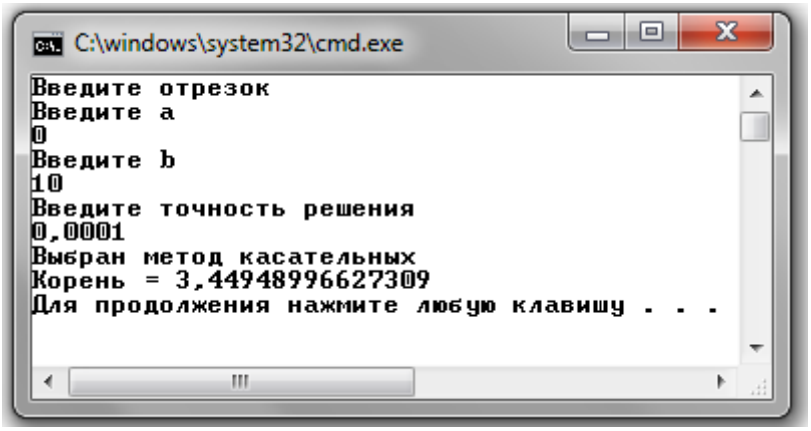


Рис 2.4. Демонстрация работы программы при выборе метода касательных

Задания для самостоятельной работы

1. Создать класс «Студент», который определяется полями ФИО, номер группы, название факультета, название специальности, средний балл успеваемости. Пусть имеется массив объектов этого класса. Разработать метод выбора студентов из массива по условию (учится на конкретном факультете, имеет средний балл более заданного уровня и пр.). Для определения, удовлетворяет ли объект условию, передать в метод параметр-делегат.

2. Для массива объектов класса «Студент» из задания 1 создать метод сортировки по различным критериям (по фамилии, по среднему баллу успеваемости). Метод сравнения двух объектов передать в метод сортировки как параметр-делегат.

3. Пусть имеется класс «Матрица». Определить различные методы, которые осуществляют преобразование матрицы (транспонирование, поворот, сортировка строк, изменение порядка столбцов на обратный и пр.). В диалоговом режиме задать последовательность действий, которую нужно произвести с объектом-матрицей, сохраняя ее в переменной-делегате. Предусмотреть команду меню выполнения этих действий, которая обращается к сформированному делегату.

4. Разработать класс для решения задачи поиска точки, минимизирующей функцию на отрезке. Реализовать функции решения этой задачи различными методами – методом деления отрезка пополам, методом золотого сечения, методом касательных, методом Фибоначчи. Выбранный способ решения сохранить в делегат.

3. Решение системы линейных уравнений

Одна из базовых задач линейной алгебры заключается в решении системы линейных алгебраических уравнений (СЛАУ). В общем случае система *m* линейных уравнений с *n* неизвестными имеет следующий вид:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + .. a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + .. a_{2n}x_n &= b_2, \\ . \quad . \quad . \quad . \quad . \\ a_{m1}x_1 + a_{m2}x_2 + .. a_{mn}x_n &= b_m. \end{aligned}$$

где a_{ij} – коэффициенты системы, b_i – свободные члены, x_j – неизвестные, $i = 1..m, j = 1..n$. Решением системы называется такая совокупность *n* чисел, которая при подстановке в систему на место неизвестных обращает все уравнения в тождества.

Систему линейных уравнений удобно представлять в матричном виде:

$$AX = B,$$

где $A = \{a_{ij}\}$ – матрица коэффициентов системы размерности *m* x *n*, $X = \{x_j\}$ – вектор-столбец неизвестных размера *n*, $B = \{b_i\}$ – вектор-столбец свободных членов размера *m*.

Система уравнений называется совместной, если она имеет хотя бы одно решение, и несовместной, если не имеет решений. Совместная система называется определенной, если она имеет единственное решение, и неопределенной, если существует несколько различных решений.

Если матрица *A* является квадратной, то количество решений системы уравнений определяется по значению определителя матрицы *A* (*det(A)*). Если определитель *det(A)* не равен 0, то система имеет единственное решение и получить его можно, к примеру, методом Крамера или по формуле $X = A^{-1}B$, где A^{-1} – обратная матрица к *A*. Если определитель *det(A)* равен 0, то система может не иметь решения или иметь бесконечное число решений. Решить систему уравнений в

этом случае можно, используя метод исключений Жордана-Гаусса. Когда система имеет бесконечное число решений, формируется общее решение СЛАУ, в котором значения одних переменных выражаются через значения других переменных.

Для решения системы уравнений с прямоугольной матрицей *A* можно применять метод исключений Жордана-Гаусса поиска общего решения системы.

Определим класс `Slau`, описывающий систему линейных уравнений и основные методы ее решения.

Структурными свойствами класса системы уравнений `Slau` являются:

- количество уравнений,
- количество неизвестных,
- матрица коэффициентов,
- вектор свободных членов,
- признак совместности системы,
- ранг матрицы коэффициентов,
- вектор решений.

Значения последних трех свойств определяются в процессе решения СЛАУ.

Поскольку вектор является частным случаем матрицы (вектор – матрица, состоящая из одного столбца), будем использовать для хранения как самой матрицы, так и вектора класс `Matrix`.

```
class Slau
{
   int m;          // количество уравнений
   int n;          // количество переменных
   Matrix a;       // матрица коэффициентов
   Matrix b;       // вектор правой части
   Matrix x;       // вектор решений
   bool isSolved;  // признак совместности
   int[] reoder;   // перестановка переменных,
                   // полученная в методе Жордана-Гаусса
   int rang;       // ранг матрицы коэффициентов
   . . .
 }
```

Помимо конструктора класса и методов ввода/вывода, класс `Slau` будет содержать методы решения системы линейных уравнений различными способами:

- метод Крамера,
- метод решения с помощью обратной матрицы,
- метод исключений Жордана-Гаусса.

```
// класс, определяющий систему линейных уравнений
class Slau
{
   int m;          // количество уравнений
   int n;          // количество переменных
   Matrix a;       // матрица коэффициентов
   Matrix b;       // вектор правой части
   Matrix x;       // вектор решений
   bool isSolved;  // признак совместности
   int[] reoder;   // перестановка переменных,
                   // полученная в методе Жордана-Гаусса
   int rang;       // ранг матрицы коэффициентов

   // конструктор
   public Slau(int m1, int n1)
   {
         . . .
   }

   // метод ввода системы уравнения
   public void Input()
   {
         . . .
    }

   // метод вывода системы уравнения и ее решения
    public void Print()
   {
        . . .
    }

    public void Solve()       // метод решения СЛАУ
   {
        . . .
    }

    public void Kramer()      // метод Крамера
   {
       . . .
   }

   public void InverseMatrix()     // метод X=A-1B
   {
       . . .
   }
```

```
        public void JordanGauss ()      // метод Жордана-Гаусса
        {
             . . .
         }

        // метод вывода решения СЛАУ
        public void PrintSolution()
        {
             . . .
        }
}
```

Сначала опишем функции решения системы линейных уравнений различными методами.

Метод Крамера используется для решения системы линейных уравнений с квадратной матрицей коэффициентов. Если матрица коэффициентов СЛАУ является неквадратной, будет генерироваться исключение с соответствующим сообщением. Для квадратной матрицы будет вычисляться определитель. Если определитель не равен нулю, единственное решение системы уравнений определяется по формуле Крамера:

$$x_j = \frac{\Delta_j}{\Delta},\ j = 1..n,$$

где $\Delta -$ определитель матрицы A, а Δ_j - определитель матрицы, в которой j-й столбец исходной матрицы заменен на вектор свободных членов. Если определитель матрицы (Δ) равен нулю, то будет сгенерировано еще одно исключение.

```
    // метод Крамера решения СЛАУ
    public void Kramer()
    {
        // проверка, является ли матрица прямоугольной
        if (m != n)
            throw new Exception("Матрица не является квадратной");
        double det = a.Determinant();   // вычисление определителя
                                        // матрицы коэффициентов
        // проверка определенности системы
        if (det == 0)
            throw new Exception("Деление на 0");

        rang = m;
        // вычисление корней по формулам Крамера
        Matrix temp = a.Copy();
```

```
        for (int j = 0; j < n; j++)
        {
            for (int i = 0; i < n; i++)
                temp[i, j] = b[0, i];
            x[0, j] = temp.Determinant / det;
            for (int i = 0; i < n; i++)
                temp[i, j] = a[i, j];
        }
        isSolved = true;
    }
```

В случае, если СЛАУ с квадратной матрицей A имеет единственное решение, его можно получить по формуле:

$$X = A^{-1}B, \text{ где } A^{-1} - \text{обратная матрица к } A.$$

Данная формула применима только в случаях, когда обратную матрицу можно вычислить (матрица A – квадратная, ее определитель не равен 0). Нарушение этих условий приводит к генерации исключений (генерация исключения равенства нулю определителя предусмотрена в методе вычисления обратной матрицы класса `Matrix`).

```
    // метод решения СЛАУ с помощью обратной матрицы
    public void InverseMatrix()
    {
        // проверка, является ли матрица прямоугольной
        if (m != n)
            throw new Exception("Матрица не является квадратной");
        // вычисление обратной матрицы
        Matrix obr = ~a;
        // поскольку для эффективного использования памяти
        // вектор хранится как строка, требуется получить
        // соответствующий вектор-столбец посредством
        // транспонирования
        Matrix B = !b;
        // получение решения СЛАУ
        x = obr * B;
        x = !x;
        rang = m;
        isSolved = true;
    }
```

Метод исключений Жордана-Гаусса может быть применен, как в ситуации, когда СЛАУ имеет единственное решение, так и когда решений бесконечно много. Проведение исключений всех строк по *i*-ой строке осуществляется по формулам Жордана-Гаусса:

$$a_{ij} = \frac{a_{ij}}{a_{ii}}, \quad j = 1..n$$

$$a_{kj} = a_{kj} - a_{ki}\frac{a_{ij}}{a_{ii}}, \quad j = 1..n, \quad k = 1..m, \quad k \neq i$$

$$b_i = \frac{b_i}{a_{ii}}$$

$$b_k = b_k - a_{ki}\frac{b_i}{a_{ii}}, \quad k = 1..m, \quad k \neq i$$

Эти формулы определяют эквивалентное преобразование СЛАУ, которое не меняет ее решение и позволяет определить ранг матрицы коэффициентов. Напомним, что ранг – это максимальный порядок минора матрицы, отличный от нуля. Посредством исключения нужно добиться, чтобы один из миноров, соответствующих рангу матрицы, занял положение в ее верхнем левом углу.

Преобразование осуществляется перестановкой строк и столбцов (переменных) матрицы. Согласно формулам Жордана-Гаусса исключение производится с помощью ненулевых элементов строк, расположенных в столбце с тем же номером. Если в строке с номером *i* такой элемент равен нулю, осуществляется поиск ненулевых элементов среди тех, которые расположены ниже в том же столбце. Если будет найден ненулевой элемент в *k*-ой строке, две строки с номерами *i* и *k* меняются местами (при этом изменяется порядок следования двух уравнений системы). Решение системы при этом не изменится и можно будет провести исключение по *i*-ой строке. Очевидно, что если ненулевых элементов не будет найдено, *i*-ый столбец является нулевым. Меняем столбцы местами таким образом, чтобы все нулевые столбцы оказались последними (для отслеживания количества нулевых столбцов используется специальная переменная). Поскольку столбцам матрицы соответствуют переменные СЛАУ, изменение порядка столбцов означает изменение порядка переменных.

Для фиксации используемого порядка переменных в класс `Slau` следует ввести массив `reoder`, который будет хранить перестановку переменных в результате проведенных преобразований системы. Стандартный порядок следования переменных должен быть инициализирован в конструкторе класса `Slau`.

В результате проведения исключений по всем строкам матрицы коэффициентов в ее левом верхнем углу будет расположена единичная матрица. Порядок единичной матрицы равен рангу матрицы коэффициентов. Строки, расположенные ниже единичной матрицы, являются нулевыми.

После определения ранга матрицы могут возникнуть следующие ситуации:

- если какой-либо нулевой строке полученной матрицы будет соответствовать ненулевой элемент в столбце свободных членов, СЛАУ не имеет решений, т.е. является несовместной;
- $m > n$ и ранг матрицы равен количеству переменных. Тогда получаем n уравнений с n неизвестными, т.е. система будет иметь единственное решение, которое находится в первых n компонентах столбца свободных членов;
- ранг матрицы меньше количества переменных. Тогда СЛАУ имеет множество решений, которые записываются в виде общего решения системы. Общее решение системы отражает линейную зависимость базисных переменных (их количество равно рангу матрицы), от оставшихся свободных переменных, которые могут принимать любые значения. При подстановке конкретных значений свободных переменных в полученные зависимости значения базисных переменных определяются однозначно.

Для определения общего решения рассмотрим СЛАУ после выполненных преобразований:

$$x_i + a'_{ir+1}x_{r+1} + \ldots + a'_{in}x_n = b'_i, \quad i = 1..r$$

где r – ранг матрицы коэффициентов, a'_{ij} – новые коэффициенты при переменных, b'_i – новые значения свободных членов. В таком виде уравнения определяют зависимость базисных переменных от значений

свободных переменных, т. е. определяют общее решение системы. В программе его можно представить в виде матрицы:

$$\begin{pmatrix} b'_1 & -a'_{1r+1} & -a'_{1r+2} & \dots & -a'_{1n} \\ b'_2 & -a'_{2r+1} & -a'_{2r+2} & \dots & -a'_{2n} \\ b'_3 & -a'_{3r+1} & -a'_{3r+2} & \dots & -a'_{3n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ b'_r & -a'_{rr+1} & -a'_{rr+2} & \dots & -a'_{rn} \end{pmatrix}$$

Порядок следования базисных и свободных переменных будет сохранен в специальном массиве `reoder`.

```
// метод Жордана-Гаусса решения СЛАУ
public void JordanGauss ()
{
   // создание копий матрицы коэффициентов и свободных
   // членов для последующих преобразований
   Matrix A = a.Copy();
   Matrix B = b.Copy();
   int count_null_cols = 0;
   // проведение исключений по формулам Жордана-Гаусса
   for(int i = 0; i < m; i++)
   {
      // исключение по i-ой строке
      // проверка возможности исключения
      // по значению ведущего элемента
      if(A[i, i] != 0)
      {
         // исключение во всех строках, кроме ведущей
         for(int k = 0; k < m; k++)
         {
            if(k == i)
               continue;
            double d = A[k, i] / A[i, i];
            for(int j = i; j < n; j++)
               A[k, j] = A[k, j] - d * A[i, j];
            B[0, k] = B[0, k] - d * B[0, i];
         }
         // преобразование ведущей строки
         for(int j = i + 1; j < n; j++)
            A[i, j] /= A[i, i];
         // преобразование i-ого свободного члена
         B[0, i] /= A[i, i];
```

```
   A[i, i] = 1;
}
else
{

   // элемент главной диагонали
   // в i-ой строке равен нулю
   int k;
    // поиск ненулевого элемента ниже
    // в i-ом столбце
    for(k = i + 1; k < m; k++)
       if(A[k, i] != 0)
          break;
    if(k == m)
    {
        // все элементы столбца нулевые
        if(i == n - 1 - count_null_cols)
        {
           // элементов, которые могут быть
           // ведущими, больше нет
           count_null_cols++;
           break;
        }
        // меняем местами столбцы - текущий и
        // последний из непросмотренных
        for(int j = 0; j < m; j++)
        {
           double t = A[j, i];
           A[j, i] = A[j, n-count_null_cols-1];
           A[j, n-count_null_cols-1] = t;
        }
        // отражаем смену столбцов в перестановке
        int te = reoder[i];
        reoder[i] = reoder[n-count_null_cols-1];
        reoder[n-count_null_cols-1] = te;
        count_null_cols++;
        // далее пытаемся провести исключения
        // с той же строкой
        i--;
    }
    else
    {
       // нашли в столбце элемент, который может
       // быть ведущим - меняем местами строки
       for(int l=0; l<n; l++)
       {
          double t = A[i, l];
          A[i, l] = A[k, l];
          A[k, l] = t;
       }
       double p = B[0, i];
```

```
                B[0, i] = B[0, k];
                B[0, k] = p;
                // далее пытаемся провести исключения
                // с той же строкой
                i--;
            }
        }
    }

    // вычисление ранга матрицы после проведения исключения
    rang = m < n-count_null_cols ? m : n-count_null_cols;
    // подсчет количества нулевых строк матрицы
    int null_rows = m - rang;
    // проверка на несовместность системы –
    // если в нулевой строке
    // свободный член не равен нулю
    for(int i = rang; i < m; i++)
        if(B[0, i] != 0)
        {
            isSolved = false;
            return;
        }
    // формирование общего решения для совместной СЛАУ
    // путем переноса свободных переменных в правую часть
    Matrix res = new Matrix(rang, 1 + n - rang);
    for(int i = 0; i < rang; i++)
    {
        res[i,0] = B[0, i];
        for(int j = rang; j < n; j++)
            res[i, j - rang + 1] = -A[i, j];
    }
    x = res;
    isSolved = true;
}
```

