СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Современные среды программирования Windows приложений используют огромные библиотеки различных компонентов или управляющих элементов, позволяющих на этапе визуального программирования проектировать практически любые пользовательские интерфейсы. Особенно удобный набор компонентов использует среда визуального программирования Delphi. Естественно Андерс Хейлсберг, который долгое время руководил разработками языков программирования фирмы Borland, не мог не учесть эту особенность при разработке среды визуального программирования Visual Studio.Net для создаваемого языка программирования C#.

Поэтому во второй части учебного пособия изложения материала начинается с изучения технологий программирования предоставляемых средой визуального программирования Visual Studio.NET 2008. Подробно рассмотрены технологии использования различных форм меню программ, использование графических возможностей среды, вопросы применения диалоговых элементов управления и создание многооконных приложений.

Язык программирования C# является полностью объектно-ориентированным языком программирования. Поэтому во второй части учебного пособия много внимания уделяется технологиям объектно-ориентированного программирования.

Подробно рассмотрены вопросы инкапсуляции, наследования и полиморфизма, как основных принципов объектно-ориентированного программирования. Рассмотрены технологии программирования, предоставляемые использование интерфейсов платформы .NET, вопросы разработки своих интерфейсов и их использования в приложениях.

Необходимость отметить, что вопросы программирования на языке C# в консольных приложениях в некоторой степени отображении в различных учебниках и учебных пособиях. Однако, вопросы использования технологий программирования, для создания Windows приложений на языке C#, в опубликованных учебниках рассматриваются в контексте работы с базами данных или другими большими системами и для студентов первого курса обучения мало понятны. Поэтому в данном учебном пособии очень много внимания уделяется использованию технологий программирования, предоставляемых средой визуального программирования Visual Studio.NET, при решении различных математических и информационных задач, с которыми знакомятся студенты первого курса при изучении различных дисциплин.

При написании второй части учебного пособия были использованы лекции по дисциплине «Технология программирования» для специальности «Информационные системы», но материал учебного пособия можно использовать при изучении технологии программирования для любых специальностей.

ВВЕДЕНИЕ

Возможности современных языков программирования во многом определяются возможностями среды визуального программирования, в которой этот язык представлен. Язык программирования C# был специально разработан для применения в среде визуального программирования Visual Studio.NET, для эффективного использования преимуществ, предоставляемых платформой .NET. Поэтому изучение технологий программирования, предоставляемых средой визуального программирования Visual Studio.NET, позволит лучше понять особенности языка программирования C#, при разработке Windows приложений.

Структурно данное учебное пособия (часть 2) по технологии программирования рассматривает две группы вопросов – технологии программирования, предоставляемые средой Visual Studio.NET при разработке Windows-приложений и вопросы технологии программирования, предоставляемые C# как языком объектно-ориентированного программирования. Естественно, все технологии программирования языка C# в объеме данного учебного пособия рассмотреть невозможно, поэтому технологии работы с базами данных, разработки web-приложений, Интернет технологии будут изучаться студентами в дисциплинах старших куров специальности.

Изучение технологий программирования Windows-приложений начинается со знакомства с событийным принципом управления Windows. Рассмотрены основные элементов управления, предоставляемые средой. Подробно рассмотрены вопросы ввода и использования информации при решении различных задач. На большом количестве примеров, рассмотрены вопросы вывода информации, как в символьной форме, так и графической, ее преобразование и представление в виде таблиц, рисунков и графиков.

Отдельно рассмотрены вопросы использования различных форм меню программ. Подробно рассмотрены вопросы технологии создания многооконных Windows-приложений, технологии использования стандартных диалоговых окон Windows. В приведенных примерах рассматриваются вопросы перехода между окнами и редактирования табличных значений с помощью модальных окон.

Технологии объектно-ориентированного программирования рассматриваются в многочисленных примерах, использующих инкапсуляцию, наследование и полиморфизм.

Отдельно рассмотрены вопросы композиции и коллекции классов, использование стандартных коллекций, предоставляемых средой Visual Studio.NET.

Подробно рассмотрены технологии, предоставляемые интерфейсами классов, их использование для циклического просмотра коллекций объектов.

Приведены примеры по разработке и использованию делегатов классов и их применения для обработки различных событий.

1 СРЕДА ВИЗУАЛЬНОГО ПРОГРАММИРОВАНИЯ VISUAL STUDIO.NET

1.1 Введение в объектно - ориентированное программирование

В предыдущей части учебника мы уже отмечали, что данные, методы, да и сама программа, написанная на языке C#, должна размещаться в классах.

Введение в программирование новой структуры данных это не только количественное увеличение новых типов данных. Появление классов изменило технологию программирования. Если раньше в структурированном программировании основной единицей программы являлись функции и процедуры, то появление классов позволило создавать функционально законченные модули программы, которые объединяли не только данные, но и множество методов их обработки. Основной единицей таких программ стали классы (объекты), а технологии программирования с помощью классов получила название объектно-ориентированного программирования.

В этой части учебника будут рассмотрены вопросы, связанные с использованием технологий объектно-ориентированного программирования при проектировании Windows-приложений (сложных программных систем). Использование классов в технологиях объектно-ориентированного программирования (ООП) показывает, что класс может выполнять две функции – выступать в роли модуля программы или типа данных.

Модульность построения – основное свойство Windows-приложений. Разработка больших программных систем, без разделения системы на модули, потребует значительно больше времени, чем системы создаваемые коллективами разработчиков, использующих модульную технологию программирования. В ООП Windows-приложения, разрабатываются по модульному принципу, состоят из классов, являющихся основным видом модуля. Модульность построения – основное средство по ускорению процесса разработки сложных программных систем.

С другой стороны класс это тип данных. Объектно-ориентированная разработка Windows-приложений основана на стиле, называемом проектированием от данных. «Проектирование системы сводится к поиску абстракций данных, подходящих для конкретной задачи. Каждая из таких абстракций реализуется в виде класса, который и становится модулем – архитектурной единицей построения программной системы.»[5]

В большинстве разрабатываемых Windows-приложений классы выполняют обе функции, так что каждый модуль программной системы имеет вполне определенную смысловую нагрузку. Язык C# допускает как

классы, являющиеся типами данных, так и классы, играющие единственную роль модуля. К классам модулям относятся, например, такие классы, как Console, Math. Классы, играющие единственную роль модуля, объектов создавать не могут. Точнее, существует единственный объект этого класса, представляющий модуль. Поля и методы этого модуля обычно доступны методам других классов.

При разработке больших программных систем определяющим является среда разработки. Технологии программирования, предоставляемые средой программирования, значительно сокращают время разработки больших программных систем.

Поэтому изучение технологии программирования ООП мы начинаем со знакомства со средой программирования Visual Studio.NET при создании приложений для Windows (Windows Forms Application).

1.2 Понятие о событийном управлении Windows

Если рассматривать работу программ, написанных в консольном приложении, то следует отметить, что после их запуска начинают выполняться операторы метода Main().

Особенность поведения программ, написанных для Windows, является то, что программы после их запуска переходят в бесконечный цикл ожидания сообщений поступающих от Windows.

В общем случае сообщения – это реакция операционной системы Windows на происходящие в системе события. При этом под событием следует понимать появление любой «нестандартной» ситуации в работе компьютера, например, нажатии клавиши на клавиатуре, перемещении курсора мыши, деления на ноль и т.д.

Все события, на которые может реагировать система Windows, пронумерованы и каждому номеру – «вектору прерывания», поставлена в соответствие специальная программа (драйвер) корректно реагирующая на соответствующее событие. Например, драйверы периферийных устройств компьютера (клавиатуры, мыши или таймер).

При появлении события система Windows определяет «номер» события и запускает соответствующий драйвер. Драйвер «обрабатывает» событие и создает сообщение, которое пересылается системе Windows.

В основе работы операционной системы Windows лежит принцип событийного управления. Это означает, что и сама система и все приложения, написанные для Windows, после запуска ожидают действий пользователя или сообщений операционной системы и реагируют на них определенным образом.

Сообщение Windows является записью, которая содержит информацию о том, что произошло и дополнительную информацию (параметры) о произошедшем событии. Например, структура некоторого сообщения может включать дескриптор окна программы, код

(идентификатор) сообщения, уточняющие параметры (например, координаты x и y курсора мыши) и время создания сообщения.

Все сообщения, получаемые Windows, помещаются в системную очередь сообщений, которая существует в единственном варианте. Далее из системной очереди сообщения распределяются в очереди сообщений отдельных Windows-приложений. При этом для каждого приложения создается своя очередь сообщений. Очередь сообщения приложений может пополняться не только из системной очереди. Любое приложение может послать сообщение любому другому сообщению, в том числе и само себе.

Каждое Windows-приложений имеет непрерывный цикл обработки сообщений поступающих от Windows. С помощью этого цикла приложение извлекает «свои» сообщения и передает их соответствующим обработчикам сообщений окна приложения. Каждое окно приложения имеет свой цикл обработки сообщений, а также свою функцию окна, которой передаются сообщения, извлеченные из очереди приложения.

Обычно Windows-приложение имеет главное окно, в котором располагаются основные элементы управления – меню, кнопки, полосы прокрутки, флажки и т. д. Работая с приложением, пользователь выбирает строки меню, нажимает кнопки или использует другие элементы управления.

Каждый элемент управления (кнопка или строка меню) имеет свой идентификатор. Когда Вы нажимаете на кнопку или выбираете строку меню, в очередь сообщений приложения Windows заносит сообщение, содержащее идентификатор использованного элемента управления. Таким образом операционная система Windows направляет сообщение от использованного элемента управления в очередь того приложения, к которому принадлежит данный элемент управления.

В предложенной структуре есть очевидная часть работы программиста – написать обработчики сообщений на некоторые события, например, клик мышкой по кнопке окна вашего приложения.

В приложениях, создаваемых для Windows, (File -> New -> Project -> Windows Forms Application), всегда используется два основных типа (классы пространства имен) – Form и Application.

Класс Application управляет поведением приложения – запускает метод Main(), в котором находится цикл обработки сообщений (Application.Run();), выполняет необходимые действия при выборке сообщений и корректно завершает работу приложения (файл Program.cs).

Класс Form определяет пользовательский интерфейс приложения – он инициализирует окно формы и готовит приложение к работе (файл Form1.cs).

Более подробно работу этих классов мы будем изучать по мере необходимости при изучении материала учебника.

Рассмотрим последовательность действий при создании простого приложения для Windows**.**

1.3 Основные окна среды Visual Studio.Net

Условно процесс создания приложения для Windows включает два основных этапа.

Первый этап – этап визуального программирования предназначен для проектирования пользовательского интерфейса, т.е. задания всех необходимых элементов управления. На этом этапе очень важны дизайнерские умения программиста.

На втором этапе необходимо разработать коды обработчиков сообщений приложения, т.е. определить поведение программы при появлении того или иного сообщения от Windows.

Чтобы выполнить первый этап разработки приложения для Windows необходимо открыть среду разработки Visual Studio .Net (File -> New -> Project -> Windows Forms Application) – смотри рисунок 1.1. При выборе проекта вам необходимо указать название папки на рабочем столе, в которой будут размещаться создаваемые средой файлы.

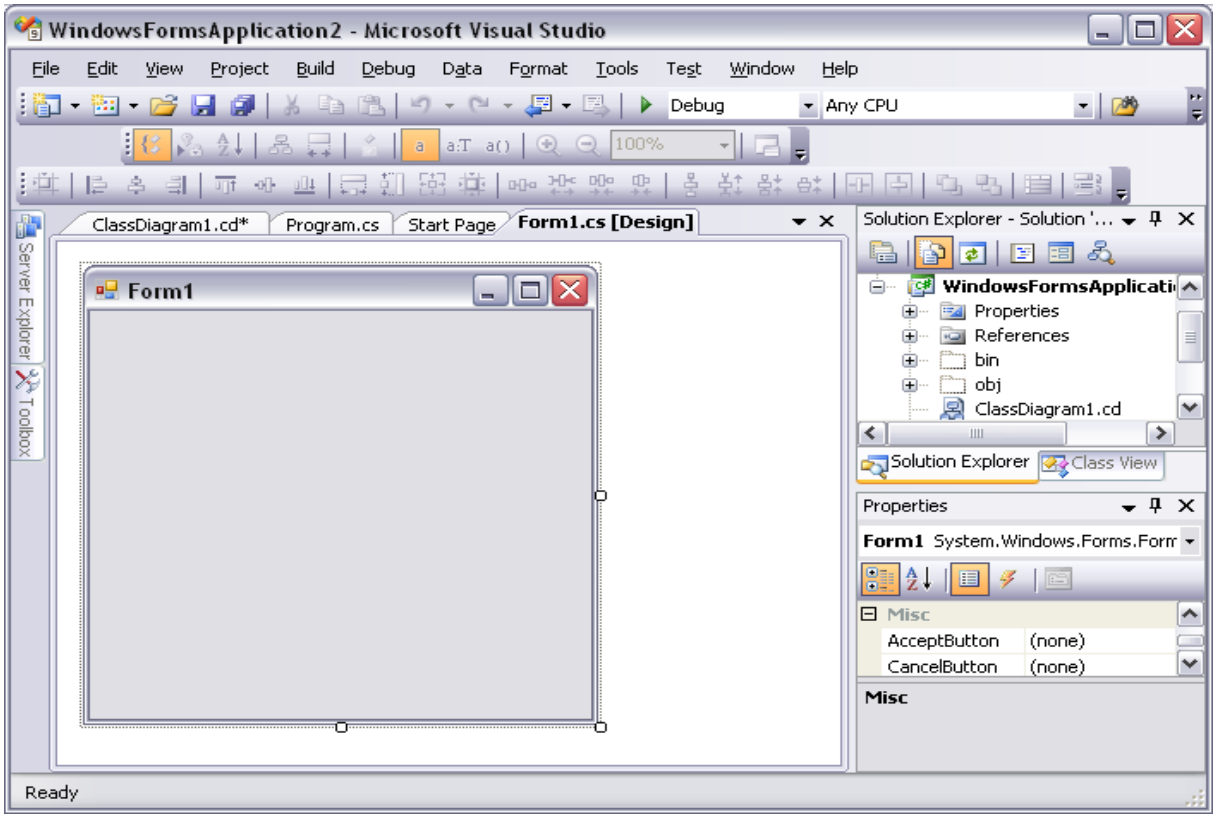


Рисунок 1.1 – Среда разработки Visual Studio .Net

В центре рисунка 1.1 расположено окно Form 1, которое относится к пространству имен System.Windows.Forms. Пока окно этой формы пустое, но разрабатываемый проект уже можно запустить на выполнение. Для этого необходимо нажать клавишу F5 или выбрать режим работы среды Debug и в нем команду Start. Окно запущенного приложения смотрите на рисунке 1.2.

Пока закроем наш проект и рассмотрим подробнее назначение отдельных окон среды разработки Visual Studio .Net.

Окно Form1 имеет 4 страницы, которые используются в различных режимах редактирования проекта, например, Form1.cs[Design] используется для размещения элементов управления на форме, а окно Program.cs для редактирования кода программы.

Окно Properties – свойства элементов управления (в нашем примере это форма) или файлов, предварительно установленных в окне Server Explorer. Страницы этого окна сгруппированы либо по категориям, либо в алфавитном порядке, либо по свойствам и событиям.

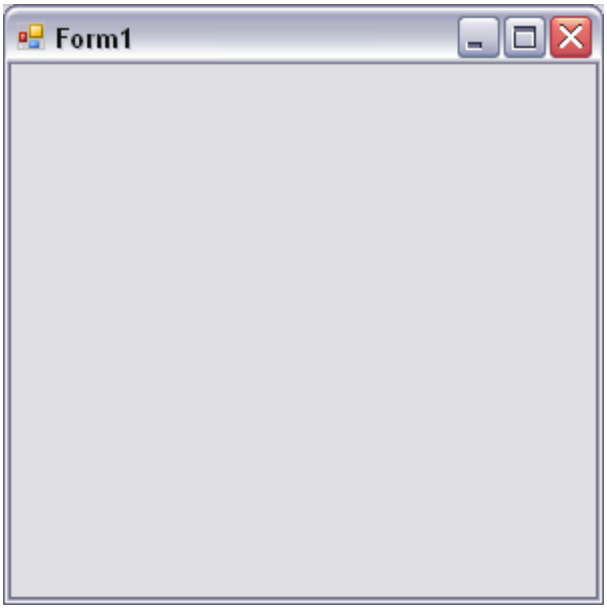


Рисунок 1.2 – Рабочее окно программы

Окно Server Explorer (на рисунке 1.1 слева в свернутом состоянии) служит для доступа к источникам данных на вашем компьютере и на удаленном сервере (информация из журнала событий и др. служб системы).

Окно Toolbox (инструменты – на рисунке 1.1 слева в свернутом состоянии) содержит различные элементы управления, которые мы будем использовать для размещения на форме.

Окно Solution Explorer – на рисунке 1.1 справа, позволяет просматривать и редактировать файлы проекта. Просмотр может осуществляться по файлам или по классам.

Обычно после запуска программы появляется окно ошибок, в котором перечислены номера строк кода программы, в которых допущены ошибки.

Подробное описание работы со средой разработки Visual Studio .Net приведено в книге А.В. Фролов, Г.В. Фролов Визуальное проектирование приложений C#[3].

1.4 Основные структурные элементы разработки проекта C#

Язык C# является объектно-ориентированным языком программирования, в котором основным понятием является понятие класса.