Выбор метода для решения СЛАУ осуществляется в функции `Solve()`. Если матрица коэффициентов системы является квадратной, то пользователю предлагается задать метод решения системы: метод Крамера или с помощью обратной матрицы. Если определитель квадратной матрицы окажется нулевым, СЛАУ будет решаться методом исключений Жордана-Гаусса. Для решения СЛАУ с прямоугольной матрицей коэффициентов применим только метод Жордана-Гаусса.

```
// функция выбора метода решения системы линейных уравнений
public void Solve()
{
   if(m == n)
      try
      {
         // матрица коэффициентов квадратная –
         // предоставляется выбор метода решения
         // пользователю
         Console.WriteLine("Метод Крамера - 1,
                            С помощью обратной матрицы - 2");
         int i = int.Parse(Console.ReadLine());
         if(i == 1)
            Kramer();
         else
            InverseMatrix();
      }
      catch(Exception e)
      {
         // генерация исключения происходит при
         // равенстве нулю определителя матрицы
         // коэффициентов. Поэтому осуществляется получение
         // общего решения системы
         JordanGauss();
      }
   else
      // СЛАУ с прямоугольной матрицей коэффициентов
      // решается методом Жордана-Гаусса для получения
      // общего решения
      JordanGauss();
}
```

В конструкторе класса `Slau` необходимо выделить память для хранения матрицы коэффициентов, вектора свободных членов и вектора решения. Это осуществляется путем вызова конструкторов внедренных объектов класса `Matrix`. Также здесь осуществляется первоначальное запоминание порядка следования переменных в массиве `reorder`.

```
// конструктор класса Slau
public Slau(int m1, int n1)
{
   m = m1;      // инициализация количества уравнений
   n = n1;      // инициализация количества переменных
   // выделение памяти под матрицу коэффициентов
   a = new Matrix (m1, n1);
   // выделение памяти под вектор свободных членов
   b = new Matrix (1, m1);
   // выделение памяти под вектор-решение
```

```
        x = new Matrix (1, n1);
        // выделение памяти и заполнение массива
        // для хранения перестановки переменных
        reoder = new int [n];
        for(int i = 0; i < n; i++)
            reoder[i] = i;
    }
```

Отдельно указываются методы ввода и вывода СЛАУ. Вывод полученного решения СЛАУ удобно оформить отдельным методом.

```
    // метод ввода СЛАУ
    public void Input()
    {
        Console.WriteLine("Матрица коэффициентов: ");
        a.Input();
        Console.WriteLine("Вектор свободных членов: ");
        b.Input();
    }

    // метод вывода СЛАУ
    public void Print()
    {
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
                Console.Write("" + a[i, j] + "\t");
            Console.WriteLine("\t" + b[0, i]);
        }
        try
        {
            Console.WriteLine("Решение СЛАУ: ");
            PrintSolution();
        }
        catch (Exception e)
        {
            // печать возможной ошибки
            Console.WriteLine(e.Message);
        }
    }

    // метод вывода полученного решения СЛАУ
    public void PrintSolution()
    {
        if(!isSolved)
        {
            Console.WriteLine("Система несовместна");
            return;
        }
        if(rang < n)
```

```
    {
        // получено общее решение системы
        for(int i = 0; i < rang; i++)
        {
            Console.Write("x" + (reoder[i] + 1) + " = " + x[i, 0]);
            for(int j = 1; j <= n - rang; j++)
            {
                if(x[i, j] == 0)
                    continue;
                if(x[i, j] > 0)
                    Console.Write("+" + x[i, j] + "*x" +
                                        (reoder[rang + j - 1] + 1));
                else
                    Console.Write(""+ x[i, j] + "*x" +
                                        (reoder[rang + j - 1] + 1));

            }
            Console.WriteLine();
        }
    }
    else
    {
        // получен единственный вектор решений
        Console.Write("(");
        for(int i = 0; i < n - 1; i++)
            Console.Write(""+ x[0, i] + ", ");
        Console.WriteLine(""+x[0, n - 1] + ")");
    }
}
```

Использование класса `Slau` может быть таким:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int m, n;
            Console.WriteLine("Введите количество
                                        уравнений системы:");
            m = int.Parse(Console.ReadLine());
            Console.WriteLine("Введите количество
                                        переменных системы:");
            n = int.Parse(Console.ReadLine());
            // создание объекта системы линейных уравнений
            Slau s = new Slau(m, n);
            s.Input();
            // решение системы линейных уравнений
            s.Solve();
            s.Print();
```

```
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Задания для самостоятельной работы

1. Разработать класс «Множество». Определить конструкторы и деструктор. Переопределить операции объединения, пересечения и разности двух множеств, методы для организации ввода-вывода. Написать методы проверки включения одного множества в другое, проверки равенства двух множеств, проверки пустоты множества.
2. Разработать класс «Многоугольник», который хранится в виде массива его вершин. Определить конструктор, методы для организации ввода-вывода и переопределить операции сравнения многоугольников по площади. Написать методы вычисления площади многоугольника, определения, принадлежит ли точка многоугольнику, определения, является ли многоугольник выпуклым.
3. Разработать класс «Целое число в заданной системе счисления». Число должно храниться в виде массива целых чисел (разрядов числа). Определить конструктор, методы для организации ввода-вывода, операции сложения, вычитания, умножения, деления и взятия остатка от деления двух чисел и операции сравнения. Написать методы перевода числа из одной системы счисления в другую.
4. Разработать класс «Бинарное дерево сортировки». Написать конструкторы и деструктор, методы добавления нового узла, удаления узла по ключевому значению, вычисления глубины дерева, объединения двух деревьев, вычисления количества узлов на заданном уровне, определения подобия двух деревьев.

4. Интерфейс для работы с математическими объектами

4.1. Разработка интерфейса

Как известно, с большинством математических объектов (дробями, комплексными числами, векторами, матрицами, полиномами и т.д.) можно выполнять одни и те же операции – сложение, вычитание, умножение и т.д. Для того, чтобы единообразно обрабатывать операции с объектами разных типов, удобно объединить в единый интерфейс все общие операции работы с объектами, а затем последовательно разрабатывать эти классы, раскрывая созданный интерфейс. Наличие общего родительского интерфейса позволит, в частности, оформлять общие по обработке алгоритмы в виде обобщенных функций, что удобно с точки зрения принципа повторного использования программного кода.

Интерфейс работы с математическими объектами, в качестве которых могут выступать матрицы, полиномы, векторы, дроби, комплексные числа должен определять общие операции: сложение, вычитание, умножение на объект и умножение на число. Также требуется включить в интерфейс метод получения строкового представления объекта, чтобы использовать полученную строку для вывода объекта.

```
interface IMathObject
{
   // метод получения суммы объектов
   IMathObject Summa (IMathObject ob);
   // метод получения разности объектов
   IMathObject Substract (IMathObject ob);
   // метод умножения объектов
   IMathObject Multiply (IMathObject ob);
   // метод умножения объекта на число
   IMathObject Multiply(double chislo);
   // метод получения строкового представления объекта
   string ToString();
}
```

4.2. Раскрытие интерфейса для класса «Матрица»

Для раскрытия интерфейса требуется определить в классе `Matrix` все методы, которые указаны в интерфейсе. Приведем код класса `Matrix` с указанием переопределенных операций.

```
class Matrix : IMathObject
{
   // количество строк и столбцов матрицы
   protected int m, n;
    // массив элементов матрицы
    protected double[,] a;

   // конструктор - осуществляет выделение
    // памяти под хранение матрицы
   public Matrix(int m1, int n1)
   {
      n = n1;
      m = m1;
      a = new double[m, n];
   }

   // определение операции сложения двух матриц,
    // раскрывающей метод интерфейса IMathObject
   public IMathObject Summa(IMathObject ob)
   {
      // приведение типа аргумента к классу Matrix
      Matrix ob1 = ob as Matrix;
      if (m != ob1.m || n != ob1.n)
         throw new Exception("Сложение таких матриц невозможно");
      Matrix res = new Matrix(m, n);
      for (int i = 0; i < m; i++)
         for (int j = 0; j < n; j++)
            res[i, j] = a[i, j] + ob1[i, j];
      return res;
   }

   // определение операции вычитания двух матриц,
    // раскрывающей метод интерфейса IMathObject
   public IMathObject Substract(IMathObject ob)
   {
      Matrix ob1 = ob as Matrix;
      if (m != ob1.m || n != ob1.n)
         throw new Exception("Вычитание таких
                                          матриц невозможно");
      Matrix res = new Matrix(m, n);
```

```
   for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
         res[i, j] = a[i, j] - ob1[i, j];
   return res;
}

// определение операции умножения двух матриц,
// раскрывающей метод интерфейса IMathObject
public IMathObject Multiply(IMathObject ob)
{
   Matrix ob1 = ob as Matrix;
   if (n != ob1.m)
      throw new Exception("Такие матрицы перемножить нельзя");
   Matrix res = new Matrix(m, ob1.n);
   for (int i = 0; i < m; i++)
      for (int j = 0; j < ob1.n; j++)
      {
         res[i, j] = 0;
         for (int k = 0; k < n; k++)
            res[i, j] = res[i, j] + a[i, k] * ob1[k, j];
      }
   return res;
}

// определение операции умножения матрицы на число,
// раскрывающей метод интерфейса IMathObject
public IMathObject Multiply(double chislo)
{
   Matrix res = new Matrix(m, n);
   for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
         res[i, j] = a[i, j] * chislo;
   return res;
}

// индексатор для получения элементов матрицы по индексам
double this[int i, int j]
{
   get { return a[i, j]; }
   set { a[i, j] = value; }
}

// метод ввода матрицы
public void Input()
{
   Console.WriteLine("Введите элементы матрицы");
   for (int i = 0; i < m; i++)
   {
      string str = Console.ReadLine();
      string [] s = str.Split(' ');
      for (int j = 0; j < n; j++)
```

```
                a[i, j] = double.Parse(s[j]);
            }
        }

        // метод получения строкового представления матрицы.
        // Элементы в матрице располагаются через символ
        // табуляции, при переходе на новую строку матрицы
        // добавляется символ ‘\n’
        public override string ToString()
        {
            string str = "";
            for (int i = 0; i < m; i++)
            {
                for (int j = 0; j < n - 1; j++)
                    str = str + a[i, j] + "\t";
                str = str + a[i, n - 1] + "\n";
            }
            return str;
        }
    }
```

4.3. Раскрытие интерфейса для класса «Полином»

Реализуем в классе «Полином» методы интерфейса `IMathObject`. Как известно, полином n-ой степени – это функция одной переменной следующего вида:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0 .$$

Как видно из этой формулы, полином задается с помощью максимальной степени аргумента и массива коэффициентов, размер которого на единицу больше степени полинома.

Выполнение операций над полиномами может вызвать обнуление коэффициентов при старших степенях. Поэтому создадим закрытый метод класса, который будет осуществлять сокращение степени полинома и уменьшение размера массива коэффициентов.

```
class Polynom : IMathObject
{
    int n;      // степень полинома
    double[] a; // коэффициенты полинома
```

```
    // конструктор полинома с указанием степени полинома
    public Polynom(int n1)
    {
       n = n1;
       a = new double[n+1];
    }

    // конструктор полинома с указанием степени полинома
    // и массива коэффициентов
    // вызывает первую версию конструктора для выделения
    // памяти, копирует массив коэффициентов и осуществляет
    // преобразование степени, если коэффициенты при
    // старших степенях равны нулю
    public Polynom(int n1, double[] coef): this(n1)
    {
       coef.CopyTo(a,0);
       if(a[n] == 0)
          PreobrPolynom();
    }

    // функция устранения нулевых коэффициентов
    // при больших степенях
    void PreobrPolynom()
    {
       // поиск максимальной степени
       // с ненулевым коэффициентом
       for (int i = n-1; i > 0; i++)
          if (a[i] != 0)
          {
             // полином должен иметь i-ую степень,
             // создаем новый массив и меняем
             // значения полей текущего объекта
             double[] b = new double[i + 1];
             a.CopyTo(b, i);
             a = b;
             n = i;
             return;
          }
       // все коэффициенты нулевые – в любом случае получаем
       // полином нулевой степени
       n = 0;
       double c = a[0];
       a = new double[1]; a[0] = c;
    }
   . . .
}
```

Ввод и вывод полинома определим через символьные строки. Для этого в класс `Polynom` добавим операцию преобразования полинома в тип `string` и статическую функцию `Parse()`, которая создает

объект полинома, получая данные из символьной строки, содержащей его коэффициенты. Для удобства обращения к коэффициентам полинома создадим индексатор.

```
// индексатор для получения коэффициента полинома по степени
double this[int i]
{
   get {  return a[i]; }
   set { a[i] = value; }
}

// операция получения строкового представления полинома
public static implicit operator string(Polynom ob)
{
   string str = "";
   for (int i = ob.n; i >0; i--)
      str = str + string.Format("{0}*x^{1}+", ob[i], i);
   str = str + string.Format("{0}\n", ob[0]);
   return str;
}

// функция формирования полинома из
// строки с коэффициентами полинома
public static Polynom Parse(string str)
{
   // разделение коэффициентов полинома в строке
    string[] s = str.Split(' ');
    // формирование полинома по количеству элементов в
    // массиве строк
   Polynom res = new Polynom(s.Length - 1);
   for (int i = 0; i < s.Length; i++)
      res[i] = double.Parse(s[s.Length - 1 - i]);
   return res;
}
```

Наконец, опишем те операции, которые следует определить согласно интерфейсу `IMathObject`.

```
// функция получения строки, представляющей полином
public override string ToString()
{
   // вызов оператора преобразования полинома в строку
   return this;
}
```

```
// функция суммирования двух полиномов
public IMathObject Summa(IMathObject ob)
{
   Polynom ob1 = ob as Polynom;
   if (n > ob1.n)
   {
      // степень первого полинома больше, чем второго
      Polynom res = new Polynom(n);
      for (int i = 0; i < ob1.n + 1; i++)
         res[i] = a[i] + ob1[i];
      for (int i = ob1.n + 1; i < n + 1; i++)
         res[i] = a[i];
      // вызов преобразования полинома-результата, если
      // коэффициенты при старших степенях обнуляются
      if (res[n]==0)
         res.PreobrPolynom();
      return res;
   }
   else
   {
      // степень второго полинома больше, чем у первого
      Polynom res = new Polynom(ob1.n);
      for (int i = 0; i < n + 1; i++)
         res[i] = a[i] + ob1[i];
      for (int i = n + 1; i < ob1.n + 1; i++)
         res[i] = ob1[i];
      // вызов преобразования полинома-результата, если
      // коэффициенты при старших степенях обнуляются
      if (res[n] == 0)
         res.PreobrPolynom();
      return res;
   }
}

// функция вычитания полиномов
public IMathObject Substract(IMathObject ob)
{
   // вычитание осуществляется как сумма полиномов
   // второй операнд при этом должен быть умножен на -1
   Polynom ob1 = ob as Polynom;
   Polynom temp = ob1.Multiply(-1.0) as Polynom;
   Polynom res = Summa(temp) as Polynom;
   return res;
}

// функция умножения двух полиномов
public IMathObject Multiply(IMathObject ob)
{
   Polynom ob1 = ob as Polynom;
   Polynom res = new Polynom(n + ob1.n);
   for (int i = 0; i < n + 1; i++)
```

```
            for (int j = 0; j < ob1.n + 1; j++)
               res[i + j] += a[i] * ob1[j];
         return res;
      }


      // функция умножения полинома на число
      public IMathObject Multiply(double chislo)
      {
         Polynom res = new Polynom(n);
         for (int i = 0; i < n + 1; i++)
            res[i] = chislo * a[i];
         return res;
      }
```

4.4. Использование интерфейса IMathObject

Приведем простой пример параметризованной функции, демонстрирующей использование методов, которые были объявлены в интерфейсе `IMathObject`.

```
static void Demo<T>(T ob1, T ob2) where T : IMathObject
{
    Console.WriteLine(ob1);
   Console.WriteLine(ob2);
   Console.WriteLine("Сумма");
   IMathObject res = ob1.Summa(ob2);
   Console.WriteLine(res);
   Console.WriteLine("Вычитание");
   res = ob1.Substract(ob2);
   Console.WriteLine(res);
   Console.WriteLine("Умножение");
   res = ob1.Multiply(ob2);
   Console.WriteLine(res);
   Console.WriteLine("Умножение на число");
   res = ob1.Multiply(4);
   Console.WriteLine(res);
}
```

Данный метод получает в качестве параметров два объекта одинакового типа, который раскрывает интерфейс `IMathObject`. Далее с этими объектами производятся те операции, которые определены в интерфейсе – суммирование этих объектов, получение

их разности, умножение объектов и умножение первого из них на число.

Продемонстрируем вызов этого метода:

```
static void Main(string[] args)
{
   Console.WriteLine("Выберите режим работы с объектами:
                                    1 - Матрица, 2 - Полином");
   string str = Console.ReadLine();
   switch (str)
   {
      case "1":
          {
              // выбран режим работы с матрицами
             Console.WriteLine("Введите две матрицы:");
             Matrix a = new Matrix(3, 3);
             a.Input();
             Matrix b = new Matrix(3, 3);
             b.Input();
             Console.WriteLine("Демонстрация операций с
                                                  матрицами:");
             // вызов демо-функции
             // для параметров-матриц
             Demo(a, b);
             break;
          }
      case "2":
          {
              // выбран режим работы с полиномами
              Console.WriteLine("Введите два полинома:");
              Polynom a = Polynom.Parse(Console.ReadLine());
              Polynom b = Polynom.Parse(Console.ReadLine());
              Console.WriteLine("Демонстрация операций с
                                                  полиномами:");
              // вызов демо-функции
              // для параметров-полиномов
              Demo(a, b);
              break;
          }
      }
 }
```

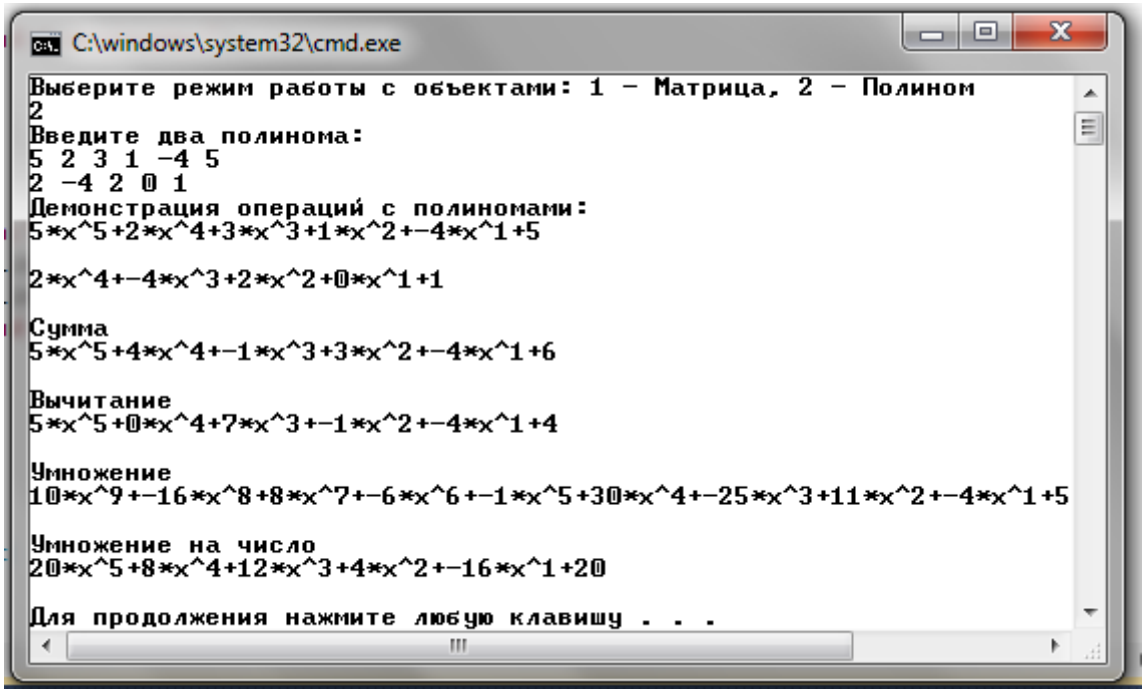


Рис. 4.1. Демонстрация режима работы с объектами-полиномами

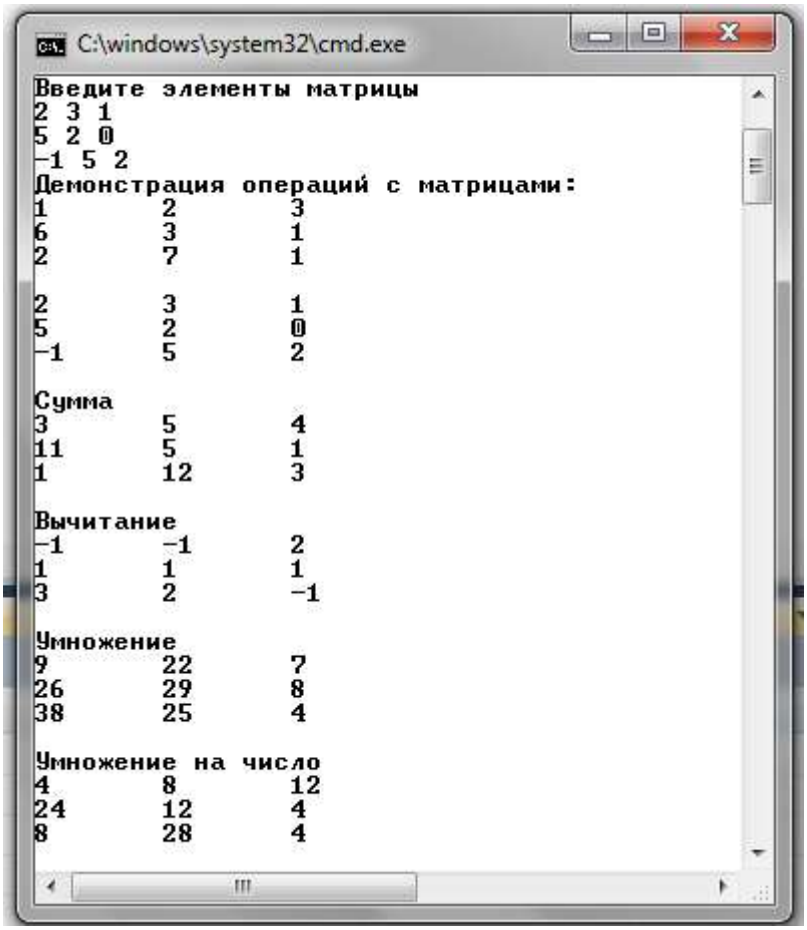


Рис. 4.2. Демонстрация режима работы с объектами-матрицами

Задания для самостоятельной работы

1. Раскрыть интерфейс `IMathObject` на примере класса «Рациональное число». Протестировать обращение к наследуемым методам без явного указания на тип объектов.

2. Раскрыть интерфейс `IMathObject` на примере класса «Комплексное число». Протестировать обращение к наследуемым методам без явного указания на тип объектов.

3. Для массива объектов, которые раскрывают интерфейс `IMathObject`, создать метод-обобщение, который находит сумму объектов массива. Протестировать метод для массива матриц, массива комплексных чисел и массива полиномов.

4. Разработать интерфейс «Фигура на плоскости». Определить для него операции перемещения, поворота, определения площади, получения местоположения и пр. Раскрыть интерфейс в классах «Треугольник», «Прямоугольник», «Многоугольник».

5. Разработать класс «Линейная функция в *n*-мерном пространстве» ($f(x) = \langle b, x \rangle + c$). Определить конструктор, переопределить операции сложения и вычитания функций, умножения функции на число. Для организации ввода-вывода переопределить операцию преобразования в строку и статический метод `Parse()`. Написать методы вычисления значения функции в точке, получения градиента функции. Наследовать от этого класса класс «Квадратичная функция в *n*-мерном пространстве» ($f(x) = \langle Ax, x \rangle + \langle b, x \rangle + c$). Переопределить все указанные операции и методы для класса-наследника.

6. Разработать класс «Граф» в виде списка смежности. Определить конструкторы и деструктор. Переопределить операции ввода-вывода. Написать методы проверки связности графа, проверки полноты графа, проверки двудольности графа, получения дополнения графа, нахождения источника графа, нахождения стока графа. Наследовать от этого класса класс «Взвешенный граф». Написать методы получения кратчайшего пути между двумя вершинами алгоритмом Дейкстры, получения каркаса минимального веса алгоритмами Прима и Краскала.

5. Множество точек на плоскости

В задачах поиска экстремумом функции на некотором множестве точек множество зачастую задается в виде системы ограничений. Каждое ограничение представляет собой уравнение или неравенство, в котором левая часть записывается в виде некоторой функции, определенной в пространстве R_n ($n = 1,2,\ldots$), а правая часть – число.

Например, в пространстве R_2 множество точек, заданное системой ограничений

$$\begin{cases} x^2 + y^2 \le 4; \\ y \ge x; \\ -2x \le y; \\ y \ge 0 \end{cases}$$

выглядит так:

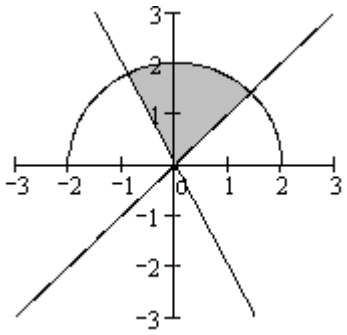


Рис. 5.1. Множество допустимых точек задачи.

Создадим систему классов для описания какого-либо множества в пространстве R_2.

5.1. Структура хранения системы ограничений

Согласно определению, множество допустимых точек задается системой ограничений. Поэтому для задания множества необходимо определить количество ограничений в системе и их набор.

Ограничение может быть представлено следующим образом:

функция в левой части	тип ограничения	правая часть
$f(x,y)$	$(=, <, >, \geq, \leq, \neq)$	C (const)

Для задания ограничения требуется знать функцию его левой части, константу, стоящую в правой части и тип неравенства/равенства.

Определим функции 1-ого и 2-ого порядков, которые будут использоваться в ограничениях:

- линейная – $f(x,y) = ax + by$;
- эллиптическая – $f(x,y) = \frac{(x-x_0)^2}{a^2} + \frac{(y-y_0)^2}{b^2}$;
- гиперболическая – $f(x,y) = \frac{(x-x_0)^2}{a^2} - \frac{(y-y_0)^2}{b^2}$;
- параболическая – $f(x,y) = (y-y_0)^2 - 2px$.

Линейная функция задается с помощью коэффициентов a и b. Для определения эллиптической и гиперболической функций требуется задать коэффициенты a и b, а также координаты точки ($x0$, $y0$), задающей смещение графика функции относительно начала координат. Параболическая функция задается параметром p и смещением графика по оси OY на величину $y0$.

Для определения функции в левой части ограничения можно объявить следующий класс `Function`:

```
// класс, задающий функцию в левой части ограничения
class Function
{
   int typeFunction;  // тип кривой: 1 – линейная,
                      //             2 – эллиптическая,
                      //             3 – гиперболическая,
                      //             4 – параболическая
    // параметры, задающие функции разных типов
   double a, b, p, x0, y0;
   . . .
}

// перечисление для определения типа ограничения
```

```
// le – <=, ge – >=, e – =, l – <, g – >, n – <>
enum TypeInequation { le, ge, e, l, g, n };


// класс, определяющий ограничение
class Constraint
{
   Function  function;        // объект, описывающий функцию
                              // в левой части ограничения
    double b;                 // правая часть
    TypeInequation type;      // тип ограничения
    . . .
}

// класс, определяющий множество
class Set
{
   Constraint [] constraints;// массив ограничений
    int n;                   // количество ограничений в системе
    . . .
}
```

При определении, удовлетворяет ли точка ограничению, требуется вычислять в этой точке значение функции, заданной с помощью объекта класса `Function`. Для этого в классе должен быть определен соответствующий метод:

```
// метод вычисления значения функции в заданной точке
public double Calculate(double x, double y)
{
   double value = 0.0;
    switch (typeFunction)
   {
       case 1:
           value = a * x + b * y; break;
       case 2:
           value = (x - x0) * (x - x0) / (a * a) +
                           (y - y0) * (y - y0) / (b * b); break;
       case 3:
           value = (x - x0) * (x - x0) / (a * a) –
                           (y - y0) * (y - y0) / (b * b); break;
       case 4:
            value = (y - y0) * (y - y0) – 2*p*x; break;
       default:
           // неизвестен тип функции,
           // поэтому генерируется исключение
            throw new Exception("Неизвестен тип функции");
   }
   return value;
```

```
}
```

Аналогичная структура программного кода (использование оператора `switch`) будет использоваться во всех методах класса `Function`. Недостатком этой структуры является необходимость внесения изменений во все методы класса, если добавляется новый тип функции или удаляется имеющийся. Если типов функций будет много, то методы становятся объемными и трудно читаемыми.

Еще одним недостатком является наличие неиспользуемых переменных класса `Function`. Например, для задания линейной функции достаточно использовать переменные `typeFunction`, `a` и `b`, а переменные `p`, `x0` и `y0` будут определены, но их значения игнорируются. Подобного неэффективного использования памяти следует избегать.

5.2. Иерархия классов кривых 1-ого и 2-ого порядков

Определить функции можно, используя другую структуру классов, которая не приводит к изменению уже написанного кода при добавлении нового типа функции и хранит для каждого типа функции столько параметров, сколько необходимо для ее задания. В этом случае используются принципы наследования и полиморфизма.

Для каждого типа функции задается собственный класс, например, `Line`, `Ellipse`, `Hyperbola`, `Parabola`. Все эти классы обладают одинаковым поведением – должно вычисляться значение функции, нужно вводить параметры функции и выводить представление функции на экран. Поэтому можно эти методы определить в отдельном классе `Function`, родительском для классов различных типов функций. Поскольку родительский класс «не знает», какая функция вычисляется, как ее распечатать, какие параметры ее определяют, все эти методы должны быть абстрактными или не иметь какой-либо содержательной обработки. Соответственно, класс `Function` должен быть абстрактным:

```
// класс, задающий функцию в левой части ограничения
abstract class Function
{
   // абстрактный метод вычисления функции
   public abstract double Calculate(double x, double y);
   // виртуальный метод ввода параметров функции
   public virtual void Input()
   {
      Console.WriteLine("Введите данные, определяющие функцию");
   }
   // абстрактный метод вывода функции на печать
   public abstract string Output();
}
```

Классы, задающие кривые различных типов, имеют одинаковую структуру. Для примера приведем определение методов класса `Ellipse`:

```
// класс, задающий эллиптическую функцию
class Ellipse : Function
{
   // параметры, задающие эллиптическую функцию
   double a, b;
   double x0, y0;

   // конструктор эллиптической функции
   public Ellipse(double a1= 1, double b1= 1,
                                  double x= 0, double y = 0)
   {
      // если параметр a или b равен нулю, эллиптическую
      // функцию определить невозможно.
      // Поэтому генерируется исключение
      if (a1 == 0 || b1 == 0)
         throw new Exception("Функция не может быть
                                  определена такими параметрами");
      a = a1;
      b = b1;
      x0 = x;
      y0 = y;
   }


   // переопределение виртуального метода
   // вычисления значения функции
   public override double Calculate(double x, double y)
   {
      return (x - x0) * (x - x0) / (a * a) +
```

```
                                (y - y0) * (y - y0) / (b * b);
    }

    // переопределение виртуального метода ввода
    // параметров функции
    public override void Input()
    {
        // вызов базовой версии метода
        base.Input();
        // ввод параметром эллиптической функции
        a = double.Parse(Console.ReadLine());
        b = double.Parse(Console.ReadLine());
        x0 = double.Parse(Console.ReadLine());
        y0 = double.Parse(Console.ReadLine());
    }

    // переопределение виртуальной функции вывода на печать
    public override string Output()
    {
        return "(x-" + x0 + ")^2/" + a * a +
                                "+ (y - " + y0 + ")^2/" + b * b;
    }
}
```

При такой структуре хранения функции изменяется суть поля `function` класса `Constraint`. Внедрение объекта абстрактного типа, определяющего функцию левой части, осуществить нельзя. Поэтому в класс ограничения включается ссылка на базовый класс, которая может хранить адрес объекта любого дочернего класса. С помощью такой ссылки будут вызваны виртуальные методы вычисления, ввода и распечатки функций дочерних классов. Таким образом, существенно изменятся методы класса `Constraint`.