Класс это тип – шаблон описание множества объектов. Объект это переменная типа класс, она динамически создается в ходе выполнения программы, реально существует в памяти компьютера и обычно удаляется из памяти компьютера по завершении выполнения программы.

При разработке проекта, обычно создается несколько классов, но в ходе работы такого проекта могут динамически появляться сотни объектов, взаимодействующих друг с другом достаточно сложным образом.

Базовый класс в библиотеке FCL, являющийся прародителем всех классов как библиотечных, так и создаваемых разработчиком, является класс Object.

Пространство имен – это объединение некоторого множества классов общей тематикой или группой разработчиков. Собственные имена классов внутри пространства имен должны быть уникальны. В разных пространствах могут существовать классы с одинаковыми именами. Полное или уточненное имя класса состоит из уникального имени пространства имен, символа точка и собственного имени класса. В пространстве имен могут находиться как классы, так и пространства имен.

Пространства имен систематизирует структура библиотеки FCL, которая содержит большое число различных пространств имен, объединяющих классы определенной тематики. Если рассматривать все пространства имен как некоторое иерархическое дерево классов и пространств имен, то пространство System, а в нем класс Object будут являться корнем такого дерева.

Проект – это с одной стороны приложение на стадии разработки, а с другой стороны единица компиляции. Результатом компиляции проекта является сборка. Каждый проект содержит одно или несколько пространств имен. На начальном этапе создания проекта по заданному типу проекта автоматически строится каркас приложения, состоящий из классов, которые являются наследниками классов, входящих в состав библиотеки FCL. Так, если разработчик указывает, что он хочет построить проект типа "Windows Forms Application", то в состав каркаса приложения по умолчанию войдет класс Form1 – наследник библиотечного класса Form.

В проект входят все файлы с классами, построенные автоматически в момент создания проекта, и файлы с классами, созданные разработчиком проекта. Помимо этого, проект содержит ссылки на пространства имен из библиотеки FCL, которые включают классы, используемые в ходе работы программы. Проект содержит ссылки на все подключаемые к проекту DLL

и другие проекты. «В проект входят установки и ресурсы, требуемые для работы. Частью проекта является файл, содержащий описание сборки.

В зависимости от выбранного типа проект может быть выполняемым или невыполняемым. К выполняемым проектам относятся, например, проекты типа Console или Windows**.** При построении каркаса выполняемого проекта в него включается класс, содержащий статический метод с именем Main. В результате компиляции такого проекта создается PE-файл (Portable Executable file) – выполняемый переносимый файл с уточнением exe. Напомним, что PE-файл может выполняться только на компьютерах, где установлен Framework .Net, поскольку это файл с управляемым кодом. [5]

К невыполняемым проектам относятся, например, проекты типа Dll**.**

Сборка – результат компиляции проекта. Сборка представляет собой коллекцию из одного или нескольких файлов, помеченных номером версии. Каждая сборка разворачивается на компьютере как единое целое. Программист работает с проектами, CLR работает со сборками. Сборка позволяет решать вопросы безопасности, так как содержит описание требуемых ей ресурсов и права доступа к элементам сборки. Каждая сборка содержит манифест, включающий полное описание сборки, ее элементов, требуемые ресурсы, ссылки на другие сборки, исполняемые файлы. Благодаря этому описанию CLR не требуется никакой дополнительной информации для развертывания сборки, трансляции промежуточного кода и его выполнения. Манифест идентифицирует сборку, специфицирует файлы, требуемые для реализации сборки, специфицирует типы и ресурсы, составляющие сборку, задает зависимости, необходимые в период компиляции для связи с другими сборками, специфицирует множество разрешений, необходимых, чтобы сборка могла выполняться на данном компьютере.

Каждый проект, создаваемый в Visual Studio.NET 2008, помещается в некоторую оболочку, называемую Решением – Solution. Решение может содержать несколько проектов, как правило, связанных общей темой. Например, в одно Решение можно поместить три проекта: DLL с разработанными классами, определяющими содержательную сторону приложения, консольный проект решения задачи и проект под управлением Windows.

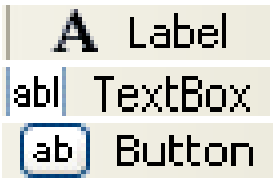
Когда создается новый проект, он может быть помещен в уже существующее Решение или может быть создано новое Решение, содержащее проект. [5]

1.5 Пример первой учебной программы

Рассмотреть пример использования формы для создания приложения, работающего в системе Windows**.** В качестве первого примера выбрана известная задача вычисления периметра треугольника.

Задача 1.1 В режиме диалога необходимо задать стороны треугольника и вычислить его периметр. После ввода значений сторон треугольника выполнять следующие проверки: все стороны треугольника должны быть больше нуля и сумма любых двух сторон больше третьей. Работу программы сопровождать необходимыми комментариями.

На этапе визуального программирования мы будем использовать три стандартных элемента управления из окна Toolbox: статический текст или метка (Labe1), поле ввода или вывода текста – окно редактирования (TextBox**)** и командную кнопку (Button)**:**



– статический текст;

– поле ввода или вывода текста;

– командная кнопка.

Метки нужны для размещения поясняющих надписей – четыре метки.

Три поля ввода, одно поле вывода результата и одна кнопка управления.

Весь процесс визуального программирования заключался в перетаскивании необходимых элементов управления из окна Toolbox на форму, и размещении их в определённом порядке. Если это требуется, то с помощью мыши можно отрегулировать размеры и расположение элементов управления, добавленных на форму, а также размеры самой формы.

Обычно окно Toolbox находится в «свернутом» состоянии. Чтобы его «развернуть» необходимо левой клавишей мыши кликнуть на панель Toolbox, раскрыв ее, и закрепить в определённом месте экрана с помощью элемента  (кликнуть на него). По окончании работы с окном Toolbox его можно «свернуть» кликнув на элементе  .

В процессе визуального программирования необходимо изменять некоторые свойства элементов управления, например, у меток и кнопки было изменено свойство Text в соответствии с рисунком 1.3. Для этого необходимо пользоваться окном Properties, которое изображено на рисунке 1.4.

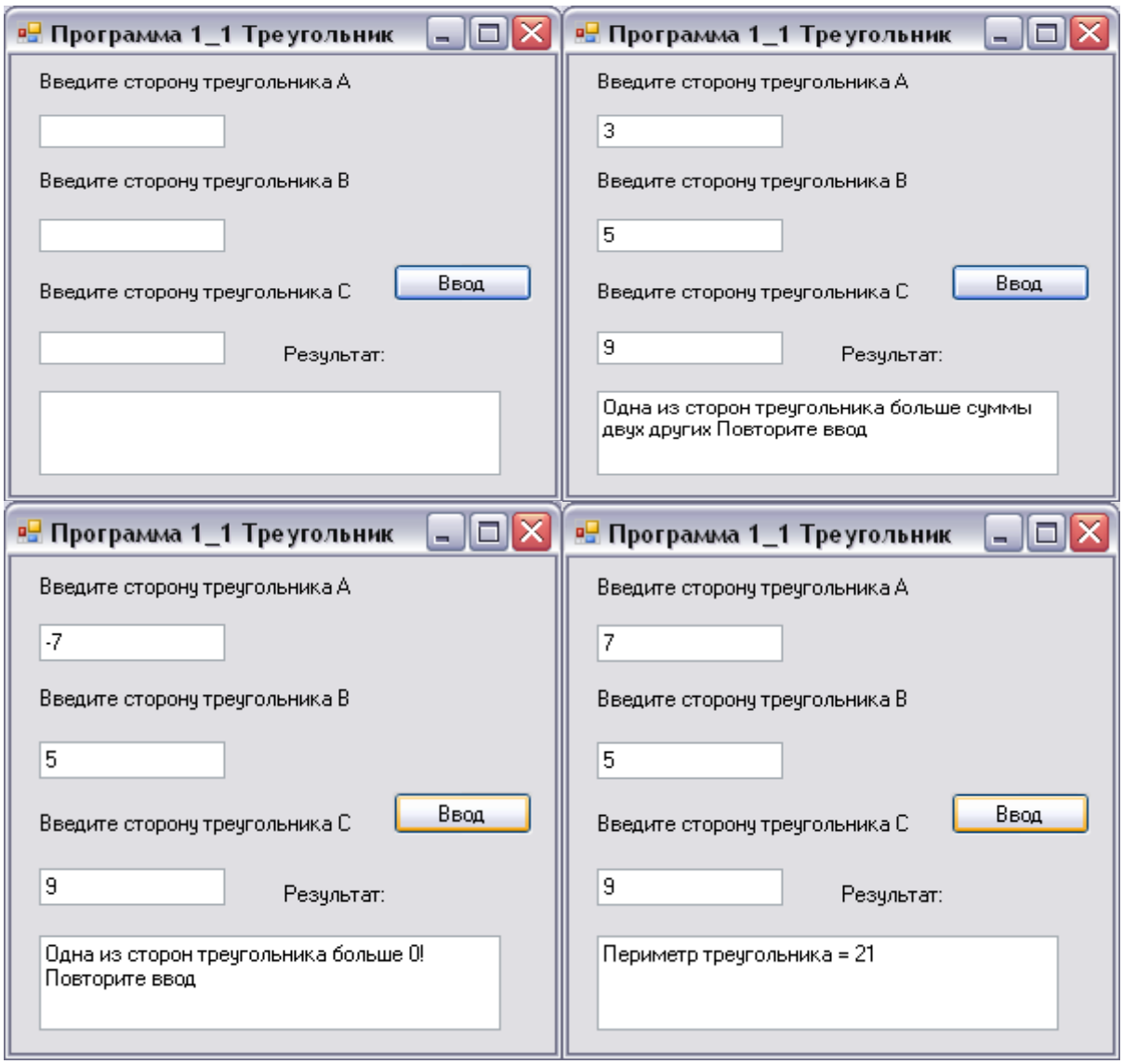


Рисунок 1.3 Окна программы «Треугольник»

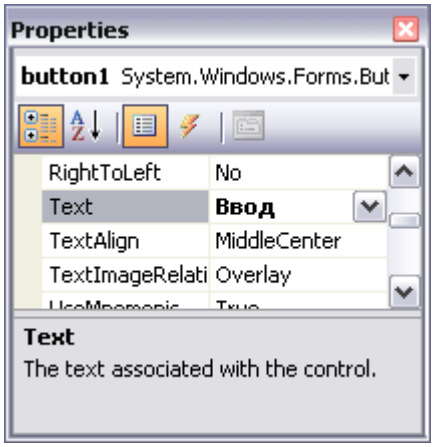


Рисунок 1.4 – Окно Properties для элемента button1

Отличие поля результата от полей ввода в том, что в нем установлено свойство `Multiline = true`.

Во всех элементах управления использовалось поле `Text`.

Для получения «пустого» метода – обработчика сообщения о нажатии на кнопку «Ввод» достаточно на этапе визуального программирования дважды кликнуть на эту кнопку. В пустую заготовку обработчика сообщения

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

включим код ввода в режиме диалога значений сторон треугольника и необходимые проверки на их соответствие треугольнику.

Исходный код файла Program.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 static class Program
  {
   /// <summary>
   /// The main entry point for the application.
   /// </summary>
   [STAThread]
   static void Main()
   {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
   }
  }
}
```

Исходный код файла Form1.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
   InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e)
  {
   int a,b,c,p;
```

```
   a = Convert.ToInt32(textBox1.Text);
   b = Convert.ToInt32(textBox2.Text);
   c = Convert.ToInt32(textBox3.Text);
   p = a + b + c;
   if (a > 0 && b > 0 && c > 0)
    if (a + b > c && a + c > b && b + c > a)
     textBox4.Text = "Периметр треугольника = " +
p.ToString();
    else
    {
     textBox4.Text = "Одна из сторон треугольника больше суммы
двух других Повторите ввод ";
    }
   else
   {
    textBox4.Text = "Одна из сторон треугольника больше 0!
Повторите ввод ";
   }
  }
 }
}
```

Среда автоматически формирует богатый список пространств имен. Рассмотрим некоторые из них.

Пространство имен System содержит определение фундаментальных и базовых классов, определяющих типы данных, события, обработчики событий и другие, необходимые в каждом приложении компоненты.

В пространстве имен System.Collections определены классы, реализующие функциональность таких контейнеров, как массивы, списки, словари, хэши и т.п.

Классы пространства System.ComponentModel используются для реализации необходимого поведения компонентов и элементов управления приложения на этапе его разработки и выполнения.

Класс System.Data необходим приложениям, работающим с базами данных посредством интерфейса ADO.NET. Этот интерфейс вы будете рассматривать при изучении баз данных.

Пространство имен System**.**Drawing необходимо для доступа к интерфейсу графических устройств (Graphics Device Interface, GDI), а точнее, к его расширенной версии GDI+. Классы, определенные в этом пространстве имен, необходимы для рисования в окнах приложений текста, линий, двухмерных фигур, изображений и других графических объектов.

Пространство System.Linq содержит классы, задающие типы, интерфейсы, стандартные операторы запроса.

Пространство имен System.Windows.Forms — в нем определены классы, реализующие поведение оконных форм, составляющих базу оконных приложений Microsoft windows на платформе Microsoft .NET Frameworks.

Реально программе нужны два пространства имен – System и System.Windows.Forms, все остальные пространства имен формируются «на вырост».

После отладки программы все файлы необходимо записать (в меню выбрать действие File->Save All).

Визуальная среда программирования даже для небольшой программы создает более 10 файлов и вложенных папок, пояснение которых уводит читателя в такие дебри работы среды, что ни один из известных авторов книг по программированию на языке C# не отважился дать подробных пояснений, какие файлы и зачем создаются в папках разрабатываемого проекта.

Мы не будем отходить от традиций, отметив только что в папке 1_1_treygolnik, созданной на рабочем столе для первой программы, находится папка WindowsFormsAplication1, в которой еще одна папка с именем WindowsFormsAplication1 и файл проекта программы, вызываемый для редактирования WindowsFormsAplication1.csproj. В очередной папке WindowsFormsAplication1 (как в матрешке) находятся еще три папки bin, obj, Properties и несколько файлов, в том числе файлы с кодом программы – Program.cs и кодом формы – Form1.cs. Здесь же находится ресурсный файл формы Form1, в котором сохраняется внешний вид формы, и файл Form1.Designer.cs, в котором запоминаются значения «свойств» формы и всех элементов управления, размещенных на форме.

Реально для работы со средой разработки Visual Studio .Net нам пока нужен файл кода формы Form1.cs.

Рекомендую, на первых этапах освоения программирования, не «вламываться» в свойства и названия остальных файлов, а ограничится работой с текстом только файла Form1.cs.

1.6 Вопросы для самопроверки

1 Понятие события в работе компьютера?

2 Как система Windows «различает» события?

3 Что делает система Windows, получив информацию о появлении события?

4 Понятие драйвера.

5 Понятие сообщения.

6 Что содержит сообщение?

7 Куда поступают сообщения, получаемые системой Windows от драйверов?

8 Зачем в каждом приложении имеется цикл обработки сообщений поступающих от Windows?

9 Какой метод приложения реализует цикл обработки сообщений поступающих от Windows?

10 Что определяет класс Form?

2 ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

2.1 Технология визуального проектирования форм

Среда визуального программирования C# .NET включает в себя Windows.Forms.Designer (конструктор или дизайнер форм Windows) – инструмент, позволяющий в интерактивном режиме выполнять визуальное проектирование формы, размещая на ней необходимые элементы управления. Преимущества Windows.Forms.Designer в том, что Вы можете размещать элементы управления на форме в соответствии с Вашим представлением красоты и при этом не думать о конкретных значениях многих свойств этих элементов, например, о свойствах (точнее числовых значениях этих свойств), определяющих их местоположение или размер. Значения большинства свойств задаются автоматически конструктором формы. Однако конструктор формы может помочь Вам до определенного момента, а потом Вам придется писать код программы вручную, попутно разбираясь в том, что для Вас сгенерировал Windows.Forms.Designer.

В Windows приложении (в отличие от консольного приложения) конструктором формы автоматически создаётся несколько классов с расширением .cs, например, класс с именем Form1.cs и класс с именем Program.cs.

Классы в C# синтаксически не являются неделимыми и могут состоять из нескольких частей, каждая из которых начинается с ключевого слова “partal”(частичный). Возможность разбиения описания одного класса на части облегчает работу над большим классом. Каждая часть класса хранится в отдельном файле со своим именем. Например, для примера предыдущей главы, автоматически были созданы два файла Form1 с расширением .cs – Form1.cs и Form1.Designer.cs.

Первая часть класса Form1, хранящаяся в файле "Form1.cs", предназначена для разработчика – именно в ней располагаются автоматически создаваемые обработчики событий, происходящих с элементами управления, код которых создается самим разработчиком. Такая технология программирования, основанная на работе с формами, называется визуальной, событийно управляемой технологией программирования.

Например, фрагмент файла Form1.cs, рассмотренного на предыдущей лекции:

```
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
  }
        private void button1_Click(object sender, EventArgs e)
```

```
        {
            int a,b,c,p;
            a = Convert.ToInt32(textBox1.Text);
            b = Convert.ToInt32(textBox2.Text);
            c = Convert.ToInt32(textBox3.Text);
      p = a + b + c;
 . . .
```

Вторая часть класса Form1 находится в файле с именем "Form1.Designer.cs". Эта часть класса заполняется автоматически конструктором формы. Когда мы занимаемся визуальным проектированием формы и размещаем на ней различные элементы управления, меняем их свойства, придаем форме нужный вид, задаем обработчики событий для элементов управления, то конструктор формы транслирует наши действия в действия над объектами соответствующих классов, создает соответствующий код и вставляет его в нужное место класса Form1.