Теперь приведем объявление классов `Constraint` и `Set` с внесенными изменениями:

```
// класс, определяющий ограничение
class Constraint
{
    Function function;              // ссылка на объект функции
                                    // в левой части ограничения
    double b;                       // правая часть
    TypeInequation type;            // тип ограничения

    // конструктор ограничения – параметры ограничения
    // инициализируются путем ввода с клавиатуры
    public Constraint()
    {
        Input();
```

```
}


// метод проверки выполнения ограничения
public bool IsExecute(double x, double y)
{
   // вычисление функции левой части ограничения
   double val = function.Calculate(x, y);
   // сравнение с правой частью согласно виду ограничения
    switch(type)
    {
       case TypeInequation.le:
                if (val <= b)
                   return true;
                break;
       case TypeInequation.ge:
                if (val >= b)
                   return true;
                break;
       case TypeInequation.e:
                if (val == b)
                   return true;
                break;
       case TypeInequation.l:
                if (val < b)
                   return true;
                break;
       case TypeInequation.g:
                if (val > b)
                   return true;
                break;
       case TypeInequation.n:
                if (val != b)
                   return true;
                break;
    }
    return false;
}

// метод проверки выполнения равенства f(x,y) = b
// для ограничений типа "<=", ">=", "="
public bool IsOnBound(double x, double y)
{
   // для ограничений видов "<",">", "<>"
   // равенство не должно выполняться
   if (type == TypeInequation.l ||
           type == TypeInequation.g || type == TypeInequation.n)
      return false;
   // вычисление значения функции в точке
   double val = function.Calculate(x,y);
   // сравнение с правой частью на выполнение равенства
```

```
        if (val == b)
            return true;
        return false;
    }

    // метод ввода ограничения
    public void Input()
    {
        int choice;
        // ввод типа ограничения
        while(true)
        {
            Console.WriteLine("Линейная - 1, Эллиптическая - 2,
                              Гиперболическая - 3, Параболическая -
                        4");
            choice = int.Parse(Console.ReadLine());
            if(choice >= 1 && choice <= 4)
                break;
        }
        // создание объекта функции левой части ограничения
        // в зависимости от введенного типа
        switch(choice)
        {
            case 1:
                function = new Line();
                break;
            case 2:
                function = new Ellipse();
                break;
            case 3:
                function = new Hyperbola();
                break;
            case 4:
                function = new Parabola();
                break;
        }
        // ввод параметров создаваемой функции
        function.Input();
        // ввод вида ограничения
        while(true)
        {
            Console.WriteLine("<= - 0, >= - 1,  = - 2,
                                            < - 3, > - 4, <> - 5");
            choice = int.Parse(Console.ReadLine());
            if(choice >= 0 && choice <= 5)
                break;
        }
        type = (TypeInequation) choice;
        // ввод правой части
        Console.WriteLine("Правая часть");
        b = double.Parse(Console.ReadLine());
```

```
        }


        // операция получения строкового представления ограничения
        static public implicit operator string(Constraint ob)
        {
            // получения строкового представления функции
             // из левой части ограничения
            string res = ob.function.Output();
            // вывод в строку знака вида ограничения
             switch(ob.type)
             {
                case TypeInequation.le:
                        res = res + "<=";
                        break;
                case TypeInequation.ge:
                        res = res +  ">=";
                        break;
                case TypeInequation.e:
                        res = res + "=";
                        break;
                case TypeInequation.l:
                        res = res +"<";
                        break;
                case TypeInequation.g:
                        res = res + ">";
                        break;
                case TypeInequation.n:
                        res = res + "<>";
                        break;
                default:
                       throw new Exception
                              ("Не существует такого вида ограничения");
             }
             // вывод правой части ограничения
             res = res + ob.b;
             return res;
        }
    }
```

В классе `Constraint` используется принцип полиморфизма при работе с объектами функций левой части ограничения. Класс содержит ссылку на абстрактный класс `Function`, которая может хранить адрес объекта класса `Line`, `Ellipse`, `Hyperbola` или `Parabola`, задающего конкретную функцию. При вводе ограничения у пользователя запрашивается вид нужной функции и создается объект соответствующего класса, адрес которого сохраняется в переменной-ссылке `function`. При вызове методов `Input()`, `Output()` и `Calculate()` через ссылку

function будут вызываться виртуальные функции того класса, адрес которого хранится в function. Такие вызовы выполняются как в функциях ввода/вывода ограничения, так и в методах проверки выполнения некоторых условий для точки. Таким образом, анализировать в этих методах тип функции левой части ограничения уже не потребуется.

Далее определим методы класса Set. Отметим, что в этом классе отсутствует метод ввода информации о системе ограничений. Ввод выполняется в конструкторе класса Set при создании массива ограничений (в конструкторе каждого ограничения).

```
// класс, задающий множество как систему ограничений
class Set
{
   Constraint[] constraints;   // массив ограничений
    int n;                     // количество ограничений
   // конструктор, задающий количество ограничений
    public Set(int n1)
   {
      n = n1;
       constraints = new Constraint [n];
      for (int i = 0; i < n; i++)
          // при создании ограничения будут
         // запрошены его параметры для ввода
         constraints[i] = new Constraint();
   }
   // метод проверки, принадлежит ли точка множеству
    public bool Belongs(double x, double y)
   {
      // точка не принадлежит множеству, если не выполняется
      // хотя бы одно из ограничений, определяющих множество
      for(int i = 0; i < n; i++)
          if (!constraints[i].IsExecute(x,y))
             return false;
       return true;
   }

   // метод проверки, лежит ли точка на границе множества
    public bool IsOnBound(double x, double y)
   {
      // точка лежит на границе, если выполняются все
      // ограничения и хотя бы одно из них – как равенство
      if (Belongs(x,y))
          for(int i = 0; i < n; i++)
             if (constraints[i].IsOnBound(x,y))
                return true;
       return false;
   }
```

```
    // операция получения строкового представления множества
    static public implicit operator string(Set ob)
    {
        string res = "";
        for (int i = 0; i < ob.n; i++)
            res = res + ob.constraints[i] + "\n";
        return res;
    }
}
```

Продемонстрируем работу методов этих классов для множества, которое определено на Рис. 5.1.

```
class Program
{
    static void Main(string[] args)
    {
        Set set=new Set(3);
        double x1 = 0, y1 = 1;
        double x2 = 1, y2 = 1;
        if(set.Belongs(x1,y1))
            Console.WriteLine("Точка (0,1) принадлежит множеству");
        else
            Console.WriteLine("Точка (0,1) не принадлежит
                                                    множеству");
        if(set.IsOnBound(x1,y1))
            Console.WriteLine("Точка (0,1) лежит на границе
                                                    множества");
        else
            Console.WriteLine("Точка (0,1) не лежит на границе
                                                    множества");
        if(set.Belongs(x2,y2))
            Console.WriteLine("Точка (1,1) принадлежит множеству");
        else
            Console.WriteLine("Точка (1,1) не принадлежит
                                                    множеству");
        if(set.IsOnBound(x2,y2))
            Console.WriteLine("Точка (1,1) лежит на границе
                                                    множества");
        else
            Console.WriteLine("Точка (1,1) не лежит на границе
                                                    множества");
    }
}
```

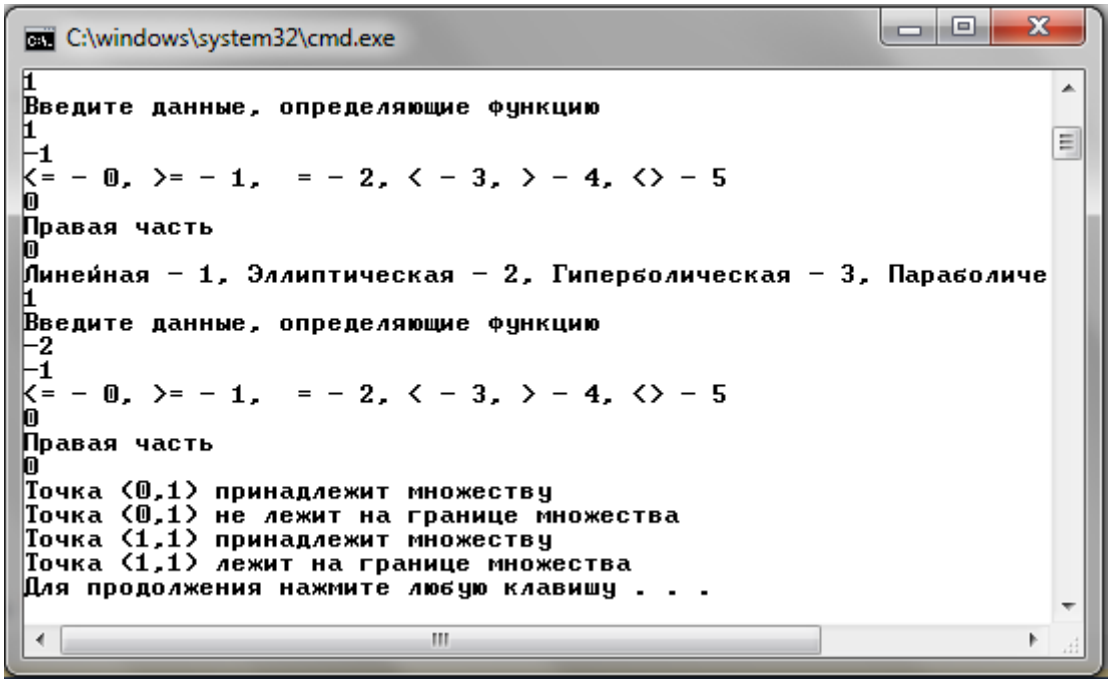


Рис. 5.2. Демонстрация работы программы,
задающей систему ограничений

Задания для самостоятельной работы

1. Определить интерфейс «Фигура на плоскости» и раскрыть его для классов «Треугольник», «Прямоугольник», «Многоугольник», «Круг» и пр. Определить класс «Рисунок» как массив объектов-фигур. Реализовать для рисунка операции перемещения, распечатки информации о рисунке, повороте и пр.

2. Создать иерархию классов «Вагоны пассажирского поезда» с разделением на купейные, плацкартные, СВ. Каждый класс вагона должен содержать информацию о количестве мест разных типов (нижнее, верхнее, нижнее боковое, верхнее боковое), о наличии дополнительных услуг и ценах на них. С помощью виртуальных функций получить полный доход от эксплуатации вагона. Создать класс «Пассажирский поезд», который хранит список вагонов. Подсчитать доход от одного рейса поезда.

3. Создать абстрактный класс «Функция в n-мерном пространстве». Наследовать от него класс «Линейная функция $f(x)=\langle b,x\rangle+c$» и класс «Квадратичная функция $f(x)=\langle Ax,x\rangle+\langle b,x\rangle+c$». Реализовать виртуальные методы вычисления значения функции и ее градиента в точке. Определить класс

«Множество точек в *n*-мерном пространстве», которое определяется как список неравенств вида «$f(x) \le b$», где $f(x)-$ линейные или квадратичные функции. Написать методы, определяющие, принадлежит ли точка множеству и лежит ли точка на границе множества.

4. Создать иерархию классов-многоугольников: «Треугольник», «Четырехугольник», «Пятиугольник», «Шестиугольник». Создать класс «Фигура на плоскости», который задает фигуру как массив объектов-многоугольников. Определить в классе методы перемещения фигуры, определения, принадлежит ли точка фигуре и др.

6. Классы - коллекции

В C# имеются специальные структуры данных, предназначенные для работы с группами объектов. Простейшим типом таких структур является массив. Более сложные структуры предполагают возможность динамического изменения их размера, т.е. при необходимости можно в структуру данных добавить новый элемент, или удалить существующий. При этом сама структура данных изменяется. Такие структуры данных называют динамическими. Они построены таким образом, чтобы операции изменения размеров происходили быстро.

Простейшим примером динамической структуры данных является динамический связный список – линейная структура данных, память под хранение элементов которой выделяется в момент добавления элементов в список. Простота изменения размера списка связана с тем, что вместе с отдельным элементом списка хранится местоположение следующего за ним элемента. Для получения доступа ко всем элементам достаточно знать адрес первого элемента списка (который часто называют заголовком `head`).

Рис.6.1. Исходный список.

Операция добавления нового элемента в список происходит следующим образом:

1) находится тот элемент, который должен предшествовать месту вставки нового (`current`) (Рис.6.2 а);

2) создается новый элемент (`help`) (Рис.6.2 а);

3) указывается, что следующим за `help` будет тот элемент, который следовал за `current` (Рис.6.2 б);

4) указывается, что следующим за `current` будет `help` (Рис.6.2 в).

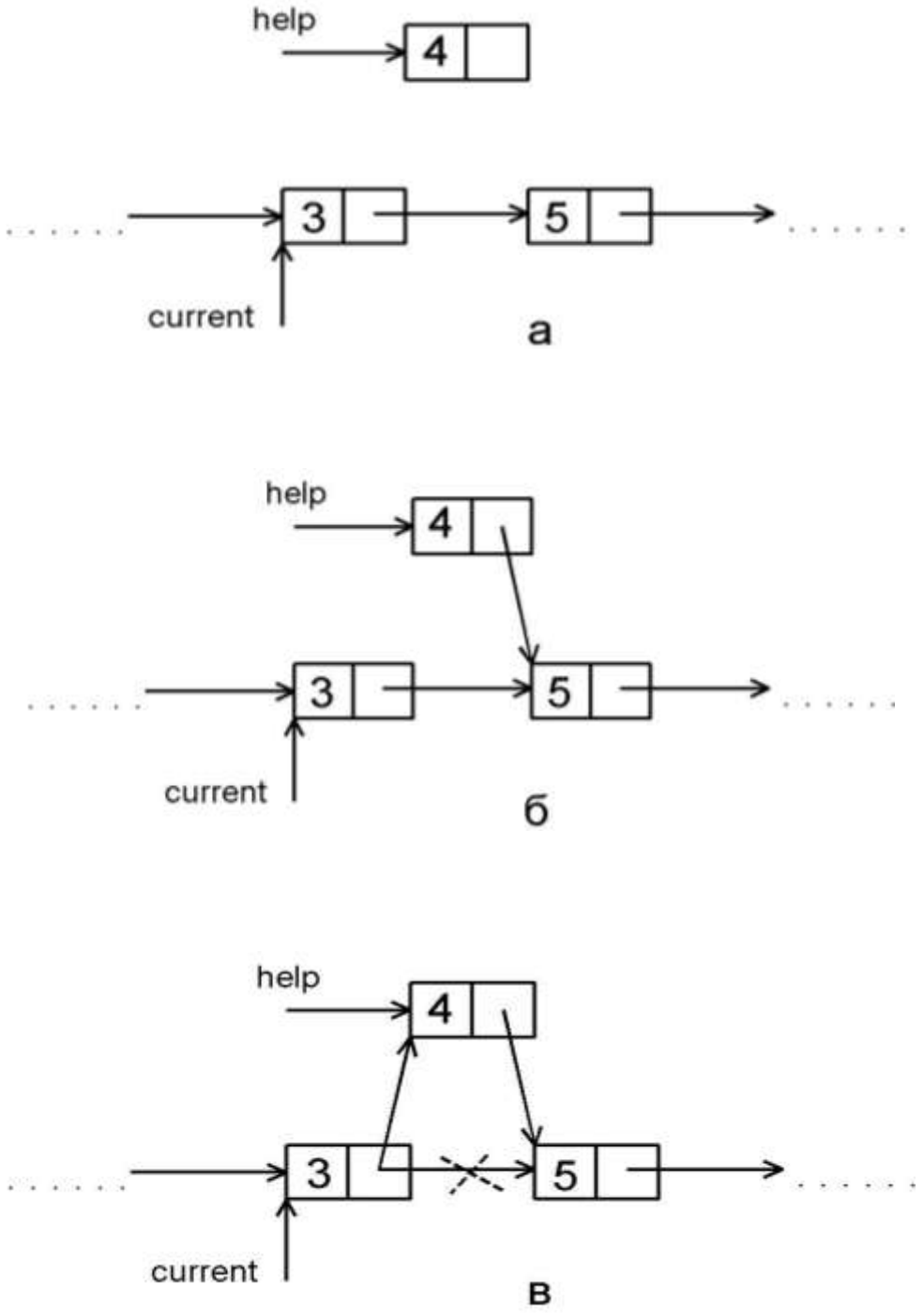


Рис 6.2. Вставка елемента в списка.

Единственная операция в данной последовательности, которая требует времени – операция поиска позиции, предшествующей месту вставки. Для этого вовсе не обязательно просматривать весь список. Для сравнения при изменении размера массива потребовалось бы:

1) выделить новую память, достаточную для его хранения;
2) скопировать все элементы массива с учетом добавления нового

элемента;

3) освободить ранее занимаемую память.

Видно, что операций потребуется выполнить намного больше.

Удаление элемента из списка также не требует больших затрат времени и производится по следующей схеме:

1) находится элемент, который должен предшествовать месту удаления (`current`) и удаляемый элемент (`help`) (Рис. 6.3 а);

2) указывается, что следующим за `current` будет тот элемент, который следовал за `help` (Рис. 6.3 б);

3) удаляется элемент `help` из памяти (Рис. 6.3 в).

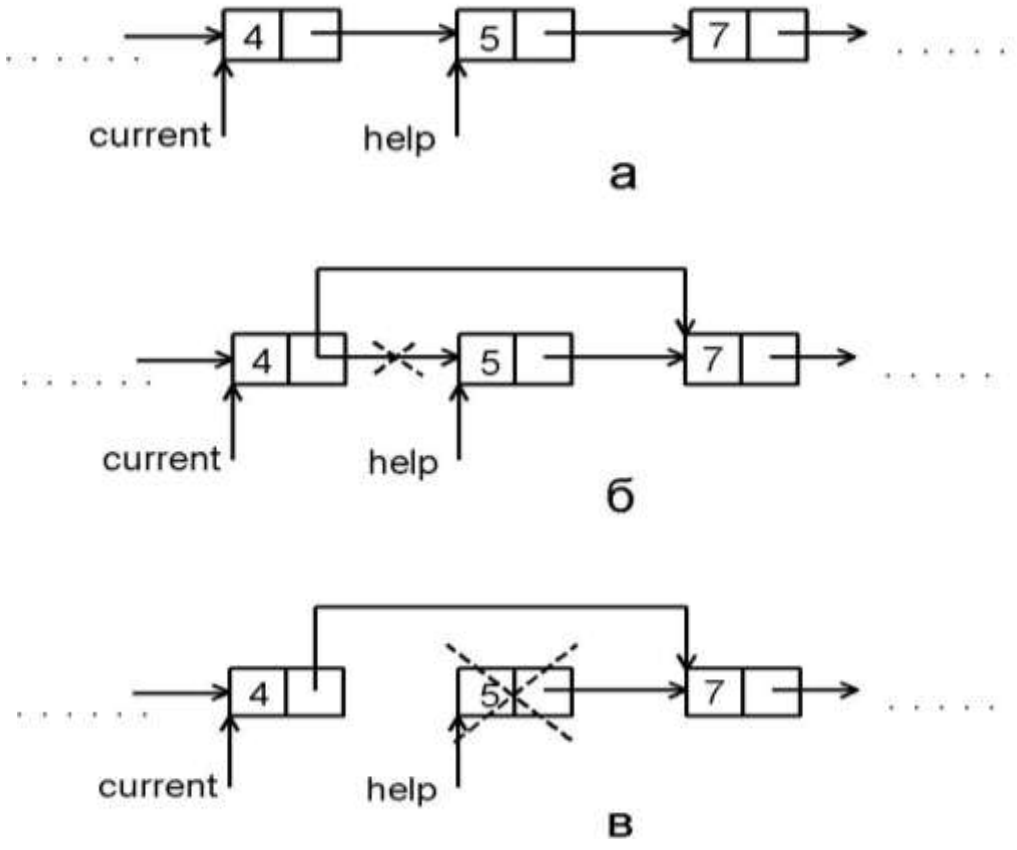


Рис 6.3. Удаление элемента из списка.

Как видим, любая динамическая структура данных требует соблюдения определенных правил вставки и удаления элементов, основанных на способе их хранения. Поскольку часто в задачах используются аналогичные структуры данных, язык программирования C# содержит специальный набор классов, которые реализуют работу с разнообразными типами коллекций.

Существует несколько разных типов коллекций – коллекции, хранящие объекты, наследуемые от класса `object` (т.е. хранящие все, что угодно), и коллекции, построенные по принципу обобщения (когда при создании объекта коллекции следует указать тип его элемента). Классы, реализующие первый тип коллекций, определены в пространстве имен `System.Collection`, а для второго типа – в его подпространстве `System.Collection.Generic`.

Таблица 1. Примеры классов- коллекций общего назначения

Класс	Описание
`Stack`	Стек – частный случай однонаправленного списка, действующий по принципу: последним пришел – первым вышел
`Queue`	Очередь – частный случай однонаправленного списка, действующего по принципу: первым пришел –первым вышел
`ArrayList`	Динамический массив, т.е. массив который при необходимости может увеличивать свой размер
`HashTable`	Хеш-таблица для пар ключ/значение
`SortedList`	Отсортированный список пар ключ/значение

Таблица 2. Примеры классов коллекций -обобщений

Класс	Описание
`Stack<T>`	Стек – частный случай однонаправленного списка, действующий по принципу: последним пришел – первым вышел
`Queue<T>`	Очередь – частный случай однонаправленного списка, действующего по принципу: первым пришел – первым вышел
`List<T>`	Динамический массив, т.е. массив который при необходимости может увеличивать свой размер
`Dictionary <TKey, TValue>`	Хеш-таблица для пар ключ/значение
и т.д.	

Основные методы и способы работы с классами, реализующими одну и ту же структуру данных в варианте коллекции общего

назначения и коллекции-обобщения, одинаковы. Так, классы `ArrayList` и `List<T>` имеют одинаковые методы для добавления элементов в список, удаления заданного элемента из списка, просмотра элементов списка и т.д. Единственным серьезным отличием является только то, что `ArrayList` не следит за типами объектов, которые хранятся в списке. Поэтому преобразование их к требуемому типу данных становится задачей разработчика программы.

Далее разберем принципы работы с указанными структурами данных на нескольких примерах.

6.1. Использование стеков и очередей

Стек – это динамическая линейная структура данных, в которой добавление элементов и их извлечение выполняются с одного конца, называемого вершиной стека (головой - `head`). При выборке элемент исключается из стека. Другие операции со стеком не определены. Говорят, что стек реализует принцип обслуживания LIFO («last in – first out» – «последний пришел – первым вышел»).

Среди коллекций языка C# реализованы классы стека как класса общего назначения, так и стека как класса-обобщения.

Среди методов и свойств класса `Stack` выделяют:

- `Count` – свойство для получения числа элементов в стеке;
- `Push(object)` – метод добавления элемента в стек;
- `Pop()` – метод извлечения элемента из стека;
- `Peek()` – метод получения элемента, который находится в вершине стека, не извлекая его;
- `Contains(object)` – метод проверки наличия в стеке заданного объекта;
- `Clear()` – очистка стека.

Стек часто используется в алгоритмах как вспомогательная структура данных в тех случаях, когда требуется изменить порядок каких-либо элементов на обратный. В качестве примера рассмотрим следующую задачу.

Пусть дано целое положительное число. Требуется перевести его в заданную систему счисления.

Напомним, что при переводе числа в другую систему счисления производится последовательное получение остатков от деления числа на основание системы счисления, которые при записи в обратном порядке образуют требуемое представление числа. Поэтому остатки от деления заносятся в стек, а затем извлекаются из него для формирования строкового представления записи числа. Если основание системы счисления больше 10 (в этом случае для записи числа используются буквенные обозначения), в строку заносится соответствующий символ (10 – ‘A’, 11 – ‘B’ и т.д.).

Решение данной задачи оформлено в виде функции, в которой используется объект класса `Stack` и его методы.

```
// определение функции перевода положительного целого числа
// из десятичной системы счисления в любую заданную
// с основанием от 2 до 16.
// number - положительное целое число
// в десятичной системе счисления
// baseSS – основание системы счисления
// функция возвращает строковое представление записи числа
static string PreobrChislo(int number, int baseSS)
{
   // создается стек для хранения целых чисел
   Stack<int> s = new Stack<int>();
   // вычисленные остатки от деления числа
   // на основание системы счисления помещаются в стек
   while(number != 0)
   {
      s.Push(number % baseSS);
      number = number / baseSS;
   }
   // формирование строкового представления записи числа
   // путем извлечения значений из стека
   string res="";
   try
   {
      while(true)
      {
         int n = s.Pop();
         if(n < 10)
            res = res + n;
         else
            res = res + (char)((int)'A' + n - 10);
      }
   }
```

```
        catch(Exception e)
        {

            // когда стек становится пустым,
            // представление числа сформировано
        }
        return res;
    }
```

Очередь, как и стек, является еще одним частным случаем однонаправленного списка. Добавление элементов в очередь выполняется в один конец («хвост»), а выборка производится с другого конца («головы»); при выборке элемент исключается из очереди. Другие операции с очередью не определены. Очередь реализует принцип обслуживания FIFO («first in – first out», первым пришел – первым вышел).

В C# очередь представляется классом `Queue`, который также как и стек реализуется как коллекция общего назначения и как класс-обобщение.

Среди методов и свойств класса `Queue` выделяют:

- `Count` – свойство получения числа элементов в очереди;
- `Enqueue(object)` – метод добавления элемента в очередь;
- `Dequeue()` – метод извлечения элемента из очереди;
- `Peek()` – метод получения элемента, который находится в вершине очереди, не извлекая его;
- `Contains(object)` – метод проверки наличия в очереди заданного объекта;
- `Clear()` – очистка очереди.

Рассмотрим пример использования очереди. Пусть дан массив целых чисел. Распечатать элементы этого массива по группам: сначала все числа, заканчивающиеся на 0, затем – на 1, и т.д.

Для решения задачи будем использовать массив очередей. Каждая очередь из массива будет хранить коллекцию чисел, заканчивающихся на цифру, которая одновременно является номером очереди.

```
static void Reordering(int [] x)
{
   // Создаем массив очередей для хранения чисел, в которых
   // последние цифры совпадают с номером элемента массива
   Queue[] numbers = new Queue[10];
   for (int i = 0; i < 10; i++)
      numbers[i] = new Queue();
   // Распределяем числа из массива
    // по соответствующим очередям
   foreach (int el in x)
   {
      int k = el % 10;
      numbers[k].Enqueue(el);
   }
   // Печать групп чисел путем извлечения
    // элементов из очередей
   for (int i = 0; i < 10; i++)
   {
      int el = 0;
      // Пока i-тая очередь не пуста, извлекаем из нее числа
      // и печатаем их. Когда очередь станет пустой, попытка
      // извлечь элемент, приведет к возникновению
      // исключительной ситуации, произойдет выход из цикла
      try
      {
         while (true)
            Console.WriteLine("" +
                              (int)numbers[i].Dequeue() + "\t");
      }
      catch (Exception e)
      {
         Console.WriteLine();
      }
   }
}
```

Классической задачей, использующей стеки и очереди, является задача построения постфиксной формы арифметического выражения и его вычисления.

Пусть дана символьная строка, содержащая правильно записанное арифметическое выражение. Требуется написать функцию перевода выражения в постфиксную форму. Постфиксной формой выражения называется такая запись, в которой знак операции следует за операндами. При этом она не содержит скобок. Например, постфиксная форма выражение $a * b$ имеет вид $ab *$, $a * b+c$ – вид $ab * c +$, $a * (b + c)$ – вид $abc + *$.

Для решения задачи можно использовать следующий алгоритм.

Рассматриваются поочередно все символы строки. В стек записывается открывающая скобка, и выражение далее анализируется посимвольно слева направо. Если встречается операнд (число или переменная), то он сразу помещается в очередь. Если встречается открывающая скобка, то она заносится в стек, а если встречается закрывающая скобка, то из стека извлекаются находящиеся там знаки операций до ближайшей открывающей скобки, которая также удаляется из стека. Все эти знаки в порядке их извлечения помещаются в очередь. Когда же встречается знак операции, то из стека извлекаются знаки операций, приоритет которых больше или равен приоритету данной операции, и они помещаются в очередь, после чего рассматриваемый знак переносится в стек. Когда выражение заканчивается, выполняются такие же действия, что и при встрече закрывающей скобки. Постфиксная форма выражения определяется путем извлечения всех операндов и операций из очереди, а потом из стека.

```
// определение функции перевода арифметического выражения
// в постфиксную форму
// str – строка, содержащая исходное арифметическое выражение
// функция возвращает строку
// с постфиксной формой этого выражения
static string Postfix(string str)
{
   Stack<char> s = new Stack<char>();
   Queue<char> q = new Queue<char>();
   char c;
   // строка анализируется посимвольно
   for(int i=0;i<str.Length; i++)
   {
      switch(str[i])
      {
         // если текущий символ – знак операции
         case '+': case '-': case '*': case '/':
              if(s.Count == 0)
                 // если стек пустой, помещаем
                 // символ операции в стек
                 s.Push(str[i]);
              else
              {
                 bool f = true;
                 c = s.Pop();
                 if(str[i] == '+' || str[i] == '-')
                 {
                    // если текущая операция '+'  или '-',
                    // из стека в очередь перемещаются все
```

```
                // знаки операций либо до достижения
                // пустоты стека, либо до достижения
                // символа '('
                while(c != '(')
                {
                    q.Enqueue(c);
                    if(s.Count != 0)
                        c = s.Pop();
                    else
                    {
                        f = false; break;
                    }
                }
            }
            else
            {
                // если текущая операция '*' или '/',
                // из стека в очередь перемещаются все
                // знаки операций либо до достижения
                // пустоты стека, либо до достижения
                // символов '(', '+'  или '-'
                while(c == '*' || c == '/')
                {
                    q.Enqueue(c);
                    if(s.Count != 0)
                        c = s.Pop();
                    else
                    {
                        f = false;
                        break;
                    }
                }
            }
            // последний извлеченный символ помещается в
            // стек, если выход из предыдущих циклов
            // осуществился не по достижению конца стека
            if(f)
                s.Push(c);
            // в стек помещается текущая операция
            s.Push(str[i]);
        }
        break;
    // если текущий символ '(', он записывается в стек
    case '(':
        s.Push(str[i]);
        break;
    // если текущий символ ')', из стека извлекаются
    // все знаки операций до ближайшей '(',
    // которая также извлекается из стека
    case ')':
        c = s.Pop();
```

```
                while(c != '(')
                {
                    q.Enqueue(c);
                    c = s.Pop();
                }
                break;
            default:
                // текущий символ – операнд.
                // Он помещается в очередь
                q.Enqueue(str[i]); break;
        }
    }
    // формирования строки-результата, извлекая сначала
    // все из очереди, а потом из стека
    string res = "";
    while(q.Count != 0)
        res = res + q.Dequeue();
    while(s.Count != 0)
        res = res + s.Pop();
    return res;
}
```

Пусть в выражении, для которого была построена постфиксная форма, операнды являлись цифрами. Вычислим это выражение. Для этого можно использовать следующий алгоритм, в котором важную роль играет стек. Выражение в постфиксной форме просматривается слева направо. Если встречается цифра, то она заносится в стек. Если встречается знак операции, то из стека извлекаются два операнда, над ними выполняется операция и ее результат записывается в стек. Когда выражение заканчивается, в стеке остается одно число – значение выражения.

```
// функция вычисления значения выражения,
// записанного в постфиксной форме,
// с операндами-цифрами
static double CalculatePostfix(string str)
{
    // массив символов доступных операций
    char[] opers={'+','-','*','/'};
    Stack<double> s = new Stack<double>();
    for (int i = 0; i < str.Length; i++)
    {
        // если i-ый символ - операнд, помещаем его в стек
        if (!opers.Contains(str[i]))
            s.Push(double.Parse(""+str[i]));
        else
        {
            // символ является операцией. Извлекаем 2 операнда
```

```
            // и выполняем операцию
            double op1 = s.Pop();
            double op2 = s.Pop();
            switch (str[i])
            {
               case '+': s.Push(op1 + op2); break;
               case '-': s.Push(op2 - op1); break;
               case '*': s.Push(op1 * op2); break;
               case '/': s.Push(op2 / op1); break;
            }
         }
      }
      // результат выражения – последнее значение в стеке
      return s.Pop();
   }
```

6.2. Использование списков для хранения разреженных матриц

Классы, реализующие работу со списками (`ArrayList` или `List<>`), являются самыми распространенными классами-коллекциями, которые используются в приложениях. Они позволяют создавать линейный массив данных, который может легко менять свой размер путем добавления в него новых элементов или удаления из него существующих. Для списков реализован индексатор, который позволяет обращаться к элементам списка по индексу, как в массиве, что делает удобным обращение с его элементами.