Предполагается (надеемся), что Вы не должны вмешиваться в работу конструктора формы и корректировать эту часть кода класса Form1. Тем не менее, иметь представление о его работе или даже понимать код, созданный конструктором формы иногда очень полезно.

Ниже приведен фрагмент файла Form1.Designer.cs, рассмотренного на предыдущей лекции:

```
namespace WindowsFormsApplication1
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components =
null;
          . . .
private void InitializeComponent()
        {
            this.label1 = new System.Windows.Forms.Label();
            this.label2 = new System.Windows.Forms.Label();
            this.label3 = new System.Windows.Forms.Label();
            this.label4 = new System.Windows.Forms.Label();
      this.button1 = new System.Windows.Forms.Button();
          . . .
     // label1
            //
            this.label1.AutoSize = true;
            this.label1.Location = new
System.Drawing.Point(12, 9);
            this.label1.Name = "label1";
          this.label1.Size = new System.Drawing.Size(174, 13);
         this.label1.TabIndex = 0;
         this.label1.Text = "Введите сторону треугольника A";
```

```
. . . и т.д. всего на 3 страницах.
```

Класс Program.cs , автоматически создаваемый для нашего проекта, содержит статический метод Main(). При запуске нашей программы система Windows ищет метод Main() и начинает выполнять указания, стоящие в нем. Часто метод Main() называют точкой входа в программу.

Ниже приведен файл Program.cs, примера, рассмотренного на предыдущей лекции:

```
namespace WindowsFormsApplication1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
         Application.EnableVisualStyles();
         Application.SetCompatibleTextRenderingDefault(false);
         Application.Run(new Form1());
        }
    }
}
```

В отличие от консольных приложений, где тело метода Main() изначально было пустым и должно было заполняться разработчиком проекта, в Windows приложениях метод Main() уже заполнен необходимыми указаниями и, как правило, разработчиком не изменяется. Что же делает автоматически созданный метод Main()? Он работает с классом Application библиотеки FCL, вызывая поочередно три статических метода этого класса.

Метод Application.EnableVisualStyles(); представляет компоненты создаваемого приложения в стиле Windows XP.

Метод Application.SetCompatibleTextRenderingDefault(false); говорят, что его назначение понятно из текста.

Метод Application.Run(new Form1()); основной метод класса и обычно он единственный в Main().

Основную работу выполняет метод Run – в процессе его вызова создается объект класса Form1 и открывается форма - визуальный образ объекта, с которой может работать пользователь проекта. После завершения инициализации всех объектов элементов, размещенных на форме, запускается цикл обработки сообщений Windows, и приложение ожидает сообщения от Windows. Пользователю остается вводить данные в поля формы и нажимать на кнопки управления.

Для реализации этапа визуального программирования необходимо кратко рассмотреть назначение элементов управления, расположенных на панели Toolbox.

2.2 Элементы управления панели Toolbox

Элементы управления, или компоненты, помещают на форму из элементов управления ToolBox (View ▸ ToolBox). В этом разделе лекции кратко описаны простейшие элементы управления панели ToolBox.

Изучение элементов управления начнем с элемента, который практически всегда присутствует на форме – текст комментариев.

2.2.1 Label – метка. Метка предназначена для размещения текста на форме. Размещаемый текст, хранится в свойстве Text. Можно задавать шрифт текста (свойство Font), цвет фона (свойство BackColor), цвет шрифта текста (ForeColor) и выравнивание (свойство TextAlign) текста метки. Метка может автоматически изменять размер в зависимости от длины текста (свойство AutoSize = True). Можно разместить на метке изображение (свойство Image) и задать прозрачность (установить для свойства BackColor значение Color.Transparent). В этом случае будут видны компоненты, расположенные на форме за надписью.

Метка, как самостоятельный элемент управления, не может получать фокус ввода – «запоминать» положение курсора мышки и создавать обработчики событий на нажатие клавиш мышки, клавиатуры или других элементов управления.

2.2.2 Button – кнопка. Элемент управления Button может получать фокус ввода, при этом основное событие, обрабатываемое кнопкой — щелчок мышью (Click). Кроме того, кнопка может реагировать на множество других событий — нажатие клавиш на клавиатуре и мыши, изменение параметров и т. д.

Если занести имя кнопки в свойство Accept Button формы, на которой расположена кнопка, то нажатие клавиши Enter вызывает событие Click, даже если кнопка не имеет фокуса ввода. Такая кнопка имеет дополнительную рамку и называется кнопкой по умолчанию.

Аналогично, если занести имя кнопки в свойство Cancel Button формы, на которой расположена кнопка, то нажатие клавиши Esc вызывает событие Click для этой кнопки.

Можно изменить начертание и размер шрифта текста кнопки, который хранится в свойстве Text, задать цвет фона и фоновое изображение так же, как и для метки.

Кнопка может содержать помимо надписи еще и изображение (свойство Image или ImageList вместе с ImageIndex).

Кнопки часто используются в диалоговых окнах. Как видно из названия, такое окно предназначено для диалога с пользователем и запрашивает у него какие-либо сведения (например, какой выбрать режим работы или какой файл открыть). Диалоговое окно обладает свойством модальности. Это означает, что дальнейшие действия с приложением невозможны до того момента, пока это окно не будет закрыто. Закрыть окно можно, либо подтвердив введенную в него информацию щелчком на

кнопке OK (или Yes), либо отменив ее с помощью кнопки закрытия окна или, например, кнопки Cancel. Для сохранения информации о том, как было закрыто окно, у кнопки определяют свойство DialogResult. Это свойство может принимать стандартные значения из перечисления DialogResult, определенного в пространстве имен System.Windows.Forms. Значения перечисления приведены в таблице 2.1.

Таблица 2.1 - Значения перечисления DialogResult

Значение	Описание	Значение	Описание
None	Окно не закрывается	Ignore	Нажата кнопка Ignore
OK	Нажата кнопка OK	Yes	Нажата кнопка Yes
Cancel	Нажата кнопка Cancel	No	Нажата кнопка No
Abort	Нажата кнопка Abort	Retry	Нажата кнопка Retry

2.2.3 Поле ввода TextBox. Компонент TextBox позволяет пользователю вводить и редактировать текст, который запоминается в свойстве Text. Можно вводить строки практически неограниченной длины (приблизительно до 32 000 символов), корректировать их, а также вводить защищенный текст (пароль) путем установки маски, отображаемой вместо вводимых символов (свойство PasswordChar). В однострочном режиме высота компонента автоматически меняется так, чтобы показывать только одну строку.

Свойство Text используется для ввода единственной строки, а свойство Lines— для ввода нескольких. Строки в этом свойстве хранятся в виде массива, что позволяет организовать индексный доступ к ним.

Для обеспечения возможности ввода и вывода нескольких строк устанавливают свойства Multiline, ScrollBars и WordWrap. Доступ только для чтения устанавливается с помощью свойства ReadOnly.

Элемент содержит методы очистки (Clear), выделения (Select), копирования в буфер (Copy), вставки из него (Paste) и другие. Может обрабатывать множество событий, основными из которых являются KeyPress и KeyDown.

2.2.4 ListBox— список. Компонент ListBox представляет собой список с возможностью выбора одного или нескольких пунктов. Свойство SelectMode может иметь одно из следующих значений: None — выбор пунктов запрещен; One — можно выбирать только один пункт; MultiSimple — можно выбирать несколько пунктов; MultiExtended — можно выбирать несколько пунктов с учетом нажатых клавиш Shift и Ctrl: если нажата и удерживается клавиша Shift, выбирается непрерывный диапазон пунктов; если нажата и удерживается клавиша Ctrl, выбирается произвольный (необязательно непрерывный) диапазон пунктов. Если в свойство MultiColumn помещено значение True, пункты списка могут располагаться в несколько колонок, при этом свойство ColumnWidth определяет ширину

колонок. Если колонки выйдут за ширину компонента, автоматически вставляется горизонтальная полоса прокрутки.

2.2.5 Переключатель RadioButton. Переключатель позволяет пользователю выбрать один из нескольких предложенных вариантов, поэтому переключатели обычно объединяют в группы. Если один из них устанавливается (свойство Checked), остальные автоматически сбрасываются. Программист может менять стиль и цвет текста, связанного с переключателем, и его выравнивание. Для переключателя можно задать цвет фона и фоновое изображение так же, как и для метки. Переключатели можно поместить непосредственно на форму, в этом случае все они составят одну группу. Если на форме требуется отобразить несколько групп переключателей, их размещают внутри компонента Group или Panel. Свойство Appearance управляет отображением переключателя: либо в традиционном виде (Normal), либо в виде кнопки (Button), которая «залипает» при щелчке на ней мышью.

2.3 Пример использования элементов управления

Для этого раздела выбран чисто учебный пример для вычисления тригонометрических функций синус, косинус и тангенс. Значение переменной (угол в радианах) задается в режиме диалога с программой. Также в режиме диалога задается имя вычисляемой функции и количество разрядов формата вывода функции на экран монитора – точность вычисления. Для реализации этой задачи в проекте использованы следующие элементы управления: Label, Button, Panel, RadioButton, ListBox и TextBox.

Код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
   InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e)
  {
   //Значения по умолчанию
   string ff = "F3";
```

```
    string fu = "sin";
    Double x=0;
    //Ввод значения угла
    x = Convert.ToDouble(textBox2.Text);
    //Выбор функции
    if (listBox1.SelectedIndex == 1) fu = "cos";
    if (listBox1.SelectedIndex == 2) fu = "tn";
    // точность вычислений
    if (radioButton1.Checked)
    {
     ff = "F3";
    }
     else
     if (radioButton2.Checked)
     {
      ff = "F4";
     } else
       if (radioButton3.Checked)
       {
        ff = "F5";
       };
    switch (fu)
    {
case "sin": textBox1.Text = " sin= " +
                            Math.Sin(x).ToString(ff); break;
case "cos": textBox1.Text = " cos= " +
                            Math.Cos(x).ToString(ff); break;
case "tn": textBox1.Text = " tn= " +
          (Math.Sin(x) / Math.Cos(x)).ToString(ff); break;
    }
   }
  }
}
```

Работа программы представлена на рисунке 2.1.

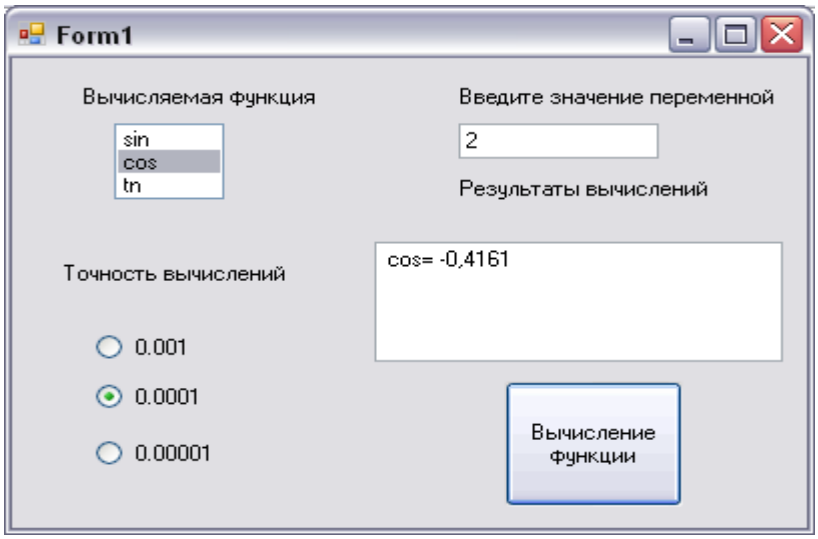


Рисунок 2.1 – Работа программы вычисления функции

Работа программы очевидна и не нуждается в дополнительных комментариях.

Другие элементы управления – меню, диалоговые окна, рисунки и т.д. будут рассмотрены в следующих лекциях дисциплины.

2.4 Вопросы для самопроверки

1 Для чего предназначен Windows.Forms.Designer платформы .NET?

2 Что означает служебное слово “partal” в описании класса формы?

3 Какая часть описания класса формы содержит обработчики сообщений?

4 С какого метода начинается выполнение Windows приложении?

5 Какой метод создает объект класса Form1 при запуске программы?

6 Для чего предназначен управляющий элемент Label?

7 Какое свойство управляющего элемента Label позволяет выводить информацию в окно формы?

8 Каким свойством управляющего элемента Label можно задать его «прозрачность»?

9 С помощью какого свойства можно нанести изображение на кнопку в окне формы?

10 Как называется диалоговое окно, блокирующее дальнейшие действия с приложением до того момента, пока это окно не будет закрыто?

3 ГРАФИЧЕСКИЙ ИНТЕРФЕЙС ЯЗЫКА C#

3.1 Пространство имен System.Drawing

Графический интерфейс приложений C# (GDI+), как и других приложений, предназначенных для работы в рамках Microsoft .NET Framework, состоит из набора классов, объединяемых пространством имен. Одно из основных пространств имен GDI+ языка C# является пространство имен System.Drawing. Классы этого пространства имен определяют перечень объектов и инструментов, предназначенных для «рисования».

К наиболее часто используемым классам пространства имен System.Drawing относятся: Brush (Brushes, SolidBrush и др.). Объекты Brush (кисть) используются для заполнения пространства внутри геометрических фигур. Тип Brush — это абстрактный базовый класс, остальные типы являются производными от Brush и определяют разные наборы возможностей.

Pen (Pens, SystemPens). Pen (перо) — это объект класса, при помощи которого можно рисовать прямые и кривые линии. В классе Pen определен набор статических свойств, при помощи которых можно получить объект Pen с заданными свойствами (например, с установленным цветом).

Font (FontFamily). Объекты типа Font определяют характеристики шрифта (имя, размер, начертание и т. п.). FontFamily представляет набор шрифтов, которые относятся к одному семейству, но имеют некоторые небольшие отличия.

Graphics. Этот класс определяет набор свойств и методов для вывода текста, изображений и геометрических фигур на экран монитора. Он позволяет приложению работать с контекстом устройств системы Windows.

Region. Этот класс определяет область, занятую геометрической фигурой.

Point (PointF). Эти структуры обеспечивают работу с координатами точки. Point работает со значениями типа int, а PointF — со значениями типа float.

В пространстве имен System.Drawing также находятся классы Icon, Image, Color, Bitmap и другие классы так или иначе связанные с отображением графической информации на экране монитора.

3.2 Класс Graphics

Основным классом для «рисования» в языке C# является класс Graphics. Он предназначен для вывода графической информации в клиентскую часть формы приложения. Для того чтобы приложение могло что-нибудь нарисовать в окне, оно должно получить или создать для этого

окна объект класса Graphics. Далее, пользуясь свойствами и методами этого объекта, приложение может рисовать в окне различные фигуры или текстовые строки.

Прежде чем создавать в приложении объект класса Graphics необходимо определиться с обработчиком события по «рисованию». В системе Windows за перемещением и изменением размера окон «следит» специальное сообщение WM_PAINT, которое при необходимости извещает приложения, о том, что им следует перерисовать содержимое окна. Любые действия с окном – перемещение его по рабочему столу экрана монитора, изменение его размеров и т.д. сопровождается требованием системы Windows «перерисовать» окно. В приложении обработчик события WM_PAINT, получив такое сообщение, должен выполнить перерисовку всего окна или его части, в зависимости от дополнительных данных, полученных вместе с сообщением WM_PAINT. Для создания заготовки обработчика сообщения WM_PAINT формы необходимо в окне свойств окна формы дважды кликнуть мышкой на пункт PAINT (смотри рисунок 3.1).

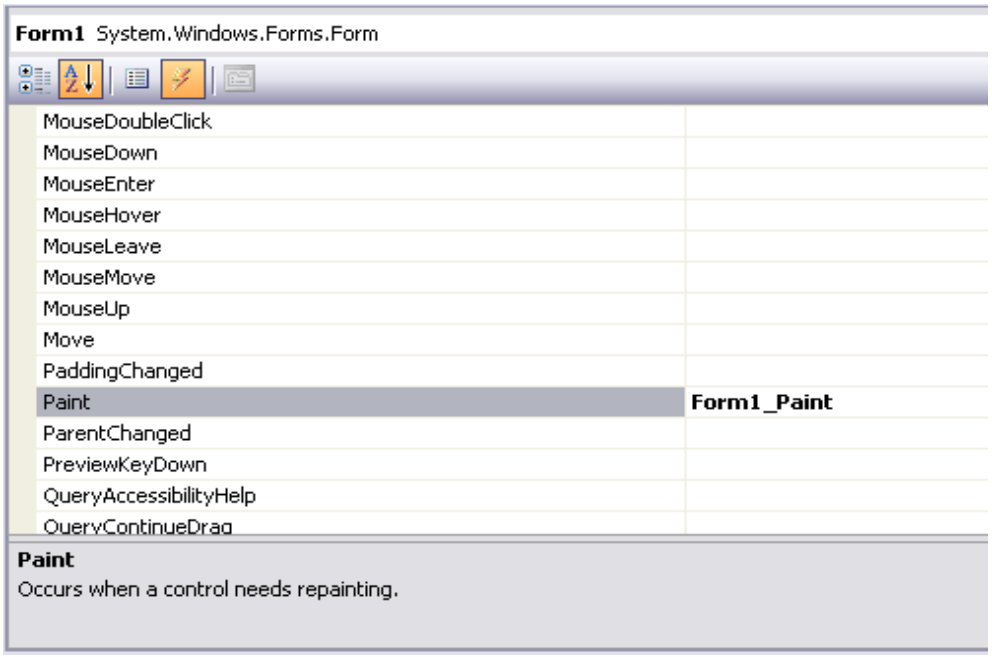


Рисунок 3.1 – Создание обработчика WM_PAINT

В результате будет создан обработчик события WM_PAINT (точнее Form1_Paint – как это видно на рисунке 3.1). Этот обработчик будет получать управление всякий раз, когда по тем или иным причинам возникнет необходимость в перерисовке содержимого окна нашего приложения.

Вот в каком виде будет создан обработчик события Paint:

```
private void Form1_Paint(object sender, PaintEventArgs e)
  {
      }
```

Обработчику Form1_Paint передаются два параметра.

Через первый параметр передается ссылка на объект, вызвавший событие. В нашем случае это будет ссылка на форму Form1 (где рисовать).

Что же касается второго параметра, то через него передается ссылка на объект класса PaintEventArgs. Этот объект имеет свойство ClipRectangle, доступное только для чтения. Через свойство ClipRectangle передаются границы области, которую должен перерисовать обработчик события Paint. Эти границы передаются в виде объекта класса Rectangle. Свойства этого класса Left, Right, Width и Height, наряду с другими свойствами, позволяют определить расположение и размеры области. По умолчанию обработчик события Paint игнорирует свойство ClipRectangle, перерисовывая содержимое окна полностью. Например,

```
private void Form1_Paint(object sender, PaintEventArgs e)
  {  Pen myPen = new Pen(Color.Red, 2);
     Graphics g = e.Graphics;
     g.DrawEllipse(myPen, 100, 100, 100, 100);
  }
```

В данном примере создается «перо» для рисования красным цветом и толщиной 2 пикселя – объект myPen. Создается объект g типа Graphics для перерисовываемой области (по умолчанию вся форма). Обратите внимание нет new. Далее для объекта g запускается метод рисования эллипса.

Особенность объекта g типа Graphics заключается в том, что фактически объект это указатель на контекст устройства монитора (специальные программы системы Windows, связывающие приложение с драйвером видеокарты компьютера). С помощью контекстов устройств система Windows обеспечивает совместимость приложений и драйверов устройств компьютера, например, независимо от типа видеокарты код нашего приложения останется неизменным, а все проблемы управления видеокартой решает контекст устройства монитора.

Работа нашей программы приведена на рисунке 3.2

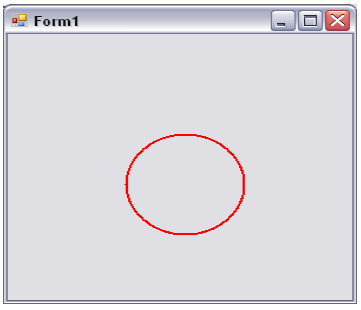


Рисунок 3.2 – Отображение эллипса

Как и большинство классов C# класс Graphics имеет свойства и методы. Рассмотрим некоторые из них.

Clear() – этот метод заполняет объект Graphics выбранным пользователем цветом, удаляя его предыдущее содержимое.

Существует большая группа методов для «рисования» некоторых геометрических фигур: DrawArc(), DrawBezler(), DrawBeziers(), DrawCurve(), DrawEllipse(), DrawIcon(), DrawLine(), DrawLines(), DrawPie(), DrawPath(), DrawRectange(), DrawRectangles(), DrawString().

Для «заполнения» внутренних областей геометрических фигур используются методы, перед которыми находится слово Fill, например, FillPie(), FillEllipse() или FillRectangle().

Работу некоторых методов можно изучить в книге «Визуальное проектирование приложений C#» авторов А.В. Фролов, Г.В. Фролов [3] (глава 10). Необходимо отметить, что получение объекта типа Graphics, для перерисовываемых областей возможно не только с помощью объекта класса PaintEventArgs обработчика событий Form1_Paint. Можно использовать метод `CreateGraphics`, описанного в классах формы и элемента управления, например,

```
Graphics g = this.CreateGraphics(); или
Graphics g = Graphics.FromHwnd(this.Handle);
```

Можно создать графический объект с помощью объекта-потомка `Image`. Этот способ используется для изменения существующего изображения, например,

```
Bitmap bm = new Bitmap( "d:\\picture.bmp" );
Graphics g = Graphics. FromImage( bm );
```

3.3 Пример программной реализации

В качестве учебной программы рассмотрим вывод на экран монитора структуры типа граф и некоторых стандартных графических фигур.

Код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
   InitializeComponent();
  }
```

```
 public static int p = 0;
 private void Form1_Paint(object sender, PaintEventArgs e)
 {
  Graphics g = e.Graphics;
  if (p == 0)
  {
   String st;
   int i, j;
   int xc, yc;
   //g.Clear(Color.White);
   int[,] a = new int[9, 9]
   {{0, 6, 1000, 4, 1000, 1000, 1000, 1000, 1000},
          {6, 0,    5,1000,3,6,1000,1000,1000},
          {1000,5,0,5,1000,3,1000,1000,1000},
          {4,1000,5,0,1000,1000,4,1000,1000},
          {1000,3,1000,1000,0,7,1000,7,1000},
          {1000,6,3,1000,7,0,6,4,1000},
          {1000,1000,1000,4,1000,6,0,9,6},
          {1000,1000,1000,1000,7,4,9,0,8},
          {1000,1000,1000,1000,1000,1000,6,8,0}};
   int[,] V = new int[9, 2]
             {{200, 40},
             {100, 80},
             {200, 100},
             {280, 60},
             { 60, 140},
             {200, 160},
             {340, 140},
             {160, 220},
             {360, 260}};
   e.Graphics.Clear(Color.Bisque);
   Pen myPen = new Pen(Color.Red, 2);
   for (i = 0; i < 9; i++)
    for (j = 0; j < 9; j++)
     if ((a[i, j] != 0) && (a[i, j] != 1000))
     {
      g.DrawLine(myPen, V[i, 0], V[i, 1], V[j, 0], V[j, 1]);
      xc = (int)(V[i, 0] + V[j, 0]) / 2;
      yc = (int)(V[i, 1] + V[j, 1]) / 2;
      st = Convert.ToString(a[i, j]);
      g.DrawString(st, new Font("Times new Roman",8),
Brushes.Blue, xc - 6, yc - 15);
     }
   for (i = 0; i < 9; i++)
   {
    g.FillEllipse(Brushes.White, V[i, 0] - 12, V[i, 1] - 12,
25, 25);
    st = Convert.ToString(i);
    g.DrawString(st, new Font("10_IC_1", 12), Brushes.Black,
V[i, 0] - 6, V[i, 1] - 6);
   }
  }
  if (p == 1)
```

```
  {
   g.Clear(Color.White);
   g.DrawRectangle(new Pen(Brushes.Blue, 2), 5, 5, 380, 270);
   Pen myPen = new Pen(Color.Black, 2);
   Point[] myPoints =
   {
    new Point(10, 10),
    new Point(100, 40),
    new Point(150, 24),
    new Point(100, 100),
   };
   g.DrawPolygon(myPen, myPoints);
   g.DrawLine(myPen, 10, 250, 360, 250);
   g.DrawLine(myPen, 10, 250, 10, 150);
   g.DrawString("Y", new Font("10_IC_1", 12), Brushes.Black,
5,130);
   g.DrawString("X", new Font("10_BT_1", 12), Brushes.Black,
360,240);
   g.FillRectangle(Brushes.Blue, 25, 180, 25, 70);
   g.FillRectangle(Brushes.Red, 75, 150, 25, 100);
   g.FillRectangle(Brushes.Green, 125, 200, 25, 50);
   g.FillPie(Brushes.Blue, 170, 10, 200, 150, 0, 120);
   g.FillPie(Brushes.Red, 170, 10, 200, 150, 120, 120);
   g.FillPie(Brushes.Green, 170, 10, 200, 150, 240, 120);
  }
 }
 private void button2_Click(object sender, EventArgs e)
    {
        if (p == 0) p = 1; else p = 0;
        this.Invalidate();
    }
 }
}
```

Работа программы представлена на рисунках 3.3 и 3.4.

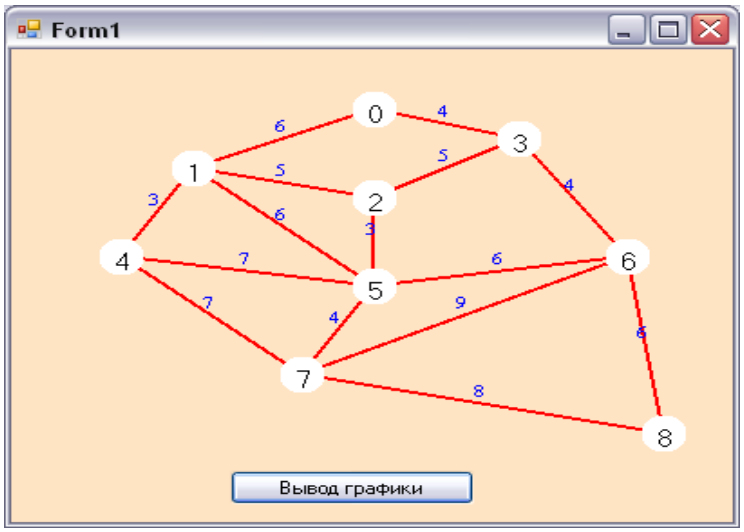


Рисунок 3.3 – Вывод графа

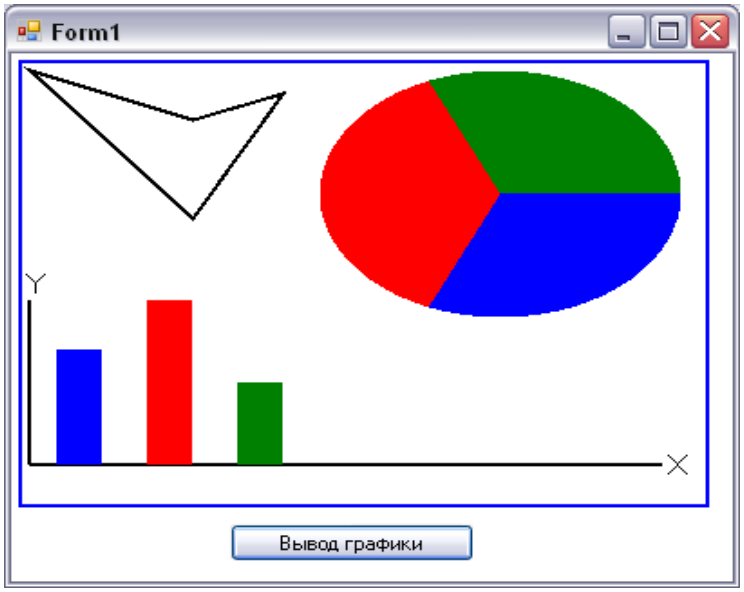


Рисунок 3.4 – Вывод различных графических фигур

Базовой литературой этого примера является книга «Визуальное проектирование приложений C#» авторов А.В. Фролов, Г.В. Фролов [3].

В программе управление выводом информации осуществляется с помощью глобальной переменной p. Если p = 0 (при запуске программы), то в окно формы выводится рисунок графа. При нажатии кнопки «Вывод графики» в окне формы значение p меняется на противоположное – если p было равно 0, то оно станет равным 1 и наоборот.

В обработчике события нажатия кнопки присутствует метод, применяемый для текущего объекта – this.Invalidate(); (указатель this всегда содержит адрес текущего объекта – объекта, с которым работает программа). В нашем примере текущим объектом является форма. Метод this.Invalidate(); требует от операционной системы Windows сформировать для формы сообщение WM_PAINT. Получив это сообщение, наша программа запустит обработчик события private void Form1_Paint(object sender, PaintEventArgs e), т.е. перерисует содержимое окна формы при новых значениях переменной p. Таким образом, можно из приложения «формировать» сообщение WM_PAINT операционной системы Windows.

Рассмотрим подробнее код учебной программы. Первая часть содержит рисунок графа из предыдущего семестра – заимствована матрица смежности. Дополнительно введен массив координат вершин графа – V. Предварительно граф был нарисован на листе бумаги и значения координат (в масштабе) взяты непосредственно с рисунка. Очищаем окно формы `e.Graphics.Clear(Color.Bisque);` - закрашиваем его цветом `Color.Bisque`. Устанавливаем значение цвет «пера» красный и толщину линий 2 пикселя – `Pen myPen = new Pen(Color.Red, 2);`. Запускаем циклы рисования линий между вершинами графа. Одновременно вычисляются координаты xc и yc для средней точки между вершинами графа. Используя эти координаты, мы над линией ребра

выводим его значение, взятое из матрицы смежности. При выводе текста по формату записи `g.DrawString` необходимо указывать тип шрифта и его размер. В следующем цикле рисуются и подписываются вершины графа. Метод `g.FillEllipse` рисует «закрашенный» эллипс, вписанный в прямоугольную область, расположение и размеры которой передаются ему в качестве параметров.

На «втором» окне формы выполнена «очиска» окна белым цветом. В учебных целях «нарисованы» многоугольник, линии осей координат X и Y, набор прямоугольников (гистограмма) и круговая диаграмма.

Метод `g.DrawPolygon(myPen, myPoints);` позволяет нарисовать многоугольник, координат вершин которого передаются массивом. Первый параметр метода определяет цвет линии, которой рисуется многоугольник.

Метод `g.FillRectangle` позволяет рисовать закрашенные прямоугольники, заданные координатой верхнего левого угла, а также шириной и высотой. В качестве первого параметра этим методам передается цвет «заполнения» внутренней области прямоугольника.

Круговая диаграмма рисуется с помощью метода `g.FillPie` предназначенного для отображения «закрашенных» сегментов. В качестве первого параметра методу нужно передать цвет «заполнения» внутренней области сегмента. Следующие четыре параметра задают расположение и размеры прямоугольника, в который вписывается эллипс. Последние два параметра определяют углы, ограничивающие сегмент эллипса – начальный угол и его приращение по часовой стрелке.

3.4 Вопросы для самопроверки

1 Что означает GDI+ в языке C#?

2 Что определяет объект класса Brush?

3 Что определяет объект класса Pen?

4 Что определяет класс Graphics?

5 Какое сообщение в системе Windows «следит» за перемещением и изменением размера окон?

6 Как называется обработчик, связанный с перерисовкой содержимого окна формы нашего приложения?

7 Что определяет первый формальный параметр обработчика Form1_Paint?

8 Что определяет второй формальный параметр обработчика Form1_Paint?

9 Понятие контекста устройства монитора.

10 Какой метод требует от операционной системы Windows сформировать для формы сообщение WM_PAINT?

4 ИСПОЛЬЗОВАНИЕ МЕНЮ В ПРИЛОЖЕНИИ

4.1 Меню программы

Меню программы должно соответствовать основным режимам работы программы, поэтому к выбору пунктов меню и команд отдельных пунктов необходимо относится с особой тщательностью. Для лучшего понимания технологии использования меню в программах рассмотрим последовательность действий при решении следующей учебной задачи.

Задача 4.1 Разработать приложение, позволяющее создавать матрицу 6*6 случайных целых чисел в диапазоне от 10 до 100. Напечатать эту матрицу с помощью richTextBox. Записывать клиентскую область приложения (включая распечатку матрицы) в текстовый файл. Считывать текстовый файл в клиентскую область – печатать содержимое текстового файла. Из содержимого клиентской области выделять матрицу. Все действия оформить с использованием меню.

Можно выделить три режима работы программы: Матрица (создание и печать), Файл (запись и чтение) и Текст (очистка и преобразование).

Создадим обычное приложение для работы в системе Windows.

Для добавления меню в главное окно нашего приложения необходимо переместить из окна элементов управления Toolbox значок меню с названием MenuStrip. Как только Вы это сделаете, окно форм примет вид, показанный на рисунке 4.1

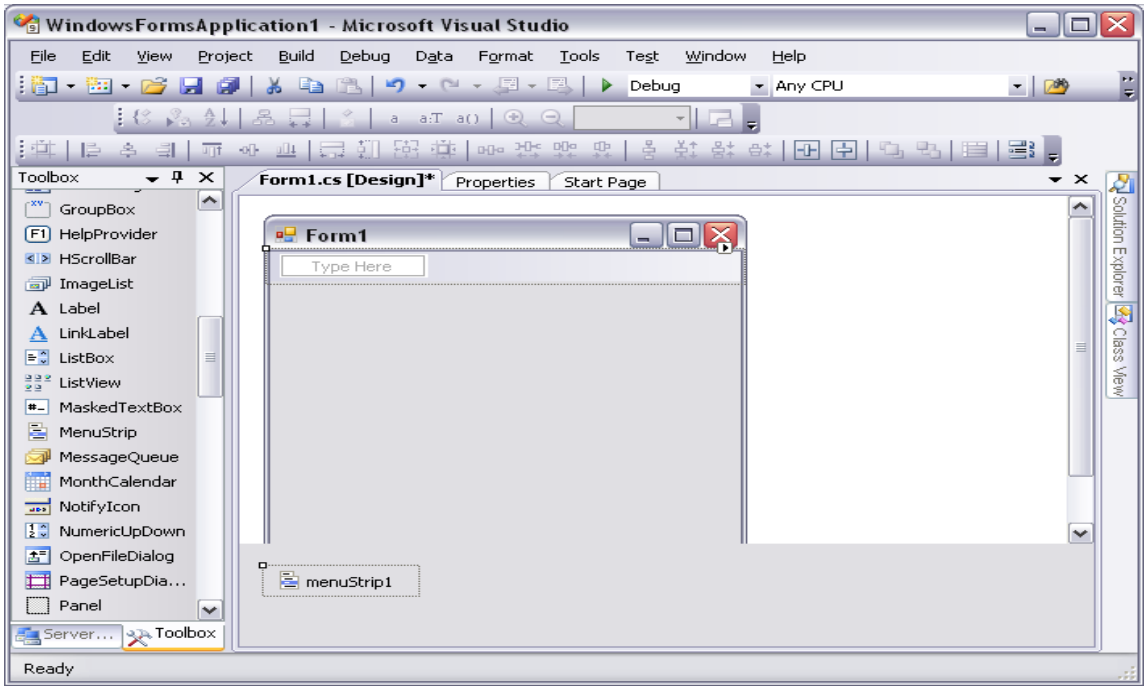


Рисунок 4.1 – Добавление меню

В нижней части этого окна появится значок управляющего элемента — меню menuStrip1. Непосредственно под заголовком окна появится пустое пока меню, представленное полем с надписью Type Here (что можно перевести как «печатать здесь»).