Основные свойства и методы классов-списков таковы:

- `Count` – свойство, которое задает количество элементов в списке;
- `Add(object)` – метод добавления объекта в конец списка;
- `Clear()` – удаление всех элементов из списка;
- `Contains(object)` – определение, присутствует ли заданный элемент в списке;
- `IndexOf(object)`, `LastIndexOf(object)` – метод, который возвращает номер первого или последнего вхождения заданного элемента в список;
- `Insert(int, object)` – метод вставки элемента в список на заданную позицию;
- `Remove(object)` – метод удаления заданного элемента из списка;
- `RemoveAt(int)` – метод удаления элемента списка, находящегося на заданной позиции;
- и т.д., в том числе и методы поиска, сортировки, реверса элементов и прочие методы.

В качестве примера, использующего динамический список, приведем класс работы с разреженными матрицами.

Разреженной называется матрица, которая содержит большое количество нулевых элементов. Такие матрицы нередко возникают в задачах линейной алгебры, математической физики и оптимизации.

Хранение этих матриц традиционным способом требует существенных затрат памяти. Особенно это неэффективно при больших размерах матрицы и большом количестве нулевых элементов. В этом случае матрицу можно хранить в виде списка, содержащего только ненулевые элементы.

Для представления разреженной матрицы можно использовать следующую систему классов:

- класс для хранения одного элемента матрицы, который содержит индексы этого элемента и его значение (`MatrixElement`);
- класс представления всей матрицы в виде списка, содержащий размеры матрицы, количество ненулевых элементов и список ненулевых элементов матрицы (`MatrixList`);
- классы исключений `BadIndexException`, `BadDimensionException`, `NonSquareMatrixException`.

Для класса «Элемент матрицы» (`MatrixElement`) требуется определить только конструктор, инициализирующий индексы элемента матрицы и его значение, а также свойства для доступа к индексам элемента и его значению. Заметим, что индексы элемента должны быть доступны только для чтения.

```
// класс для хранения одного элемента матрицы
class MatrixElement
{
   // индексы элемента матрицы
   int i, j;
   // значение элемента матрицы
   double val;

   // конструктор элемента матрицы
   public MatrixElement(int i1, int j1, double v)
   {
      i = i1;
      j = j1;
      val = v;
   }

   // свойство получения номера строки элемента
   public int I
   {
      get { return i; }
   }
```

```
        // свойство получения номера столбца элемента
        public int J
        {
            get { return j; }
        }
        // свойство получения и установки значения элемента
        public double Value
        {
            get { return val; }
            set { val = value; }
        }
    }
```

Разреженная матрица реализуется как отдельный класс, включающий в себя список элементов матрицы – объектов класса `MatrixElement`.

```
    class MatrixList
    {
        // список элементов матрицы
        List<MatrixElement> list;
        // размеры матрицы
        int m,n;
        // количество ненулевых элементов матрицы
        int count;

        // конструктор разреженной матрицы из нулевых элементов
        public MatrixList(int m1, int n1)
        {
            m = m1;
            n = n1;
            list = new List<MatrixElement>();
            count = 0;
        }
        . . .
    }
```

Для эффективного выполнения ряда операций над разреженными матрицами удобно хранить ее элементы, упорядоченными лексикографическим образом по номерам строк и столбцов. Такое представление однозначно определяет место каждого элемента в списке, что позволяет упростить процедуру поиска элемента, находящегося в заданной позиции матрицы. Поиск элемента матрицы, расположенного в `i`-ой строке и `j`-ом столбце заключается в том, что при просмотре списка пропускаются все элементы, расположенные в

строках с меньшим номером, чем `i`, а затем – в заданной строке, но в столбцах с меньшим номером, чем `j`. Такая процедура поиска используется в индексаторе, осуществляющем доступ к элементу матрицы и в других методах класса `MatrixList`.

```
// индексатор для доступа к элементу матрицы по его индексам
public double this[int i, int j]
{
   // получение значения элемента матрицы
   get
   {

      // поиск позиции искомого элемента в списке
      int index = 0;
      // пропускаем элементы, находящиеся
      // в строках с меньшим номером
      while (index < count && i > list[index].I)
         index++;
      if (index < count && i == list[index].I)
      {
         // пропускаем элементы, находящиеся в той же
         // строке, но в столбцах с меньшим номером
         while (index < count && i == list[index].I &&
                                     j > list[index].J)
            index++;
      }
      // если элемент в заданной позиции уже
      // имеется, возвращаем его значение
      if (index < count && i == list[index].I &&
                                       j == list[index].J)
         return list[index].Value;
      // если элемента в списке нет, его значение равно 0
      return 0;
   }

   // установка значения элемента матрицы
   set
   {
      bool insert = false;
      // проверка корректности позиции устанавливаемого элемента
      if (i < m && j < n)
      {
         // если новое значение элемента нулевое,
         // удаляем элемент из списка
         if (value == 0)
         {
            DeleteElement(i, j);
            return;
         }
```

```
            // поиск позиции устанавливаемого элемента
            int index = 0;
            // пропускаем элементы, находящиеся
            // в строках с меньшим номером
            while (index < count && i > list[index].I)
               index++;
            if (index < count && i == list[index].I)
            {
               // пропускаем элементы, находящиеся в той же
                // строке, но в столбцах с меньшим номером
                while (index < count && i == list[index].I &&
                                                 j > list[index].J)
                   index++;
             }
             // если элемент в заданной позиции уже
             // имеется, изменяем его значение
             if (index < count && i == list[index].I &&
                                                 j == list[index].J)
             {
                list[index].Value = value;
                insert = true;
             }
             // если элемента с такими индексами в списке не было,
             // вставляем новый элемент в список
             if (!insert)
             {
                MatrixElement element =
                                  new MatrixElement(i, j, value);
                list.Insert(index, element);
                count++;
             }
          }
          else
             // генерация исключения в случае
             // некорректной позиции вставляемого элемента
             throw new BadIndexException(m, n);
       }
    }
```

Аналогичный подход к поиску элемента в списке используется в методах удаления заданного элемента, проверки существования элемента в списке и в операции получения строкового представления матрицы.

```
// метод удаления элемента из списка
void DeleteElement(int i, int j)
{
   // список пуст, следовательно, матрица состоит только из нулей
   if(count == 0)
```

```
            return;
        // поиск позиции в списке удаляемого элемента
        int index = 0;

        // пропускаем элементы, находящиеся
        // в строках с меньшим номером
        while (index < count && i > list[index].I)
            index++;
        if (index < count && i == list[index].I)
        {
            // пропускаем элементы, находящиеся в той же
            // строке, но в столбцах с меньшим номером
            while (index < count && i == list[index].I &&
                                                j > list[index].J)
                index++;
        }
        // если элемент в заданной позиции уже
        // имеется, удаляем его из списка
        if (index < count && i == list[index].I && j == list[index].J)
        {
            list.RemoveAt(index);
            count--;
        }
    }

    // метод получения элемента списка с заданными индексами
    public MatrixElement ExistsElement(int i, int j)
    {
        MatrixElement exists = null;
        // пропускаем элементы, находящиеся
        // до требуемого элемента
        for (int k = 0; k < count; k++)
        {
            if (list[k].I == i && list[k].J == j)
            {
                exists = list[k];
                break;
            }
            // если позиция искомого элемента пройдена, то элемента нет
            if (!(list[k].I < i || (list[k].I == i && list[k].J < j)))
                break;
        }
        return exists;
    }

    // операция получения строкового представления матрицы
    static public implicit operator string(MatrixList ob)
    {
        string res = "";
        int i = 0, j = 0;
        // цикл просмотра элементов списка
```

```
    for (int k = 0; k < ob.count; k++)
    {
       // вывод нулей в качестве элементов
       // предшествующих строк
       for (; i < ob.list[k].I; i++)
       {
          for (; j < ob.n; j++)
             res = res + "0\t";
          res = res + "\n";
          j = 0;
       }
       // вывод нулей в качестве элементов в той же строке,
       // но в предшествующих столбцах
       for (; j < ob.list[k].J; j++)
          res = res + "0\t";
       // вывод текущего элемента
       res = res + ob.list[k].Value + "\t";
       // корректировка индексов для просмотра
       // следующих элементов
       j++;
       if (j == ob.n)
       {
          i++;
          j = 0;
          res = res + "\n";
       }
    }
    // вывод нулей в качестве последующих элементов строки,
    // в которой расположен последний элемент списка
    if (j != 0)
    {
       for (; j < ob.n; j++)
          res = res + "0\t";
       res = res + "\n";
       i++;
    }
    // вывод нулей в качестве элементов строк,
    // расположенных после той,
    // в которой находится последний элемент списка
    for (; i < ob.m; i++)
    {
       for (j = 0; j < ob.n; j++)
          res = res + "0\t";
       res = res + "\n";
    }
    return res;
}
```

Добавим в класс `MatrixList` функции, осуществляющие операции с матрицами, например, сложение двух матриц и функцию определения, является ли матрица трехдиагональной.

Сложение двух матриц заключается в создании нового списка на основании двух существующих. Если оба исходных списка содержат элементы с одинаковыми индексами, сумма их значений образует соответствующий элемент нового списка. Остальные элементы обоих списков просто дублируются в новом.

```
// операция сложения двух матриц
public static MatrixList operator+(MatrixList ob1, MatrixList ob2)
{
   // матрицы должны иметь одинаковые размеры
   if(ob1.m != ob2.m || ob1.n != ob2.n)
      throw new BadDimensionException
                              (ob1.m, ob1.n, ob2.m, ob2.n);
   // создается матрица-результат
   // как копия первого слагаемого
   MatrixList res = new MatrixList(ob1.m, ob1.n);
   // вызов метода копирования списка класса List
   res.list.AddRange(ob1.list);
   res.count = ob1.count;
   // просмотр элементов второй матрицы
   for (int i = 0; i < ob2.count; i++ )
   {
      MatrixElement exists = res.ExistsElement
                              (ob2.list[i].I, ob2.list[i].J);
      if (exists != null)
         // если в матрице-результате элемент с такими
         // индексами уже имеется, суммируем элементы
         exists.Value += ob2.list[i].Value;
      else
         // если в матрице-результате элемент с
         // такими индексами не существует,
         // добавляем новый элемент в матрицу-результат
         res[ob2.list[i].I, ob2.list[i].J] = ob2.list[i].Value;
    }
    return res;
 }
```

Матрица является трехдиагональной, если ее ненулевые элементы расположены только на главной диагонали и на двух соседних, параллельных ей.

```
// метод проверки, является ли матрицы трехдиагональной
bool IsTripleDiagonal()
{
   // если матрица неквадратная, генерируется исключение
   if(m != n)
      throw new NonSquareMatrixException();
   MatrixElement exists1;
   // в каждой строке проверяется наличие ненулевых
   // элементов, расположенных до трех центральных
   // диагоналей и после них. Если ненулевой элемент будет
   // найден, матрица не является трехдиагональной.
   for(int i = 0; i < m; i++)
   {
      for(int j = 0; j < i - 1; j++)
      {
         exists1 = ExistsElement(i, j);
         if(exists1 != null)
            return false;
       }
       for(int j = i + 2; j < n; j++)
       {
          exists1 = ExistsElement(i, j);
          if(exists1 != null)
             return false;
       }
    }
    return true;
 }
```

Как видно из программного кода методов, при возникновении ошибочных ситуаций, когда те или иные операции с матрицами будут невыполнимы, генерируются исключения. Для понятного информирования пользователя о возникшем исключении удобно создать собственную иерархию классов-исключений. Например, она может быть такой:

```
// класс исключения обращения
// к несуществующему элементу матрицы
class BadIndexException : Exception
{
   int m1, n1;          // размеры матрицы

   public BadIndexException(int m1_, int n1_)
   {
      m1 = m1_;
      n1 = n1_;
   }
```

```
        // переопределенное свойство сообщения об исключении
        public override string  Message
        {
            get
            {
                return string.Format("Матрица состоит из {0} строк и {1}
                                                        столбцов",m1,n1);
            }
        }
    }

    // класс исключения некорректных размеров матриц при сложении
    class BadDimensionException : Exception
    {
        int m1, n1, m2, n2;  // несовпадающие размеры двух матриц


        public BadDimensionException(int m1_, int n1_, int m2_, int
n2_)
        {
            m1= m1_; n1 = n1_;
            m2= m2_; n2=n2_;
        }

        // переопределенное свойство сообщения об исключении
        public override string  Message
        {
            get
            {
                return string.Format("Невозможно выполнить операцию над
                    матрицами размера {0}x{1} и {2}x{3}",m1,n1, m2, n2);
            }
        }
    }

    // класс исключения некорректных размеров матрицы
    // для операций с квадратными матрицами
    class NonSquareMatrixException : Exception
    {
        public NonSquareMatrixException()
        {}

        // переопределенное свойство сообщения об исключении
        public override string Message
        {
            get
            {
                return "Матрица не является квадратной";
            }
        }
    }
```

6.3. Использование словарей для создания телефонной книги

Еще одним распространенным типом структур данных является словарь, который реализован с помощью классов `HashTable` и `Dictionary<T>`. Словари представляют собой коллекции, в которых для хранения объектов используется принцип хеширования. Суть хеширования состоит в том, что для определения уникального значения (хеш-кода), используется значение соответствующего ему ключа. Хеш-код затем используется в качестве индекса, по которому в словаре отыскиваются данные, соответствующие этому ключу. Преобразование ключа в хеш-код выполняется автоматически.

Данный вид коллекции удобно применять тогда, когда данные определяются некоторым ключевым полем. Это поле становится индексом элемента в коллекции. Заметим, что в качестве ключевого поля может быть выбран объект любого типа данных, например, строка, число или объект класса.

Этот тип хранения информации позволяет сокращать время выполнения таких операций, как поиск, считывание и запись данных, даже для больших объемов информации.

Классы, которые реализуют словари, обладают следующими методами и свойствами:

- `ContainsKey(key)` – метод проверки наличия записи с заданным ключом в словаре;
- `ContainsValue(value)` – метод проверки наличия записи с заданным значением в словаре;
- `Keys` – свойство, с помощью которого можно получить доступ к списку всех ключей словаря;
- `Values` – свойство, с помощью которого можно получить доступ к списку всех значений в словаре;
- `Add(key, value)` – метод добавления новой записи в словарь;
- `Remove(key)` – метод удаления записи, соответствующей заданному ключу

- и пр.

Обратиться к элементу хэш-таблицы по ключу можно с помощью следующего синтаксиса: `имя_хэш_таблицы[ключ]`.

Продемонстрируем принципы работы со словарями на примере приложения работы с телефонной книгой.

Нередко возникают приложения, работающие с большим количеством структурированной информации, объем которой постоянно меняется. Для таких приложений основной функцией является поиск информации по заданным критериям. Поэтому выбор используемой динамической структуры данных должен быть обусловлен эффективностью выполнения операции поиска информации.

Простым примером подобных приложений является «Телефонная книга». В этом приложении должно храниться произвольное количество записей о контактах (имя абонента и его телефон). Основной функцией приложения является поиск телефона нужного абонента.

Для хранения информации о контактах телефонной книги реализуем собственную хэш-таблицу, элементами которой будут являться «страницы» телефонной книги.

В самом простом случае все данные, которые хранятся в хэш-таблице, разбиваются на определенное количество групп, для хранения каждой из которых используется отдельная область памяти. Определение номера группы, в которую должны быть помещены данные, происходит по значению некоторого ключевого поля с помощью специальной хэш-функции. Поскольку хэш-функция однозначно определяет номер группы, в которую попадает конкретная запись, при ее последующем поиске значительно сокращается количество просматриваемых данных (достаточно просмотреть только одну группу хэш-таблицы). Для эффективного поиска желательно подбирать хэш-функцию таким образом, чтобы обеспечить равномерное заполнение всех групп.

Для хранения телефонной книги будем использовать хэш-таблицу из 33 групп (по количеству букв русского алфавита). Ключевым значением поиска будет являться имя абонента. Хэш-функция будет определять номер группы по первой букве этого имени. Таким образом, как и в бумажных телефонных книгах, записи будут сгруппированы по начальным буквам имен абонентов.

Данная хэш-функция является простой, однако она не обеспечивает равномерного распределения информации по группам: некоторые буквы гораздо чаще встречаются в именах, чем другие.

При реализации хэш-таблицы хранение каждой группы будем осуществлять в виде односвязного списка, а сама хэш-таблица будет представлять собой словарь из этих списков с ключом – буквой. Таким образом, приложение «Телефонная книга» содержит два класса:

- класс информации об абоненте (`Info`);
- класс для хранения хэш-таблицы (`PhoneBook`).

Приведем код класса информации об абоненте:

```
class Info
{
   string fio;          // имя абонента
   string phone;        // номер телефона

   // конструктор с инициализацией данных об абоненте
   public Info(string f, string p)
   {
      fio = f;
      phone = p;
   }

   // конструктор по умолчанию
   public Info()
   {
      fio = "";
      phone = "";
   }
    // свойство доступа к имени абонента
   public string Fio
   {
      get { return fio; }
   }

   // свойство доступа к телефону
    public string Phone
   {
      get { return phone; }
   }
    // операция получения строкового представления записи
   public static implicit operator string(Info ob)
   {
      return "Абонент: "+ob.fio + " Телефон: " + ob.phone;
   }
}
```

Как видно, класс `Info` не представляет особой сложности и может быть легко дополнен новыми полями и свойствами. Остановимся подробнее на классе `PhoneBook`. Класс содержит словарь, в котором буквам соответствуют списки записей об абонентах, чьи имена начинаются с этой буквы. Данный словарь, а также пустые списки записей об абонентах, создаются в конструкторе класса `PhoneBook`:

```
// класс хэш-таблицы в виде массива списков
class PhoneBook
{
   // хэш-таблица записей об абонентах
   Dictionary<char, List<Info>> book;
   // конструктор хэш-таблицы
   public PhoneBook()
   {
      book = new Dictionary<char, List<Info>>();
      // создание списка (группы абонентов) для каждой буквы
      for (char c = 'А'; c <= 'Я'; c++)
         book.Add(c, new List<Info>());
   }
  . . .
}
```

Основные операции с телефонной книгой заключаются в добавлении нового абонента, удалении существующего, получении информации о заданном абоненте и проверке существования заданного абонента. Данные операции реализованы в виде соответствующих методов.

```
// метод добавления новой записи об абоненте
public void PushAbonent(string abonent, string phone)
{
   // ключ группы соответствует первой букве в имени абонента
   // буква задана в верхнем регистре
   char key = Char.ToUpper(abonent[0]);
   // добавление записи с информацией об абоненте (объект Info)
   // в группу с ключом key
   book[key].Add(new Info(abonent.ToUpper(), phone));
}


// метод удаления из списка записи по имени абонента
public void DeleteAbonent(string abonent)
{
   // ключ группы соответствует первой букве в имени абонента
   char key = Char.ToUpper(abonent[0]);
```

```
        // поиск записи с информацией об абоненте
        // и удаление его из списка с ключом key
        Info info;
        if (FindAbonent(abonent, out info))
            book[key].Remove(info);
    }


    // метод поиска записи по имени абонента
    // информация об абоненте возвращается через out-параметр info

    public bool FindAbonent(string abonent, out Info info)
    {
        // ключ группы соответствует первой букве в имени абонента
        char key = Char.ToUpper(abonent[0]);
        // возвращаем true, если абонент найден в группе key,
        // false – в противном случае
        abonent = abonent.ToUpper();
        for (int i = 0; i < book[key].Count; i++)
            if (book[key][i].Fio == abonent)
            {
                info = book[key][i];
                return true;
            }
        info = null;
        return false;
    }

    // метод проверки наличия в хэш-таблице
    // заданного имени абонента
    public bool HasAbonent(string abonent)
    {
        Info info;
        return FindAbonent(abonent, out info);
    }
```

Все методы, реализующие операции с записью хэш-таблицы, начинаются с вычисления хэш-функции, т.е. ключа группы, который соответствует этой записи. Далее в этой группе производится требуемая операция.

Печать телефонной книги осуществляется с помощью операции получения строкового представления списка записей книги:

```
  static public implicit operator string(PhoneBook ob)
  {
      string res = "Телефонная книга\n\n";
      // перебор всех ключей телефонной книги
      foreach (char c in ob.book.Keys)
          // если в группе, соответствующей ключу, есть записи,
```

```
        // выводим их в строку-результат
        if (ob.book[c].Count != 0)
        {
           // печать буквы-ключа
           res = res + c + ":\n\n";
            // печать всех записей из списка группы
           for (int i = 0; i < ob.book[c].Count; i++)
              res = res + "\t" + ob.book[c][i]+"\n";
           res = res + "\n\n";
        }
     return res;
  }
```

Приведем далее функцию `Main()` приложения, которое формирует и использует телефонную книгу. Приложение выводит меню с операциями, которые можно выполнять с телефонной книгой, и осуществляет выбранную операцию. Для окончания работы приложения в меню содержится команда `"Exit"`.

```
  static int Menu()
  {
     int k = 0;
     while (k <= 0 || k > 6)
     {
        Console.WriteLine("----------- Меню ------------");
        Console.WriteLine("Добавить абонента - 1,\n
                           Удалить абонента - 2,\n
                           Найти абонента - 3,\n
                           Проверка существования абонента - 4,\n
                           Печать телефонной книжки - 5,\n
                           Выход - 6");
        Console.WriteLine("Введите команду:");
        k = int.Parse(Console.ReadLine());
     }
     return k;
  }

  static void Main(string[] args)
  {
     PhoneBook phoneBook = new PhoneBook();
     string phone;
     string str;
     Info i;
     while (true)
     {
        switch (Menu())
        {
           case 1:        // вставка новой записи об абоненте
               Console.WriteLine("--- Добавление абонента ---");
```

```
                Console.Write("Введите имя:");
                str = Console.ReadLine();
                if (phoneBook.HasAbonent(str) == true)
                {
                   Console.WriteLine("Такой абонент уже есть");
                   break;
                }
                Console.Write("Введите номер телефона:");
                phone = Console.ReadLine();
                phoneBook.PushAbonent(str, phone);
                break;
            case 2:        // удаление записи по имени абонента
                Console.WriteLine("--- Удаление абонента -----");
                Console.Write("Введите имя:");
                str = Console.ReadLine();
                phoneBook.DeleteAbonent(str);
                break;
            case 3:        // поиск телефона заданного абонента
                Console.WriteLine("---- Поиск абонента -------");
                Console.Write("Введите имя:");
                str = Console.ReadLine();
                if (phoneBook.FindAbonent(str, out i) == false)
                   Console.WriteLine("Абонента не существует");
                else
                   Console.WriteLine(i);
                break;
            case 4: // проверка существования абонента
                    // с заданным именем
                Console.WriteLine("- Существование абонента -");
                Console.Write("Введите имя:");
                str = Console.ReadLine();
                if (phoneBook.HasAbonent(str) == false)
                   Console.WriteLine("Абонента не существует");
                else
                   Console.WriteLine("Абонент существует");
                break;
            case 5: // печать телефонной книги
                Console.WriteLine(phoneBook);
                break;
            case 6: // выход из приложения
                return;
        }
    }
}
```

6.4. Язык запросов LINQ на примере приложения «Магазин»

Одна из типовых задач работы с коллекциями – это задача поиска данных, удовлетворяющих определенным условиям. Именно для этих целей в языке C# был создан специальный язык запросов LINQ (Language Integrated Query), в чем-то схожий с языком запросов для баз данных SQL. Язык LINQ предоставляет универсальный способ выборки данных независимо от того, каков их источник – различные коллекции (массивы, списки, словами), xml-документы, базы данных и пр.

Запрос указывает, какую информацию нужно извлечь из источника данных. В запросе могут быть указаны способ сортировки и группировки данных. Запрос хранится в переменной и инициализируется выражением запроса, которое содержит три основные части: `from`, `where` и `select`. Часть `from` указывает, из каких источников следует выбирать данные, часть `where` предназначена для задания условий отбора данных, наконец, часть `select` указывает, каким образом выбранные данные должны храниться в результате запроса.

Переменная запроса не имеет четкого типа данных (указывается тип `var`). Выполнение запроса происходит в момент его обработки или кэширования. Обработка запроса предусматривает организацию цикла просмотра выбранных данных. Например, в следующем примере из массива целых чисел выбираются только четные элементы.

```
// формирование источника данных для запроса
int[] array = new int[] { 4, 5, 2, 0, 3, 9, 1, 7, 8, 6 };

// формирование запроса получения всех четных элементов массива
var query = from el in array where (el % 2) == 0 select el;

// выполнение запроса и печать элементов из выборки
foreach (int a in query)
   Console.Write("{0} ", a);
```

В части `from` данного запроса указывается, что просматриваются все элементы `el` из массива `array`. В части `where` задано условие отбора элементов (для которых `el%2==0`). Часть `select` указывает, что в выборку входят сами элементы `el`. Собственно выполнение запроса осуществляется при выполнении следующего за запросом цикла.

Можно выполнять запрос и без цикла `foreach`. Например, это можно сделать с помощью агрегирующих функций `Count()`, `Average()`, `Max()` и др. Так, при получении максимального четного числа из массива выполнение запроса могло бы быть таким:

```
  // формирование источника данных для запроса
  int[] array = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  // формирование запроса получения всех четных элементов массива
  var query = from el in array where (el % 2) == 0 select el;

  // выполнение запроса и печать максимального
  // из элементов в выборке
  Console.WriteLine("Максимальный четный элемент - {0}",
query.Max());
```

Другим способом выполнения запроса без цикла является кэширование результата запроса в массиве или списке, который можно обработать позднее. Это делается с помощью методов запроса `ToArray()` или `ToList()`. Например,

```
  int[] res = query.ToArray();
```

Подробнее про язык запросов можно прочитать, например, в [9]. Мы же рассмотрим пример приложения, в котором используются коллекции и требуется выбор данных из этих коллекций. Для осуществления этого выбора будем использовать язык запросов.

Требуется написать приложение, которое отслеживает работу магазина. Имеется каталог товаров, которые могут продаваться в магазине. Каждый товар характеризуется категорией, названием и ценой. Данные о товарах хранятся в файле:

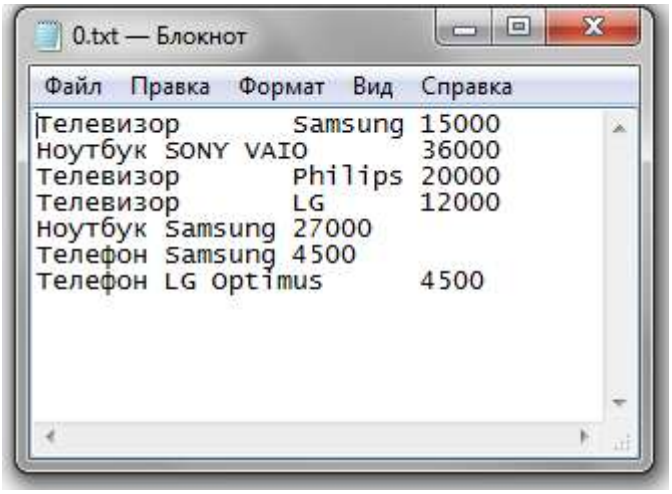


Рис.6.4. Файл с информацией о товарах

Для каждого товара в файле отводится одна строка, в которой данные разделены символом табуляции.

Для хранения информации о товаре в приложении создан класс `Tovar`:

```
// класс описания товара
class Tovar
{
    string category;     // категория товара
    string name;         // название товара
    int price;           // цена товара

    // конструктор класса
    public Tovar(string c, string n, int p)
    {
        category = c;
        name = n;
        price = p;
    }


    // свойства для доступа к полям класса
    public string Category
    {
        get { return category; }
    }


    public string Name
    {
        get { return name; }
    }
```

```
        public int Price
        {
            get { return price; }
        }


        // получение информации о товаре из символьной строки
        static public Tovar Parse(string str)
        {
            // разделение строки по символу табуляции
            string[] s = str.Split('\t');
            // создание и возврат объекта-товара из данных строки
            Tovar t = new Tovar(s[0], s[1], int.Parse(s[2]));
            return t;
        }

        // операция получения строки с информацией о товаре для печати
        static public implicit operator string(Tovar t)
        {
            return t.category + " " + t.name + " Цена:" +
                                                t.price + " рублей";
        }


        // переопределение функции получения строки с
        // информацией о товаре для сохранения в файл
        public override string ToString()
        {
            return category + "\t" + name + "\t" + price;
        }
    }
```

Для работы с каталогом товаров и поиска в нем нужной информации создан класс `PriceList`, который хранит список объектов `Tovar`, которые могут продаваться в магазине, и имеет методы для поиска в каталоге товаров, удовлетворяющих различным критериям поиска.

```
    // класс для описания списка товаров, которые продаются в магазине
    class PriceList
    {
        List<Tovar> list;

        // конструктор
        public PriceList(string file)
        {
            // создание списка товаров по информации из файла
            list = new List<Tovar>();
            // открытие файла для чтения
```

```
            StreamReader sr = new StreamReader(file);
            string str;
            // считывание файла построчно
            while ((str = sr.ReadLine()) != null)
            {
                // получение товара из строки с его информацией
                Tovar t = Tovar.Parse(str);
                // добавление товара в список
                list.Add(t);
            }
            sr.Close();
        }
        . . .
    }
```

Методы печати информации по выбранным критериям имеют единую схему:

- формируется запрос на выбор из списка тех товаров, которые удовлетворяют требуемым условиям;
- проверяется, не пуст ли результат выборки;
- если выборка не пуста, печатаются все выбранные элементы, в противном случае выводится сообщение о том, что по заданным критериям поиска товаров не найдено.

По той же схеме работает и метод получения заданного товара (по категории и названию).

```
// получение объекта товара по названию и категории
public Tovar GetTovar(string c, string n)
{
    // формирование и кэширование в список запроса
    // но поиск в списке товаров того, который
    // имеет заданную категорию и название
    List<Tovar> items = (from s in list
                         where s.Category.Equals(c) &&
                               s.Name.Equals(n)
                         select s).ToList<Tovar>();
    // если товаров не найдено, генерируется исключение
    if (items.Count() == 0)
        throw new Exception("Такого товара нет на складе");
    // товар найден – возвращаем его объект
    return items[0];
}
```

Магазин имеет склад товаров. Информация о товарах, которые хранятся на складе, записана в текстовый файл (Рис. 6.5).