Напечатайте в этом поле строку «&Matric». В результате этого в окне нашего приложения появится меню Matric (рисунок 4.2).

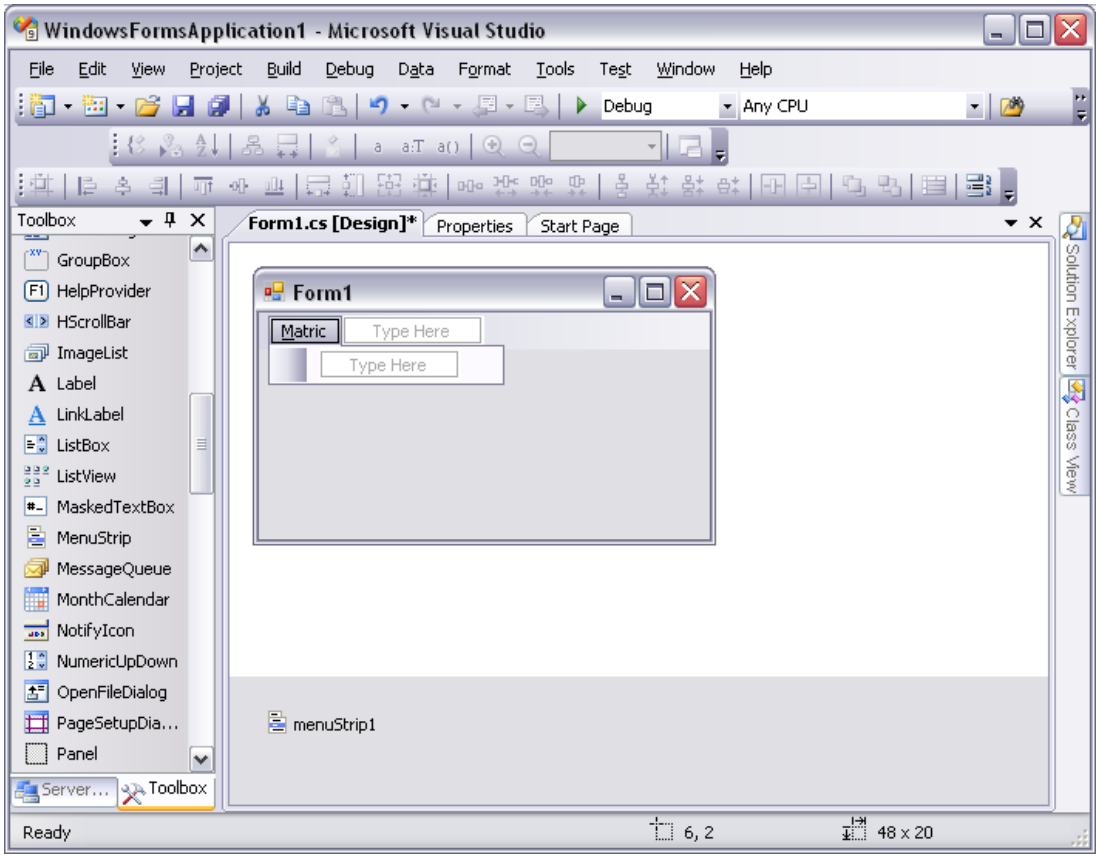


Рисунок 4.2 – Создание меню для работы с матрицей

Ввод символа & позволяет создать клавиатурный акселератор для ввода команды с помощью клавиатуры (Alt+P). Применение акселераторов является альтернативным мышке способом выбора команд. В комбинацию с клавишей Alt включается буква, стоящая за символом амперсанд. В тексте меню программы буквы, включенные в акселератор, отображаются подчеркнутыми.

После ввода режима работы программы поле ввода Type Here опустилось вниз, предлагая вводить команды заданного режима. Ведем две команды &SozM и &PrintM. После этого переходим в правое поле ввода Type Here для ввода режима работы с файлом. Последним введем меню режима работы с текстом клиентской области приложения.

Если при вводе Вы допустили ошибку, то ее можно исправить. Для этого нужно кликнуть правой кнопкой мышки в нужной строке меню и в появившемся на экране контекстном меню выбрать необходимый режим редактирования.

С помощью строки Insert New Вы можете вставить новую строку меню между уже существующих строк. Строка Insert Separator предназначена для вставки разделительной линии между строками меню. При помощи строки Edit Names можно отредактировать идентификаторы строк и меню.

Если Вам нужно изменить введенные названия строк и меню, это можно сделать по месту, выбрав нужную строку мышью.

После этого Вы можете оттранслировать приложение и запустить его, нажав кнопку F5. Убедитесь, что меню отображается, и Вы можете выбирать его строки. Если все нормально, можно переходить к следующему этапу создания нашего приложения.

Работу с текстом в нашем приложении мы будем выполнять с помощью элемента RichTextBox, позволяющего работать с файлами разного типа. Для этого перетащим из окна Toolbox на нашу форму элемент RichTextBox.

Настройте свойства компонента, чтобы он занимал все окно приложения. Далее отредактируйте свойство Dock. Это свойство задает расположение выбранного компонента внутри содержащей его формы – необходимо щелкнуть в центре (см. рисунок 4.3).

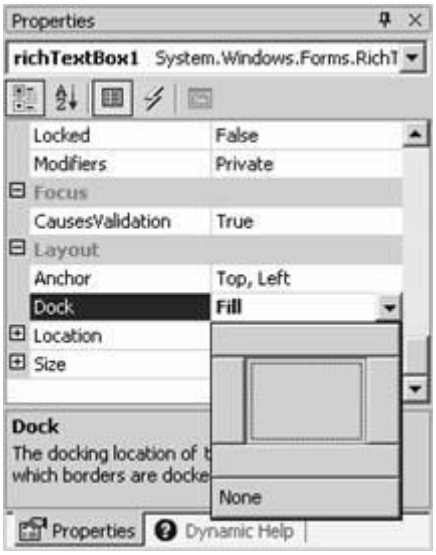


Рисунок 4.3 – Редактирование свойства Dock

Если запустить полученное приложение на выполнение, то в результате на экране должно появиться окно, показанное на рисунке 4.4.

Рисунок 4.4 – Рабочее окно программы

Обработчики событий от команд меню создаются таким же образом, что и обработчики событий от кнопок. Чтобы создать обработчик события для команды меню, ее нужно щелкнуть дважды левой клавишей мыши.

Создадим обработчики событий для всех команд, но не режимов работы программы. Все обработчики событий создаются с пустым телом, например:

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

Во многих приложениях основные команды меню программы дублируются «иконками», на инструментальных панелях. Создадим инструментальную панель для команд нашего приложения.

4.2 Создание инструментальной панели приложения

Чтобы добавить инструментальную панель в окно нашего приложения, перетащим мышью ее значок ToolStrip из окна элементов управления среды Visual Studio .NET в окно проектирования формы нашего приложения.

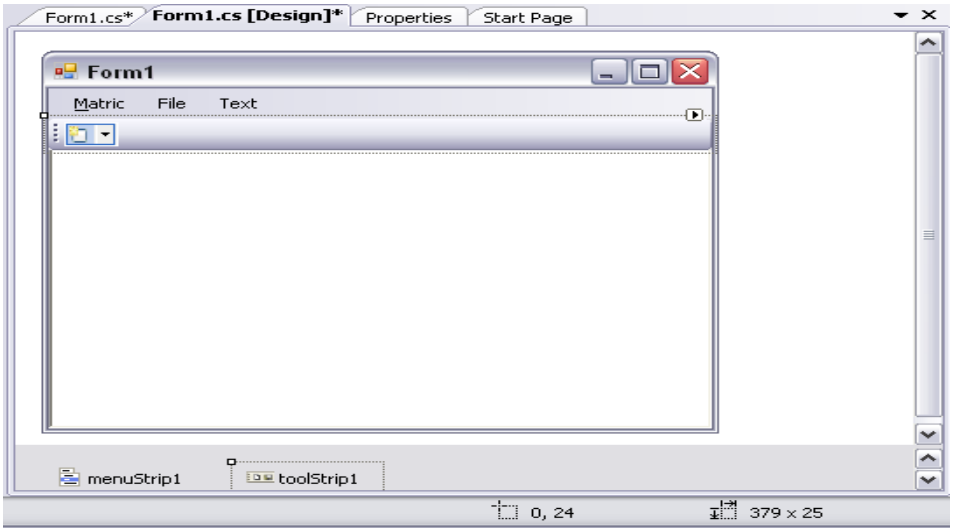


Рисунок 4.5 – Первый этап создания инструментальной панели приложения

По умолчанию окно инструментальной панели появится в верхней части формы. В только что добавленной панели размещена одна кнопка. На рисунке 4.5 показан первый этап создания инструментальной панели.

На рисунке 4.5 видно, что окно создаваемой инструментальной панели находится внутри окна редактора текста (в верхней его части). Чтобы исправить это положение, щелкните правой кнопкой мыши окно редактора текста, а затем выберите из контекстного меню строку Bring to Front (но не Properties). В результате окна примут правильное взаимное расположение.

Если навести курсор мыши на кнопку элемента ToolStrip в инструментальной панели, то появляется подсказка – Add ToolStripButton. Кликнув левой кнопкой мышки по этой кнопке элемента ToolStrip, мы получим справа в инструментальной панели новую аналогичную кнопку. При этом на всех кнопках изображен один и тот же рисунок.

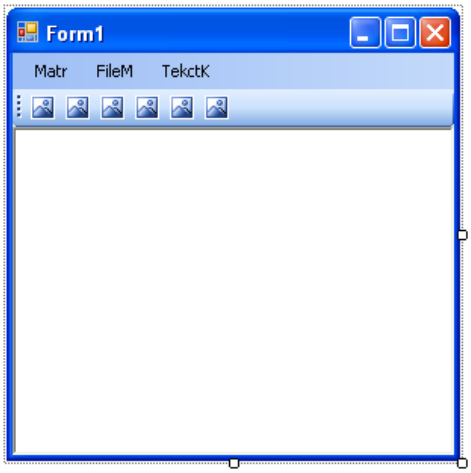


Рисунок 4.6 – Этап создания кнопок инструментальной панели

Поэтому на следующем этапе необходимо изменить изображения на кнопках в инструментальной панели. Прежде чем приступать к замене изображений на кнопках в инструментальной панели, Вам нужно подготовить эти изображения при помощи любого графического редактора или найти необходимые изображения и поместить их в отдельную папку своего проекта (в нашем случае папка с именем ImaList). Можно запустить поиск на диске С по шаблону *.bmp и среди найденных изображений выбрать наиболее подходящие по смыслу.

Далее, начиная с первой кнопки, в ее свойствах выбираем команду Imaje и открываем окно Select Resource, а в нем команду Import и в открывшемся диалоговом окне Open перемещаемся в папку ImaList и выбираем нужные изображения.

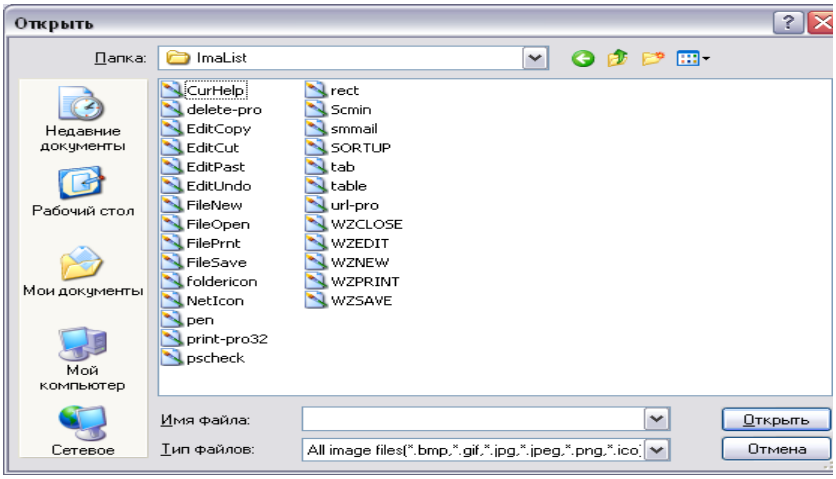


Рисунок 4.7 – Выбор файла изображения для кнопки инструментальной панели

Редактирование свойства ToolTipText каждой кнопки инструментальной панели, позволит нам снабдить каждую кнопку окном с поясняющим сообщением, которое появляется при наведении на кнопку курсора мыши.

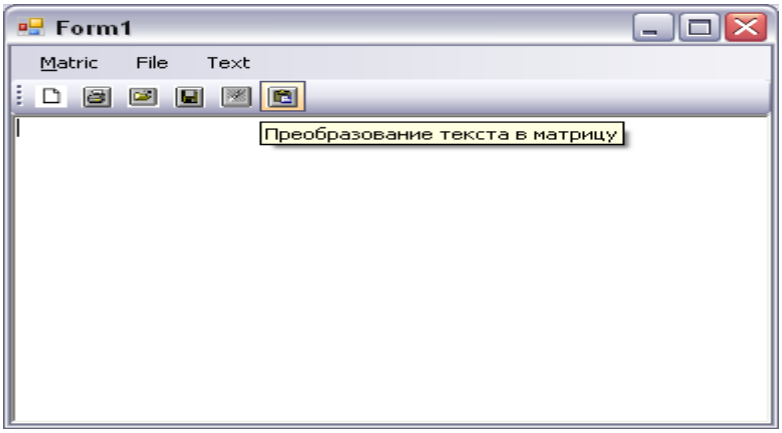


Рисунок 4.8 – Рабочее окно приложения при наведении курсора мышки на кнопку ObrabT инструментальной

Обработчики событий от кнопок инструментальной панели необходимо связать с обработчиками событий от команд меню. Для этого в свойствах каждой кнопки запускаем страницу «события» и выбираем событие Click (смотри рисунок 4.9).

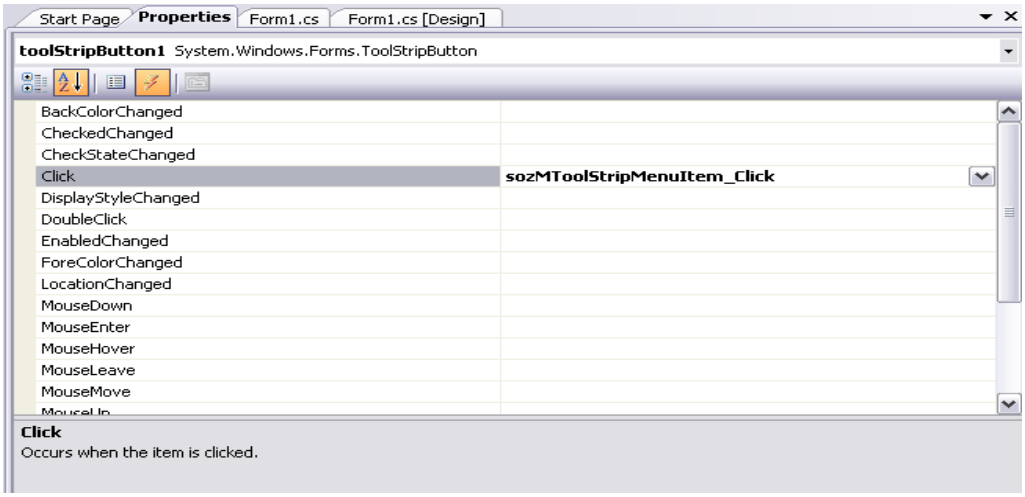


Рисунок 4.9 – Выбор обработчика событий каждой «иконки»

Дважды щелкаем по правой части таблицы события Click, при этом откроется диалоговое окно уже созданных обработчиков событий для команд нашего меню (смотри рисунок 4.10). Выбираем нужный по смыслу обработчик событий – в нашем примере это первая кнопка инструментальной панели и ей необходимо поставить в соответствие событие sozMToolStripMenuItem_Click.

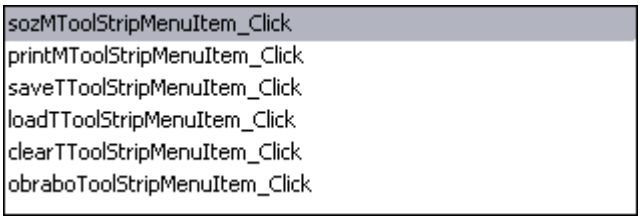


Рисунок 4.10 – Перечень обработчиков событий команд меню

4.3.Вопросы для самопроверки

1 Для чего создается меню приложения?

2 Чему должны соответствовать команды меню приложения?

3 Какой элемент управления окна ToolBox позволяет создавать меню приложения?

4 Какое поле редактора меню приложения используется для ввода команды?

5 С помощью какой команды редактора меню приложения можно вставить новую строку меню приложения между уже существующих строк?

6 Зачем при создании меню приложения желательно использовать клавиатурные акселераторы?

7 Для чего предназначена команда Insert Separator редактора меню приложения?

8 В чем принципиальное отличие элемента управления RichTextBox от TextBox?

9 Какой элемент управления используется для хранения изображений «иконок», дублирующих основные команды меню приложения?

10 Какое свойство кнопки инструментальной панели, позволит снабдить каждую кнопку окном с поясняющим сообщением, которое появляется при «наведении» на кнопку курсора мыши?

5 ИСПОЛЬЗОВАНИЕ ДИАЛОГОВЫХ МЕНЮ

5.1 Обработчики событий для работы с матрицей

Продолжая разработку приложения с меню и инструментальной панелью, нам необходимо написать код обработчиков сообщений для команд создания матрицы 6×6 и вывод (печать) матрицы в клиентскую область нашего приложения.

Создание матрицы необходимо заканчивать выводом на экран сообщения об успешном окончании работы обработчика, например, «Матрица создана».

Результатом метода «Печать матрицы» является сама матрица, поэтому дополнительных сообщений не требуется.

Исходный код программы будет рассматриваться фрагментами по мере наполнения обработчиков событий.

Начальный исходный код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public static int[,] a = new int[6, 6];
  public Form1()
  {
   InitializeComponent();
  }
  private void sozMToolStripMenuItem_Click(object sender,
     EventArgs e)
  {
   Random rnd = new Random();
   for (int i = 0; i < 6; i++)
    for (int j = 0; j < 6; j++)
     a[i, j] = rnd.Next() % 90 + 10;
   richTextBox1.AppendText("Матрица создана \n");
  }
  private void printMToolStripMenuItem_Click(object sender,
EventArgs e)
  {
   string ss;
   richTextBox1.AppendText("\n");
   for (int i = 0; i < 6; i++)
   {
```

```
    ss = "";
    for (int j = 0; j < 6; j++)
     ss = ss + Convert.ToString(a[i, j]) + "\t";
    richTextBox1.AppendText(ss + "\n");
   }
  }
```

. . .

Код остальных обработчиков событий пока «пустой».

5.2 Обработчик событий для открытия файла

Следующей по очереди на инструментальной панели находится кнопка открытия файла – команда LoadT. Рассмотрим порядок действий при написании этого обработчика события.

Для того чтобы реализовать в нашем приложении обработчик открытия файла, нам потребуется элемент OpenFileDialog. Перетащим значок этого элемента из окна Toolbox в окно нашей формы. Значок элемента OpenFileDialog1 появится внизу на панели под нашей формой (смотри рисунок 5.1).

Рисунок 5.1 – Подключение элемента OpenFileDialog

Использование элемента OpenFileDialog (особенно в части диалога) это целая технология программирования, позволяющая просто обращаться к дискам, папкам и файлам нашего компьютера. Как и любой элемент окна Toolbox он представлен классом, имеющим множество методов для работы с файлами, например, метод openFileDialog1.ShowDialog, отображает на экране стандартное диалоговое окно выбора файла (смотри рисунок 5.2), в котором можно «путешествовать» по компьютеру в поисках нужного файла.

Настраивая свойства элемента openFileDialog1, можно задать фильтр имен открываемых файлов. Для этого необходимо элементу openFileDialog1 в окне Properties изменить свойство Filter. Присвойте этому свойству следующую текстовую строку:

```
Text files|*.txt|RTF files|*.rtf| All files|*.*
```

Строка фильтра состоит из блоков, разделенных символом «|». Первый блок задает название типа файла Text files, а второй— маску для имен файлов. Для текстовых файлов применяется маска *.txt.

Далее следует название формата RTF files. Для RTF используется маска *.rtf.

И, наконец, чтобы приложение могло открывать файлы любых типов (All files), используется маска *.* (пример на рисунке 5.2).

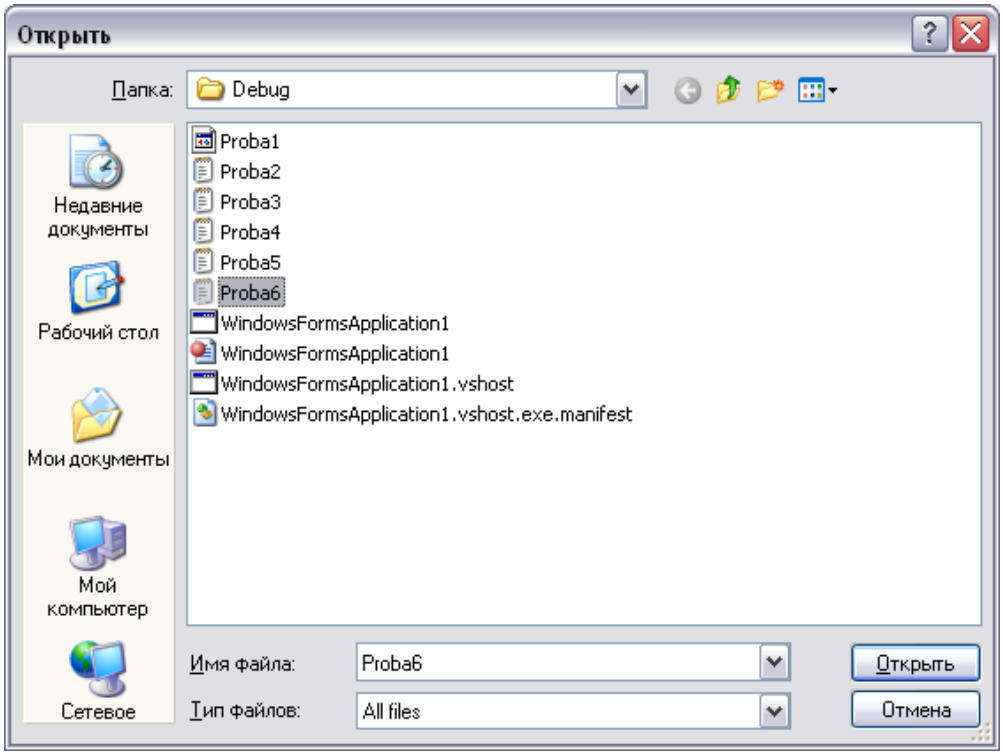


Рисунок 5.2 – Работа программы при открытии диалогового окна

По условию задачи мы будем работать с текстовыми файлами, но можно посмотреть, что читается при выборе файлов другого типа.

Дополнение к исходному коду программы:

```
private void loadTToolStripMenuItem_Click(object sender,
 EventArgs e)
{
 if (openFileDialog1.ShowDialog() == DialogResult.OK &&
  openFileDialog1.FileName.Length > 0)
 {
 try
 {
  richTextBox1.LoadFile(openFileDialog1.FileName,
RichTextBoxStreamType.PlainText);
 }
 catch (System.ArgumentException ex)
 {
 richTextBox1.LoadFile(openFileDialog1.FileName,
RichTextBoxStreamType.RichText);
 }
 this.Text = "Файл [" + openFileDialog1.FileName + "]";
      }
  }
```

Если в окне выбора файла пользователь щелкнул кнопку Открыть, метод ShowDialog возвращает значение DialogResult.OK. В условие

включена дополнительная проверка, что файл был выбран (длина строки полного пути к выбранному файлу openFileDialog1.FileName.Length должна быть больше нуля).

При открытии файла методом richTextBox1.LoadFile ему передается два параметра. В качестве первого параметра этому методу передается путь к файлу, а в качестве второго — тип файла.

Поскольку мы должны работать с текстовыми файлами, то основным типом является RichTextBoxStreamType.PlainText.

В том случае, если выбранный файл имеет формат, отличный от PlainText в методе LoadFile возникает исключение System.ArgumentException и наш обработчик выполняет повторную попытку загрузить файл, но на этот раз уже как файл типа RichText. Этот тип соответствует формату RTF.

Дополнительно, для информации, полный путь к открытому файлу будет отображен в заголовке главного окна нашего приложения.

5.3 Обработчик событий для записи в файл

Следующей по очереди на инструментальной панели находится кнопка записи клиентской области приложения в файл – команда SaveT. Рассмотрим порядок действий при написании этого обработчика события.

Для того чтобы реализовать в нашем приложении обработчик записи в файл, нам потребуется элемент SaveFileDialog. Перетащим значок этого элемента из окна Toolbox в окно нашей формы. Значок элемента SaveFileDialog1 появится внизу на панели под нашей формой.

Настраивая свойства элемента SaveFileDialog1, нужно отредактировать его свойства Filter и FileName.

Свойству Filter необходимо указать тип тестового файла – Text files|*.txt, а свойству FileName шаблон имени документа – doc1.txt. При этом по умолчанию документы будут сохраняться в файле с этим именем.

Дополнение к исходному коду программы:

```
private void saveTToolStripMenuItem_Click(object sender,
 EventArgs e)
{
 if (saveFileDialog1.ShowDialog() == DialogResult.OK &&
    saveFileDialog1.FileName.Length > 0)
 {
  richTextBox1.SaveFile(saveFileDialog1.FileName,
             RichTextBoxStreamType.PlainText);
  this.Text = "Файл [" + saveFileDialog1.FileName + "]";
 }
}
```

Дополнительно, для информации, полный путь к записываемому файлу будет отображен в заголовке главного окна нашего приложения.

Второй параметр метода richTextBox1.SaveFile позволяет выбрать тип сохраняемого файла. В нашем примере программа настроена на работу только с текстовыми файлами. Если передать через этот параметр значение RichTextBoxStreamType.RichText, то документ будет сохранен в формате RTF.

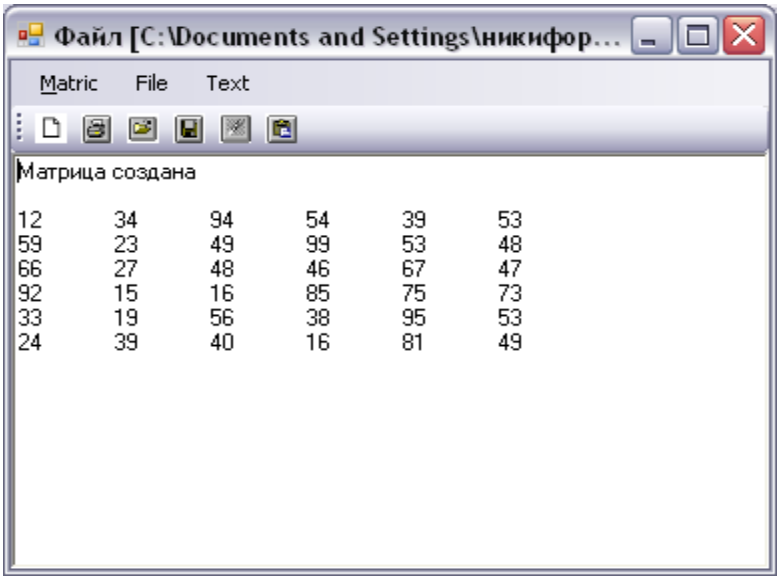


Рисунок 5.3 – Пример работы приложения с текстом

5.4 Обработчик событий для работы с текстом

Для работы с текстом клиентской области нашего приложения предусмотрено два обработчика событий, соответствующих командам меню очистка и преобразование.

Обработчик события очистка текста содержит один метод редактора текстов richTextBox1, который удаляет все строки текста:

```
private void clearTToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            richTextBox1.Clear();
        }
```

Обработчик события преобразования текста предназначен для выделения из текста клиентской области значений матрицы 6*6, обычно считанной из файла.

Исходный код метода обработки текста имеет следующий вид:

```
private void obraboToolStripMenuItem_Click(object sender,
EventArgs e)
{
 string ss;
 string[] clova;
 int k, n;
```

```
 int[] masi = new int[100];
 n = 0;
 ss = richTextBox1.Text;
 clova = ss.Split('\t', ' ');
 for (int i = 0; i < clova.Length; i++)
 {
  k = clova[i].Length;
  if (k == 2 || k == 3)
  {
   masi[n] = Int32.Parse(clova[i]); n++; }
   richTextBox1.AppendText(i.ToString() + " = " + clova[i] +
 " k= " + k.ToString() + "\n");
  }
 int j1, j2; j1 = j2 = 0;
 for (int i = 0; i < 36; i++)
 {
  a[j1, j2] = masi[i];
  j2++;
  if (j2 == 6) { j1++; j2 = 0; }
  if (j1 == 6 && j2 == 6) break;
 }
}
```

Здесь необходимы некоторые комментарии.

Объявлен массив clova типа string, в который из текста клиентской области приложения с помощью метода Split заносятся любые сочетания символов выделенных знаком табуляции, пробелом или возвратом :

```
ss = richTextBox1.Text;
clova = ss.Split('\t',' ','\n');
```

Определяется общее количество слов текста clova.Length.

В цикле все слова длиною 2 или 3 символа преобразуются в целые числа и записываются в массив целых чисел masi. Одновременно все слова и их длина выводятся на экран в клиентскую область (для контроля).

В последней части обработчика события выделенные целые числа из массива переписываются в матрицу 6×6.

Примерный вид содержимого клиентской области:

```
     Матрица создана

12   34   94   54   39   53
59   23   49   99   53   48
66   27   48   46   67   47
92   15   16   85   75   73
33   19   56   38   95   53
24   39   40   16   81   49
0 = Матрица k= 7
1 = создана k= 7
2 =  k= 0
3 =  k= 0
4 = 12 k= 2
5 = 34 k= 2
```