Через символ-разделитель, которым в данном случае является ‘!’, в файл записаны категория товара, его название и количество на складе.

Для управления складом в приложение добавим класс `Sklad`, содержащий словарь, в котором объекту товара ставится в соответствие количество данного товара на складе. Информация в словарь загружается из файла в конструкторе класса `Sklad`:

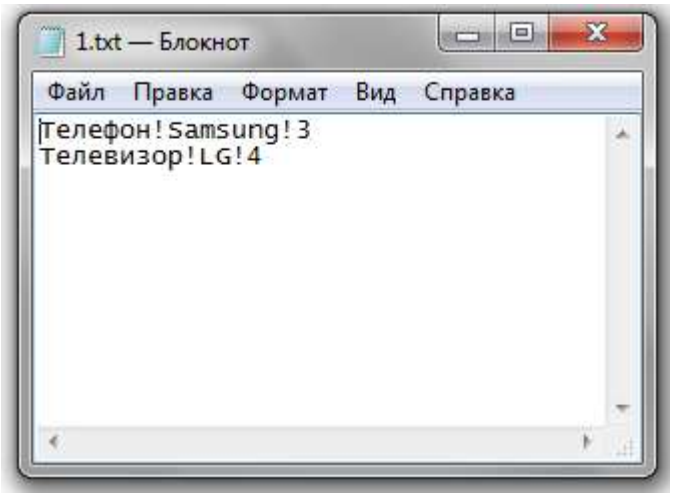


Рис.6.5. Файл с информацией склада.

```
// класс для описания работы склада
class Sklad
{
   // словарь, который содержит информацию
   // о  наличии товаров на складе
   Dictionary<Tovar, int> sklad;

    // конструктор класса
   public Sklad(string file, PriceList l)
   {
      sklad = new Dictionary<Tovar, int>();
      StreamReader sr = new StreamReader(file);
      string str;
       // считывание строки из файла
      while ((str = sr.ReadLine()) != null)
      {
          string[] s = str.Split('!');
          // получение объекта-товара по категории и названию
          Tovar t = l.GetTovar(s[0],s[1]);
          // добавление в словарь записи о товаре
          sklad.Add(t, int.Parse(s[2]));
      }
      sr.Close();
   }
  . . .
```

```
}
```

Класс `Sklad` также имеет методы, которые регистрируют операции поступления товара на склад (`AddTovar()`) и реализации товара покупателю (`SaleTovar()`). Метод `AddTovar()` с помощью запроса находит в словаре запись о поступившем товаре. Если такого товара на складе не было, добавляется новая запись об этом товаре, в противном случае в найденной записи корректируется количество с учетом поступления.

```
// метод, регистрирующий поступление товара на склад
public void AddTovar(Tovar t, int count)
{
   // запрос к словарю на поиск записи с заданным ключом-товаром
    List<Tovar> items = (from s in sklad
                         where s.Key.Equals(t)
                         select s.Key).ToList<Tovar>();
   if (items.Count() == 0)
   {
      // товара не было найдено -
       // добавляем информацию о его поступлении
      sklad.Add(t, count);
      return;
   }
   // товар уже есть на складе - увеличиваем его количество
   sklad[t] = sklad[t] + count;
}
```

Алгоритм метода `SaleTovar()` предусматривает поиск записи в словаре, соответствующей продаваемому товару. Отсутствие такой записи приводит к генерации исключения (товара нет на складе). Аналогичное исключение возникает, когда товар имеется, но в недостаточном количестве. Если же данные корректны, в найденной записи словаря изменяется количество товара с учетом реализации.

```
// метод, регистрирующий покупку товара
public void SaleTovar(Tovar t, int count)
{
   List<KeyValuePair<Tovar, int>> items =
                  (from s in sklad where s.Key.Equals(t)
                   select s).ToList <KeyValuePair<Tovar, int>>();
   if(items.Count() == 0)
      throw new Exception("Необходимого товара нет на складе");
   foreach (KeyValuePair<Tovar,int> p in items)
   {
      if (p.Value < count)
```

```
            throw new Exception("Необходимого количества
                                                нет на складе");
         sklad[p.Key] = sklad[p.Key] - count;
         // если весь товар реализован, удаляем запись о нем
         if (sklad[p.Key] == 0)
            sklad.Remove(p.Key);
         return;
      }
   }
```

Чтобы узнать, сколько товара имеется на складе, добавим метод `CountTovar()`:

```
   // метод получения количества заданного товара на складе
   public int CountTovar(Tovar t)
   {
      // получение количества из записи с ключом-товаром
      List<int> count = (from s in sklad where s.Key.Equals(t)
                         select s.Value).ToList<int>();
      // если результат запроса пуст, товара нет на складе
      if (count.Count == 0)
         return 0;
      // возвращаем найденное количество
      return count[0];
   }
```

Помимо указанных методов удобно добавить в класс `Sklad` метод получения символьного представления списка товаров на складе, метод записи в файл и метод получения списка всех товаров, которые имеются на складе. Сложностей в написании данные методы не представляют.

Покупка оформляется в виде заказа, указывающего, какой товар и в каком количестве требуется покупателю. Для хранения информации о заказе создадим класс `Zakaz`:

```
   // класс для описания заказа покупателя
   class Zakaz
   {
      Tovar t;         // заказанный товар
      int count;       // количество


      // конструктор класса
      public Zakaz(Tovar a, int c)
      {
         t = a;
         count = c;
      }
```

```
    // свойства для получения доступа к полям заказа
    public Tovar tovar
    {
        get { return t; }
    }



    public int Count
    {
        get { return count; }
        set { count = value; }
    }

    // операция получения строкового представления заказа
    static public implicit operator string(Zakaz ob)
    {
        return ""+ob.t.Category+"!"+ob.t.Name + "!" + ob.count;
    }
}
```

Поступившие заказы фиксируются в списке заказов, для которого имеется соответствующий класс. Список текущих заказов сохраняется и загружается в текстовый файл:

```
// класс для описания списка заказов
class ListZakaz
{
    List<Zakaz> list;

    // конструктор, считывающий информацию
    // о невыполненных заказах из файла
    public ListZakaz(string file, PriceList l)
    {
        list = new List<Zakaz>();
        StreamReader sr = new StreamReader(file);
        string str;
        while ((str = sr.ReadLine()) != null)
        {
            string[] s = str.Split('!');
            Tovar t = l.GetTovar(s[0],s[1]);
            Zakaz z = new Zakaz(t,int.Parse(s[2]));
        }
        sr.Close();
    }
   . . .
}
```

Новый заказ добавляется в список с помощью метода `AddZakaz()`. Производится обслуживание не конкретного заказа, а сразу нескольких заказов на конкретный товар. В функции `RemoveZakaz()` с помощью запроса выбираются все заказы на заданный товар, далее они просматриваются, пока не будет исчерпано количество товара. Если товара на складе достаточно для полного выполнения заказа, он считается обслуженным и удаляется из списка заказов. Возможно частичное выполнение заказа, если в наличии товара меньше, чем требовалось покупателю.

```
// метод добавления заказа в список
public void AddZakaz(Tovar t, int count)
{
   Zakaz z = new Zakaz(t, count);
   list.Add(z);
}

// метод выполнения заказа и удаления его из списка
public void RemoveZakaz(Tovar t, int count)
{
   // заказов может быть выполнено несколько,
   // поэтому обращаемся к ним в цикле
   while (true)
   {
      // если количество исчерпано, заканчиваем метод
      if (count == 0) return;
      // ищем заказы на данный товар
      List<Zakaz> items = (from s in list
                           where s.tovar.Equals(t)
                           select s).ToList<Zakaz>();
      // если заказов нет, заканчиваем реализацию
      if (items.Count() == 0) return;
      foreach (Zakaz z in items)
      {
         if (z.Count <= count)
         {
            // заказ выполнен полностью - корректируем
            // количество и удаляем заказ из списка
            count -= z.Count;
            list.Remove(z);
            break;
         }
         else
         {
            // заказ выполнен не полностью
            z.Count -= count;
            count = 0;
            break;
```

```
            }
        }
    }
}
```

Для получения количества заказанных товаров, добавим еще один метод:

```
// метод получения количества заказов на данный товар
public int CountTovar(Tovar t)
{
   // получение количества товара, которое уже заказано
   List<int> items = (from s in list where s.tovar.Equals(t)
                      select s.Count).ToList<int>();
   // подсчет суммы этого количества
   int count = 0;
   foreach (int a in items)
      count += a;
   return count;
}
```

Основной класс приложения – класс `Shop`, который объединяет в себе все операции, выполняемые в магазине.

```
// класс описания магазина
class Shop
{
    Sklad sklad;         // объекта склада
    ListZakaz list;      // объект для списка заказов
    PriceList tovars;    // каталог товаров
    // конструктор
    public Shop()
    {
       // создание объектов и загрузка данных из файлов
       tovars = new PriceList("../../0.txt");
       sklad = new Sklad("../../1.txt", tovars);
       list = new ListZakaz("../../2.txt", tovars);
    }

    // деструктор - записывает последнюю информацию
     // о заказах и состоянии склада в файлы
    ~Shop()
    {
       sklad.WriteFile("../../1.txt");
       list.WriteFile("../../2.txt");
    }
   . . .
}
```

В данном классе содержится большое число методов, которые являются посредниками при вызове методов каталога товаров. Эти методы не нуждаются в комментариях. Главными же являются методы двух основных операций с товарами – поступление нового заказа и поступление товара на склад. В первом случае нужно проверить наличие товара на складе. В зависимости от того, выполнен заказ полностью или частично, будет добавлена запись в список заказов на недостающее количество. Второй метод предполагает, что требуется инспектировать список заказов на данный товар. В результате заказы могут быть выполнены полностью или частично, а оставшаяся часть товара должна быть отправлена на склад.

```
// оформление заказа
public void Zakaz(string cat,string tov,int c)
{
   try
   {
      Tovar t = tovars.GetTovar(cat, tov);
      // определение количества товаров на складе
      int count_sklad = sklad.CountTovar(t);
      if (count_sklad >= c)
      {
         // если товар на складе есть, обслуживаем заказ сразу
         Console.WriteLine("Заказ обработан");
         // меняем количество товара на складе
         sklad.SaleTovar(t, c);
      }
      else
      {
         if (count_sklad > 0)
         {
            Console.WriteLine("Заказ обработан частично
                                        ({0})", count_sklad);
            // запись оставшейся части заказа
            list.AddZakaz(t, c - count_sklad);
            // уменьшение товара на складе
            sklad.SaleTovar(t, count_sklad);
         }
         else
         {
            // добавление заказа в список
            list.AddZakaz(t, c);
         }
      }
   }
```

```
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}

// поступление товаров от поставщиков
public void Postavka(string cat,string tov,int c)
{
    try
    {
        Tovar t = tovars.GetTovar(cat, tov);
        // определение количества товара, на которое
        // уже есть заказы
        int count_zakaz = list.CountTovar(t);
        if (c <= count_zakaz)
        {
            // если заказов больше, чем поступивших товаров,
            // корректируем список заказов
            list.RemoveZakaz(t, c);
        }
        else
        {
            if (count_zakaz > 0)
            {
                // можно частично обработать заказы
                list.RemoveZakaz(t, count_zakaz);
                // оставшееся количество товара
                // отправляем на склад
                sklad.AddTovar(t, c - count_zakaz);
            }
            else
            {
                // заказов на данный товар нет,
                // приходуем все на склад
                sklad.AddTovar(t, c);
            }
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Задания для самостоятельной работы

1. Разработать класс «Полином», в котором информация о коэффициентах хранится в виде списка. Реализовать для класса методы ввода-вывода, сложения и умножения полиномов, умножения полинома на число, интегрирования и дифференцирования полинома.

2. Использовать классы стека и очереди для решения следующих задач:

- Дана символьная строка с некоторым выражением, в котором могут содержаться скобки трех видов – (), {}, []. Написать метод проверки правильности расстановки скобок в этой строке.
- Дана символьная строка, которая содержит правильное скобочное выражение. Для каждой пары скобок (открывающей и соответствующей ей закрывающей) распечатать номера их позиций в строке, упорядочив пары: а) по возрастанию номеров открывающих скобок; б) по возрастанию номеров закрывающих скобок.
- Дана символьная строка, содержащая правильно записанное математическое выражение следующего вида:

```
<формула>::=<цифра>|M(<формула>,<формула>)
                   |m(<формула>,<формула>)
```

`M` – операция вычисления `max` из двух выражений, `m` - операция вычисления `min` из двух выражений. Написать функцию вычисления значения этого выражения.

- Дана символьная строка, содержащая правильно записанное логическое выражение следующего вида:

```
<формула> ::= T | F |
          And(<формула> , <формула>) |
          Or(<формула> , <формула>) |
          Not (<формула>)
```

`And` – операция логического И, `Or` – операция логического ИЛИ, `Not` - операция логического НЕ.

Написать функцию вычисления этого выражения (функция

должна возвращать `true`, если значение выражения равно T, `false` – в противном случае).

3. Описать класс «Предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых это слово встречается. Предусмотреть возможность формирования указателя с клавиатуры и из файла, печати предметного указателя, сохранения в файл, вывода номеров страниц для заданного слова, добавления и удаления элемента из указателя.

4. Описать класс «Каталог библиотеки». Каждая запись каталога содержит информацию о книге – название, автор, количество экземпляров, количество экземпляров «на руках». Предусмотреть возможность формирования каталога с клавиатуры и из файла, печати каталога, сохранения в файл, поиска книги по какому-либо признаку (например, автору или названию), добавления книг в библиотеку, удаления книг из нее, операции получения или возврата книги читателем.

5. Описать класс «Расписание занятий». Каждая запись содержит день недели, время, название учебной дисциплины, аудиторию. Предусмотреть возможность формирования расписания с клавиатуры и из файла, печати всего расписания и расписания на конкретный день (печать должна быть осуществлена в хронологическом порядке), добавления и удаления записей, сохранения в файл.

6. Описать класс «Расписание приема пациентов». Каждая запись содержит дату, время, фамилию пациента. Время приема одного пациента должно быть равно одному часу. Предусмотреть возможность формирования расписания с клавиатуры и из файла, печати всего расписания, или расписания в конкретный день, добавления и удаления записей, сохранения в файл. При добавлении записи следует учитывать, что время записи должно быть свободно (не существует уже созданной записи с этим же временем).

Литература

1. Пышкин, Е.В. Основные концепции и механизмы объектно-ориентированного программирования [Текст]/ Е.В.Пышкин. – СПб: БХВ-Петербург, 2005. – 640 с.

2. Шилдт, Г.. C# 4.0: полное руководство [Текст]: Пер. с англ. / Герберт Шилдт. – М.: ООО "И.Д. Вильямс", 2011. – 1056 с.

3. Дейтел, Х. C# в подлиннике. Наиболее полное руководство [Текст]: пер. с англ./ Харви Дейтел, Пол Дейтел. – СПб: БХВ-Петербург, 2006 г.. – 1056 с.

4. Троелсен, Э. Язык программирования C# 2010 и платформа .NET 4 [Текст]: Пер. с англ. / Эндрю Троелсен. – М.: ООО "И.Д. Вильямс", 2011. – 1392 с.

5. Уотсон, К. Visual C# 2010: полный курс [Текст]: Пер. с англ./ Карли Уотсон, Кристиан Нейгел, Якоб Хаммер Педерсен, Джон Д. Рид, Морган Скиннер. – М.: Диалектика, 2010. – 960 с.

6. Трей, Нэш C# 2010: ускоренный курс для профессионалов [Текст]: Пер. с англ./ Нэш Трей. - М.: ООО "И.Д. Вильямс", 2011. – 592 с.

7. Кубенский, А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++ [Текст] / А.А. Кубенский. – СПб: БХВ-Петербург, 2004. – 464 с.

8. Вирт, Н. Алгоритмы и структуры данных [Текст]: пер. с англ. / Никлаус Вирт. – СПб: Невский Диалект, 2008. – 352 с.

9. Сайт центра разработки на Visual C# [Интернет-ресурс]. URL: http://msdn.microsoft.com/ru-ru/vcsharp/. Дата обращения: 10.11.2011.