```
6 = 94 k= 2
7 = 54 k= 2
8 = 39 k= 2
9 = 53 k= 2
10 =  k= 0
11 = 59 k= 2
12 = 23 k= 2
13 = 49 k= 2
14 = 99 k= 2
15 = 53 k= 2
16 = 48 k= 2
17 =  k= 0
18 = 66 k= 2
19 = 27 k= 2
20 = 48 k= 2
21 = 46 k= 2
22 = 67 k= 2
23 = 47 k= 2
24 =  k= 0
25 = 92 k= 2
26 = 15 k= 2
27 = 16 k= 2
28 = 85 k= 2
29 = 75 k= 2
30 = 73 k= 2
31 =  k= 0
32 = 33 k= 2
33 = 19 k= 2
34 = 56 k= 2
35 = 38 k= 2
36 = 95 k= 2
37 = 53 k= 2
38 =  k= 0
39 = 24 k= 2
40 = 39 k= 2
41 = 40 k= 2
42 = 16 k= 2
43 = 81 k= 2
44 = 49 k= 2
45 =  k= 0
46 =  k= 0
```

После визуальной проверки клиентскую область можно очистить. Далее запускаем режим печати матрицы (смотри рисунок 5.4):

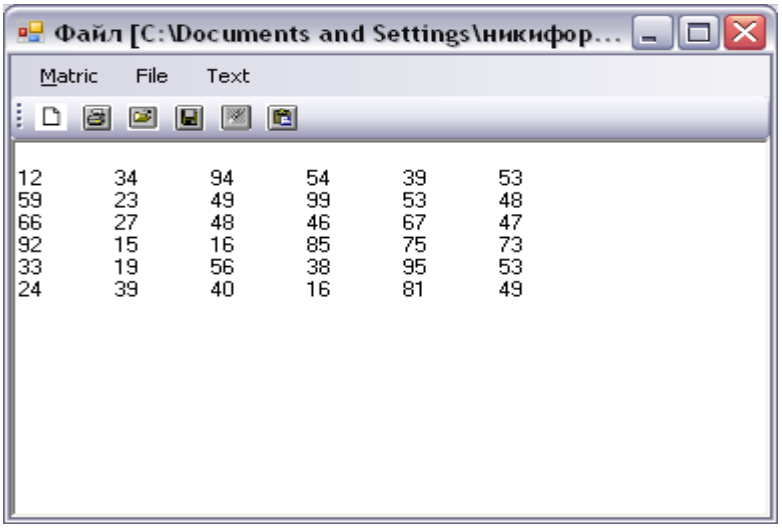


Рисунок 5.4 – Печать матрицы после преобразования клиентской области текста.

Видно, что нет привычного сообщения «Матрица создана».

5.5 Вопросы для самопроверки

1 Какие элементы управления относятся к диалоговым категориям?

2 Какой элементы управления предназначен для создания и обслуживания диалогового окна открытия файла?

3 Какой метод отображает на экране стандартное диалоговое окно системы Windows для выбора файла при его открытии?

4 Как настроить элемент управления OpenFileDialog на открытие только текстовых файлов?

5 Что произойдет, если в диалоговом окне открытия файла пользователь кликнул на кнопку «Открыть»?

6 Что означает часть условия openFileDialog1.FileName.Length > 0 в записи:
if (openFileDialog1.ShowDialog() == DialogResult.OK && openFileDialog1.FileName.Length > 0)?

7 Какой метод элемента управления RichTextBox1 используется для открытия файла?

8 Для чего предназначен элементы управления SaveFileDialog?

9 Что определяет свойство FileName элемента управления SaveFileDialog?

10 Что определяет параметр RichTextBoxStreamType.PlainText метода richTextBox1.SaveFile?

6 МНОГООКОННЫЕ ПРИЛОЖЕНИЯ

6.1 Создание «кнопочной» главной формы

Рассмотренные в предыдущих лекциях приложения были посвящены созданию однооконных приложений – все элементы управления и вывода результатов работы программ размещались в одном окне главной формы. В этой лекции мы рассмотрим технологию проектирования приложений с многооконным интерфейсом (Multiple-document interface, MDI).

Во многих многооконных приложениях используется так называемая главная кнопочная форма – главная форма приложения, на которой меню реализовано в виде «больших» кнопок с необходимыми комментариями.

В каких ситуациях имеет смысл проектировать главную форму как главную кнопочную форму? Так поступают достаточно часто. Представьте себе, что создаваемый проект предоставляет конечному пользователю несколько различных сервисов, и пользователь, начиная работу с проектом, выбирает нужный ему сервис. Главная форма может иметь меню, команды которого и позволяют пользователю выбирать нужный ему сервис. Если каждый сервис достаточно сложен и требует собственного интерфейса, то в таких ситуациях вместо стандартного меню удобнее использовать главную кнопочную форму. Роль команд меню в ней играют расположенные в форме командные кнопки. Выбор командной кнопки на форме соответствует выбору команды меню.

Освоение материала новой технологии программирования всегда понятнее при решении некоторой учебной задачи – задачи, при решении которой упор делается на освоение новой технологии, а не на алгоритм решения самой задачи. В таких учебных задачах алгоритм ее решения должен быть очевидным или рассмотрен в предыдущих лекциях.

Задача 6.1 Разработать приложение, позволяющее создавать массив из 6 геометрических фигур типа прямоугольник. Прямоугольники задаются координатами двух противоположных вершин. Координаты вершин формируются случайным образом в диапазоне целых чисел от минус 100 до 100. Пользовательский интерфейс должен содержать 3 формы. Главная «кнопочная» форма, на которой должны располагаться кнопки режимов работы программы с соответствующей поясняющей информацией. Окно формы для табличного представления и редактирования информации. Окно формы для графического представления информации.

Таким образом, главное окно должно содержать 3 кнопки режимов работы программы:

- создание массива прямоугольников;
- переход на окно табличной формы представления и редактирования значений координат вершин прямоугольников;

- переход к окну с графической формой представления прямоугольников.

На главной форме можно поместить дополнительную кнопку для информации об авторе программы, которая может содержать имя автора, его контактный телефон и фото автора (в дополнительной форме).

Естественно, для задания режимов работы программы можно использовать меню, рассмотренное в предыдущих лекциях, а пространство окна главной формы использовать для отображения одного из режимов работы. Однако, в учебных целях, мы рассмотрим организацию «кнопочной» формы как альтернативу меню.

Создадим проект пока с одной главной формой. Для красоты разместим на ней некоторый рисунок или фотографию. Для этого необходимо в окне элементов управления проекта выбрать элемент PictureBox и разместить его в окне формы. В свойствах элемента PictureBox необходимо выбрать Image, в правой части которого открыть диалоговое окно выбора файла изображения. Найти необходимое изображение и подтвердить его выборку. После этого изображение появится в окне главной формы приложения (смотри рисунок 6.1).

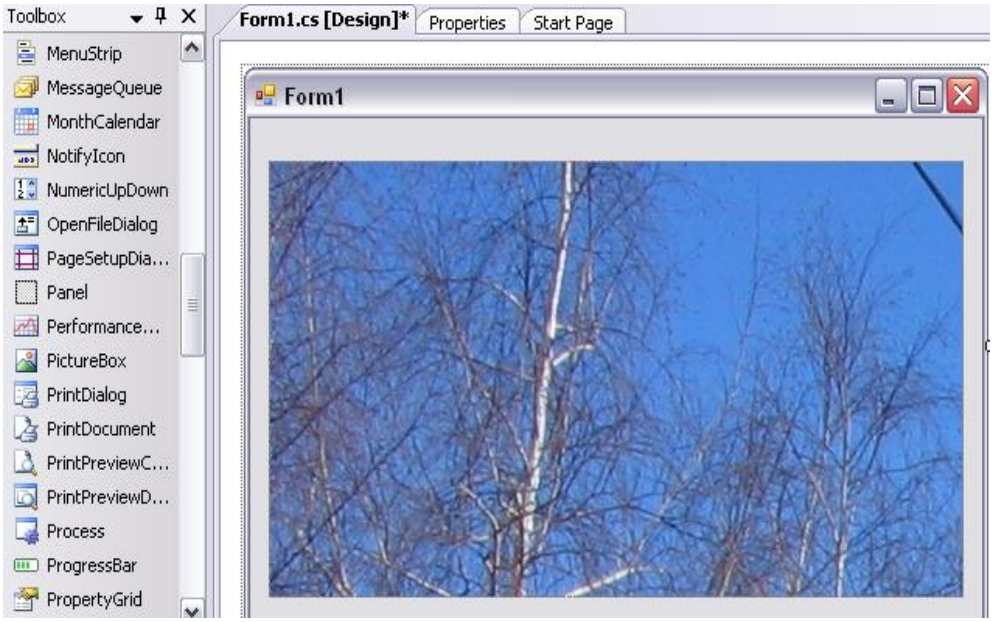


Рисунок 6.1 – Размещение рисунка в окне формы

Разместим на рисунке четыре кнопки – соответствующие режимам работы программы (смотри рисунок 6.2).

Реально в нашей программе три режима работы, но три кнопки в окне формы выглядят как-то неуютно, и мы добавим еще одну кнопку – сведения об авторе. Очень часто в меню программы включаются дополнительные режимы работы, например, режим Help, в котором содержится инструкция по работе с программой и отображается вся информация о нашем приложении.

В учебные программы часто включаются дополнительные теоретические сведения по алгоритму решения задачи или правильной организации режима диалога с программой.

Рисунок 6.2 – Главная «Кнопочная форма» приложения

6.2 Добавление новых форм приложения

Существует несколько вариантов добавления в проект приложения новых форм. Рассмотрим два основных.

Для того чтобы добавить в проект новую форму, щелкните правой клавишей мыши строку названия проекта WindowsFormsAplication1 в окне Solution Explorer (см. рисунок 6.3).

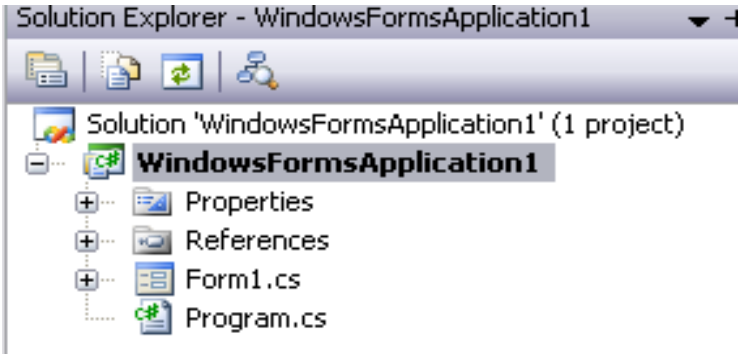


Рисунок 6.3 – Первый вариант добавления формы в проект

В появившемся меню режимов работы выберите режим Add и в нем команду Add Windows Form (см. рисунок 6.4).

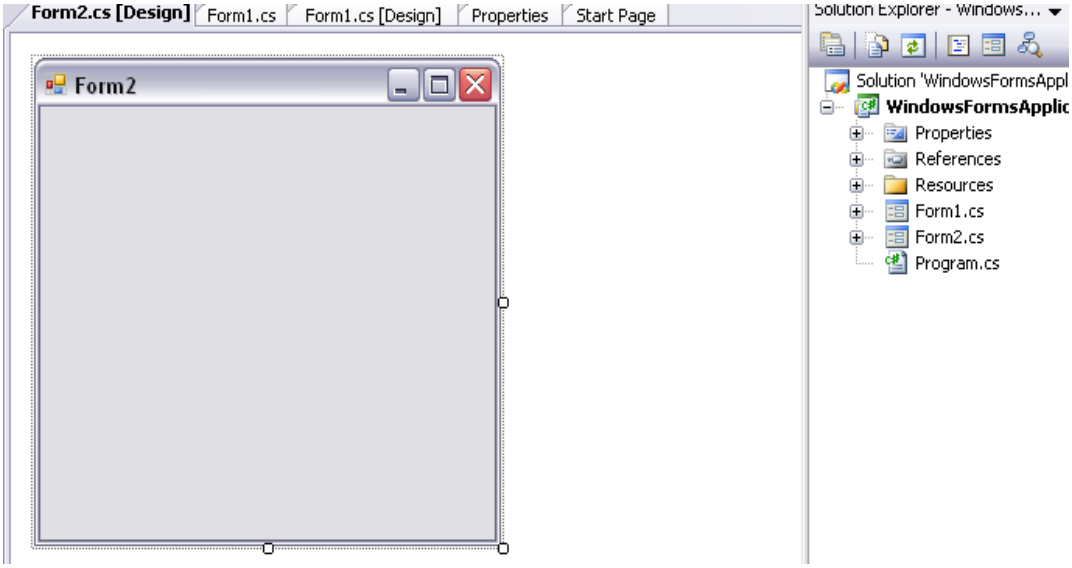


Рисунок 6.4 – Подключение новой формы к проекту приложения

Обратите внимание запись Form2.cs появилась в окне Solution Explorer проекта WindowsFormsAplication1.

Во втором варианте Вам необходимо выбрать режим Project, а в нем команду Add Windows Form . . . После этого подтвердить название предлагаемой формы нажатием кнопки Add.

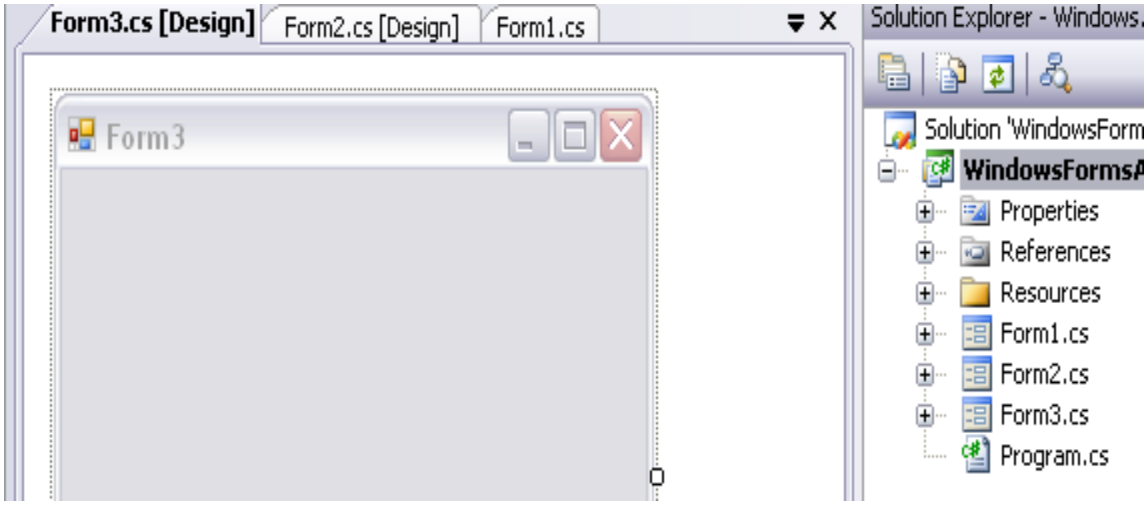


Рисунок 6.5 – Подключение третьей формы к проекту

Контроль подключения форм можно осуществлять как в окне Solution Explorer так и в окне редактирования форм.

6.3 Обработчики событий главной формы

Режим создания массива 6 прямоугольников можно реализовать в главной форме, а отображение результатов работы этого режима в формах 2 и 3.

Исходный код главной формы будет иметь следующий вид:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public static int[,] a = new int[6, 6];
  public Form1()
  {
   InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e)
  {
   Random rnd = new Random();
   for (int i = 0; i < 6; i++)
   {
    for (int j = 0; j < 4; j++)
    {
     a[i, j] = rnd.Next() % 201 - 100;
    }
   }
   for (int i = 0; i < 6; i++)
   {
    if(Math.Abs(a[i,0]-a[i, 2]) == Math.Abs(a[i, 1] - a[i,
3]))
       a[i, 4] = 1; else a[i, 4] = 0;
    a[i,5]=Math.Abs(a[i,0]-a[i,2])*Math.Abs(a[i,1]-a[i,3]);
   }
  }
  private void button2_Click(object sender, EventArgs e)
  {
   int y;
   Form2 f2 = new Form2();
   f2.ShowDialog();
  }
  private void button3_Click(object sender, EventArgs e)
  {
   Form3 f3 = new Form3();
   f3.ShowDialog();
```

```
  }
  private void button4_Click(object sender, EventArgs e)
  {
   Form4 f4 = new Form4();
   f4.Show();
  }
 }
}
```

При создании массива, в столбец 5 будем записывать информацию о прямоугольнике: 0 – квадрат, 1 – обычный прямоугольник, а в столбец 6, значение площади прямоугольника.

В нашем проекте запланировано несколько форм. Естественно возникает вопрос можно ли открывать одновременно несколько форм и как переходить с одной формы на другую? Ответ на этот вопрос зависит от того как было открыто окно.

Каждое окно (форма) можно открыть как модальное (как "диалоговое окно") или как немодальное (как обычное окно).

Если окно открывается методом Show(), то это задает обычное окно; если окно открывается методом ShowDialog(), то оно открывается как диалоговое окно. В чем разница? Из диалогового окна нельзя выйти, не закончив диалог и не закрыв форму. Открыв диалоговое окно, нельзя переключиться на работу с другой формой, не закончив диалог. Закрыть диалоговое окно можно разными способами. Можно щелкнуть по крестику, расположенному в правом верхнем углу формы или специальной кнопкой.

Если форма открыта методом Show, как не диалоговое окно, то, не закончив работу с открытой формой, можно перейти в главную форму или другую немодальную форму, поработать там, нажав какие-нибудь командные кнопки, получив нужную информацию, а затем снова вернуться к исходной форме.

В нашем приложении мы используем оба способа открытия окон форм.

Для закрытия окна можно также применять два метода – Hide() и Close(). Первый из этих методов скрывает форму, второй закрывает. Для диалоговых окон можно применять как метод Hide(), так и метод Close(): эффект будет одинаков – диалоговое окно будет закрыто.

Метод Hide() можно применять и для немодальных форм, открытых методом Show(). Окно, открытое не для диалога, можно временно скрыть, вызвав метод Hide(), а затем показать, вызвав метод Show().

Еще одно замечание, связанное с закрытием форм. Когда закрывается главная форма, закрываются все открытые к этому времени формы и приложение заканчивает свою работу. Когда же закрывается любая другая форма, закрывается только эта форма, остальные формы остаются открытыми.

6.4 Табличная форма представления и редактирования значений

Рассмотрим обработчики событий окна табличной формы представления и редактирования значений координат вершин прямоугольников.

Обработчики реализованы на форме 2, изображенной на рисунке 6.6. Для табличной формы представления значений массива использован элемент DataGridView. Рассмотрим его применение в интерфейсе, позволяющем пользователю вводить и отображать двумерные массивы.

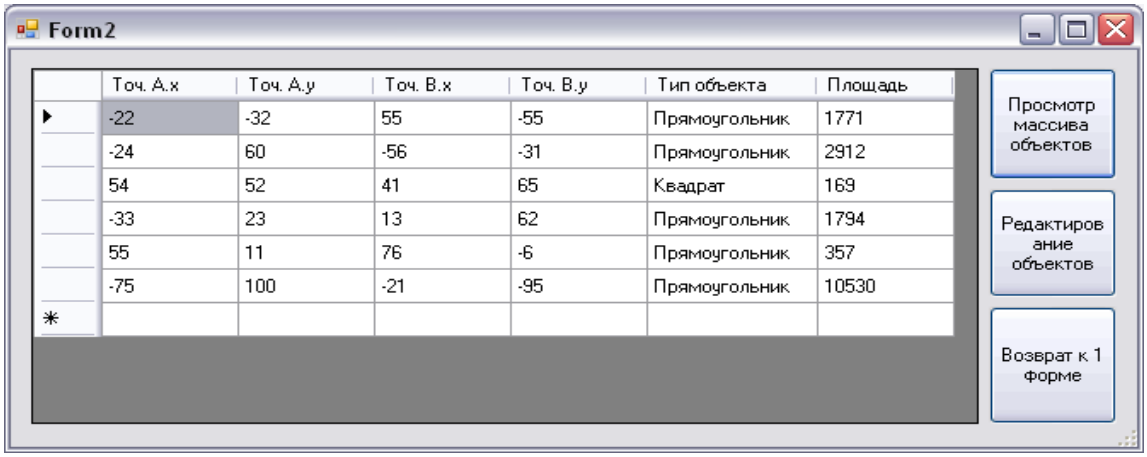


Рисунок 6.6 – Табличная форма представления значений массива

В свойствах элемента DataGridView выбираем Columns и запускаем редактор параметров столбцов таблицы, в котором мы добавляем необходимое количество столбцов и определяем название столбцов в окне формы и имя столбцов в программе (см. рисунок 6.7). С помощью редактора столбцов определяется их ширина в окне формы.

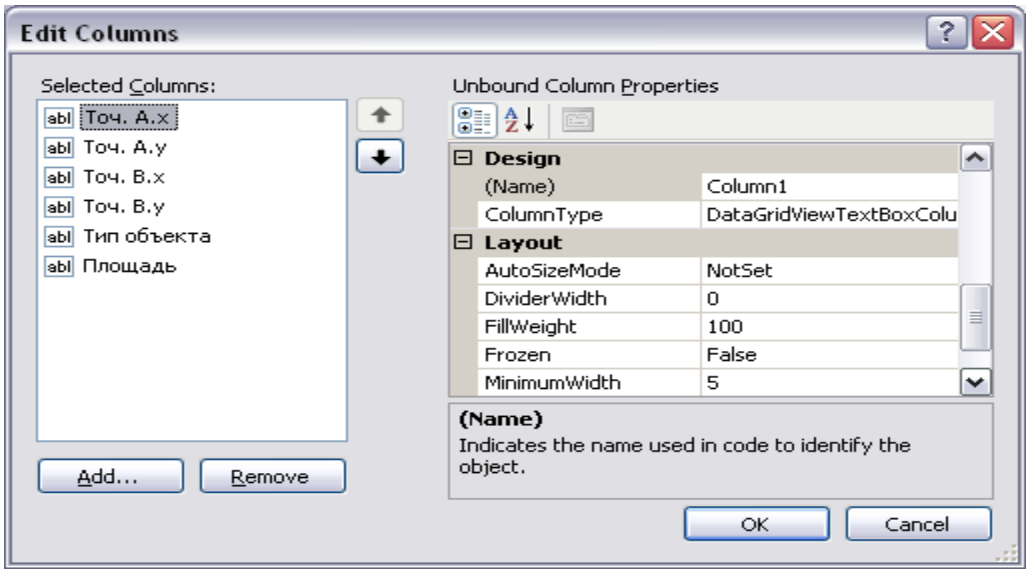


Рисунок 6.7 – Работа с редактором столбцов DataGridView

Исходный код формы 2 имеет следующий вид:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form2 : Form
 {
  public static int i, j;
  public static string kop;
  public Form2()
  {
   InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e)
  {
   for (i = 0; i < 6; i++)
   {
    dataGridView1.Rows.Add();
    for (j = 0; j < 6; j++)
    {
dataGridView1.Rows[i].Cells[j].Value=Form1.a[i,j].ToString();
    }
    if (Form1.a[i,4]==0) dataGridView1.Rows[i].Cells[4].Value
= "Прямоугольник";
      else dataGridView1.Rows[i].Cells[4].Value = "Квадрат";
   }
  }
  private void button2_Click(object sender, EventArgs e)
  {
   Close();
  }
  private void button3_Click(object sender, EventArgs e)
  {
   string elem = "";
   bool ok;
   int k;
   for (i = 0; i < 6; i++)
    for (j = 0; j < 4; j++)
    {
     do
     {
      ok = true;
      try
      {
       elem = dataGridView1.Rows[i].Cells[j].Value.ToString();
       Form1.a[i, j] = int.Parse(elem);
```

```
       }
       catch (Exception any)
       {
        Form5 f5 = new Form5();
        if (f5.ShowDialog() == DialogResult.OK) k = 0;
        dataGridView1.Rows[i].Cells[j].Value = kop;
        ok = false;
       }
      } while (!ok);
     }
     for (i = 0; i < 6; i++)
     {
      if (Math.Abs(Form1.a[i,0]-Form1.a[i,2]) ==
Math.Abs(Form1.a[i, 1] - Form1.a[i, 3]))  Form1.a[i, 4] = 1;
                 else Form1.a[i, 4] = 0;
      Form1.a[i, 5] = Math.Abs(Form1.a[i, 0] - Form1.a[i, 2]) *
Math.Abs(Form1.a[i, 1] - Form1.a[i, 3]);
     }
    }
   }
}
```

На форме 2 реализованы 3 обработчика событий – вывод матрицы в таблицу элемента DataGridView, редактирование значений матрицы и возврат к 1 форме. Рассмотрим каждый фрагмент кода отдельно.

Вывод матрицы в таблицу элемента DataGridView содержит цикл перебора строк таблицы, в котором программно добавляется очередная строка и запускается цикл перебора столбцов таблицы.

Поскольку массив сформирован в виде глобальной переменной формы 1, то его использование в форме 2 требует следующей записи Form1.a[i,j].

Значение столбца 4 двумерного массива анализируется на наличие 0 – признак равенства сторон прямоугольника (квадрат).

В последний столбец выводится значение площади прямоугольника.

Процесс редактирования элементов DataGridView включает непосредственное изменение значений на форме 2 и запуск обработчика «Редактирование объекта», в котором содержимое элементов DataGridView переписывается в массив формы 1. Процесс перезаписи находится в охраняемом блоке. Если вы вместо цифр случайно введете символ буквы или другой знак, то запускается обработчик ошибочной ситуации, который создаст и откроет диалоговое окно 5 для повторного ввода ошибочного значения (см. рисунок 6.8).

Исходный код формы 5:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
```

```
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form5 : Form
 {
  public Form5()
  {
   InitializeComponent();
   label1.Text = "a[" + Form2.i.ToString() + "," +
Form2.j.ToString() + "]= ?";
  }
  private void button1_Click(object sender, EventArgs e)
  {
   Form2.kop = textBox1.Text;
   Close();
  }
 }
}
```

Исправление ошибок ввода продолжается пока не будет введено правильное числовое значение.

Если в результате редактирования элементов DataGridView изменяется тип прямоугольника, например, получается квадрат, то это учитывается в столбце 4 массива. Одновременно перезаписывается новое значение площади нового прямоугольника.

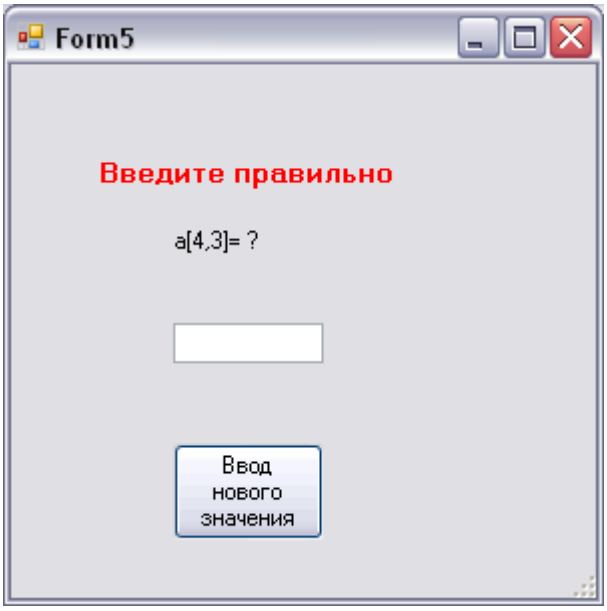


Рисунок 6.8 – Исправление ошибки ввода

В приведенном коде формы 2 программно добавляется очередная строка элемента DataGridView. Можно программно формировать и столбцы элемента DataGridView (обычно это делается при однотипных столбцах). Например, если нам необходимо подготовить элемент

DataGridView на 6 столбцов, то можно использовать следующий фрагмент кода для программного добавления столбцов:

```
dataGridView1.Columns.Clear();
DataGridViewColumn column;
for (int i = 0; i < 6; i++)
{
 column = new DataGridViewTextBoxColumn();
 column.DataPropertyName = "Column" + i.ToString();
 column.Name = "Column" + i.ToString();
 dataGridView1.Columns.Add(column);
}
```

В первой строке кода мы удаляем предыдущее «состояние столбцов» элемента DataGridView, объявляем переменную column, затем в цикле создаем объект column, которому присваиваем имя для формы и имя для программы, потом добавляем созданный объект нашему элементу.

6.5 Графическая форма представления прямоугольников

Рассмотрим обработчики событий в окне графической формы представления прямоугольников.

Обработчики реализованы на форме 3, изображенной на рисунке 6.9.

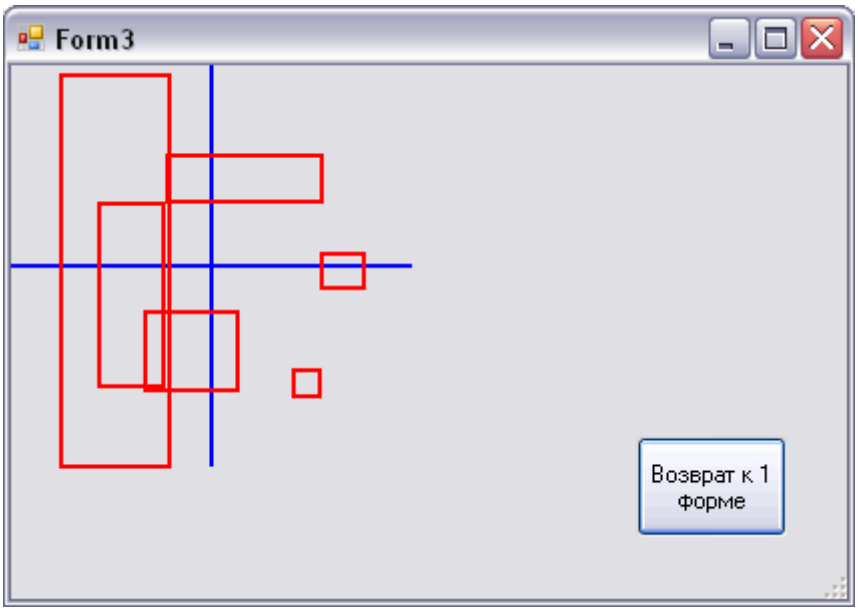


Рисунок 6.9 – Окно графической формы представления прямоугольников

Сразу хочется отметить, что дизайн представленной формы долек от совершенства. В коде формы 3 нет ничего того, что для Вас является новым (эта форма добавлена в проект для большего числа форм). Поэтому просто приведем исходный код формы:

```
using System;
```

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form3 : Form
 {
  public Form3()
  {
   InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e)
  {
   Close();
  }
  private void Form3_Paint(object sender, PaintEventArgs e)
  {
   int ax, ay, bx, by;
   Pen myPen = new Pen(Color.Blue, 2);
   Graphics g = e.Graphics;
   g.DrawLine(myPen, 0, 100, 200, 100);
   g.DrawLine(myPen, 100, 0, 100, 200);
   myPen = new Pen(Color.Red, 2);
   for (int i = 0; i < 6; i++)
   {
    if (Form1.a[i, 0] < Form1.a[i, 2]) ax = Form1.a[i, 0];
           else ax = Form1.a[i, 2];
    if (Form1.a[i, 1] < Form1.a[i, 3]) ay = Form1.a[i, 1];
           else ay = Form1.a[i, 3];
    bx = Math.Abs(Form1.a[i, 0] - Form1.a[i, 2]);
    by = Math.Abs(Form1.a[i, 1] - Form1.a[i, 3]);
    g.DrawRectangle(myPen, ax+100, ay+100, bx, by);
   }
  }
 }
}
```

Еще одна форма проекта «Информация об авторе» также добавлена для демонстрации возможностей многооконного приложения.

Рисунок 6.10 – Окно режима программы «Информация об авторе»

Исходный код формы 4 содержит только обработчик кнопки возврата к 1 форме. Остальное реализовано с помощью свойств элементов формы 4:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form4 : Form
    {
        public Form4()
        {
            InitializeComponent();
        }

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

6.6 Вопросы для самопроверки

1 Что означает сокращение MDI?

2 В каких ситуациях имеет смысл проектировать главную форму как главную кнопочную форму?

3 Какой элемент управления обычно используется для размещения рисунков на форме?

4 Как добавить в проект новую форму с помощью окна Solution Explorer?

5 Как добавить в проект новую форму с помощью режима Project?

6 Чем отличается диалоговое (модальное) окно от обычного окна формы?

7 Каким методом открывается обычное (не модальное) окно формы?

8 Каким методом открывается модальное окно формы?

9 Что делает следующий фрагмент программы:

```
private void button3_Click(object sender, EventArgs e)
    {
       Form5 f5 = new Form5();
       if (f5.ShowDialog() == DialogResult.OK) k = 0;
    }   ?
```

10 Какой элемент управления часто используется для табличной формы представления информации?

7. ПОНЯТИЕ КЛАССА

7.1 Понятие класса

Мы уже отмечали, что язык C# является объектно-ориентированным языком программирования, основным понятием которого является понятие класса. Во всех примерах программ при изучении языка C# мы постоянно использовали структуры типа класс и даже делали робкие попытки дать определение класса, но всегда при этом говорили, что изучением класса мы займемся в одной из следующих лекций.

В этой лекции мы будем заниматься только классами.

Начнем с понятия класса. Естественно начнем с определений, которые приводят в своих книгах по программированию на языке C# преподаватели ВУЗов.

Фаронов В.В. определяет класс как фрагмент кода, перед которым стоит зарезервированное слово class [1]. Или «Класс – это всего лишь тип данных, то есть «схема», по которой изготавливаются объекты – реальные экземпляры класса.».

Павловская Т.А. [2] приводит следующее определение класса «Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых экземплярами класса.».

Необходимо отметить, что ООП существовало до появления языка C# и понятие класса существует уже давно. Самое короткое определение, которое мы встречали в литературе это следующее определение класса – «Классы – это типы, определяемые программистом».

В этом определении отображена очень важная особенность класса – это тип данных, новый по сравнению с массивами, записями или структурами. Однако не всякий тип данных, определяемый программистом, является классом. Второй важной особенность в определение класса должен быть состав класса или хотя бы намек на его состав и в тоже время определение класса должно быть кратким и легко запоминаемым, например.

```
Класс  это тип данных, содержащий поля, методы и события.
```

Тип данных - это семантическая единица, которая описывает свойства и поведение множества объектов, называемых экземплярами класса.

Синтаксически класс представляет описание данных, называемых полями класса, описание методов класса и описание событий класса.

Некоторые авторы выделяют классы представленные модулями в самостоятельные группы, например, классы элементов управления при визуальном программировании. Такие классы несут дополнительную

нагрузку. Они являются самостоятельными архитектурными единицами построения проектов.

Изучение класса начнем с изучения его формата записи.

Обязательный формат записи класса содержит служебное слово class, за которым, располагается его имя и далее в фигурных скобках находится тело класса. Это так называемый минимальный состав описания класса.

Общее описание класса, включающее необязательные элементы (они выделены квадратными скобками), имеет следующий формат записи:

```
[ атрибуты ] [ спецификаторы ]
class    имя_класса  [ : родители ]
   { тело_класса  } ,
```

где

атрибуты – задают дополнительную информацию о классе;

спецификаторы – определяют условие доступа к составляющим класса;

родители – базовые классы, которые наследует наш класс;

тело класса – определяет состав элементов класса.

Возможными спецификаторами в объявлении класса могут быть abstract, sealed и protected**,** о которых подробно будет говориться при рассмотрении наследования. Спецификаторы private, public, static и internal определяют доступность класса для программы. Говорят, что спецификатор private полностью закрывает видимость класса, а public делает класс видимым (доступным) для любого фрагмента программы. По умолчанию класс имеет спецификатор доступа internal – класс доступен в сборке, в которой он определен. Спецификатор static позволяет использовать класс и его элементы, не создавая переменной этого класса (объекта класса).

Все перечисленные спецификаторы применимы как для класса, так и для отдельных его членов, например, полей, методов.

Некоторые необязательные элементы формата описания класса мы будем рассматривать по мере изучения дисциплины.

Класс это тип данных – шаблон, который можно «наполнить» некоторыми значениями, т.е. получить экземпляр класса – переменную типа класс или объект.

Подставляя в программу (в модель решения задачи) различные значения, мы будем получать разные объекты класса (различный результат работы программы), но тип класса (код программы) останется неизменным.

Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется вложенным.

В языке C# класс является ссылочным типом и для размещения объекта класса в памяти компьютера необходимо использовать оператор new.

7.2 Состав класса

Как мы уже отмечали, тело класса может содержать данные, методы их обработки и события – все эти составляющие часто называют элементами класса.

Рассмотрим основные элементы класса и их назначение:

– константы класса хранят неизменяемые значения;

– поля класса (типы и имена переменных класса);

– методы класса это поименованный фрагмент кода программы, предназначенный для работы с данными класса;

– свойства класса это совокупность методов, позволяющих классу обмениваться (читать или записывать) значениями полей класса с другими классами программы;

– конструкторы класса это специальные методы класса, которые предназначены для создания объектов класса и присваивания начальных значений полям класса;

– деструкторы класса это специальные методы, определяющие порядок действий при освобождении ресурсов, выделенных объекту;

– события класса это специальные методы, позволяющие классу реагировать на действия пользователя или на определенные изменения в программе;

– типы это типы данных, внутренние по отношению к классу. Например, перечисления, структуры, классы, делегаты, интерфейсы.

– индексаторы это средство доступа к элементам данных класса (обычно массивам) по их порядковому номеру;

–операции это специальные действия с объектами класса с помощью знаков операций.

Данными класса могут быть константы или переменные (поля) класса. При объявлении данных в классе обычно указывается спецификатор доступа к нему, например,

```
private int a;
```

Общий формат записи данных класса при их объявлении имеет следующий вид:

```
[ атрибуты ] [ спецификаторы ]
[ const ] тип имя [= начальное_значение].
```

Обычно данные класса «закрывают для программы» – используют спецификатор private. Если перед данными используются спецификатор public, то они являются доступными «программе».

По умолчанию, как для данных, так и для методов применяется спецификатор private.

Объект – это переменная типа класс и при его создании в памяти компьютера выделяется отдельная область, в которой хранятся значения элементов класса.

Однако в классе могут присутствовать статические элементы класса, которые существуют в единственном экземпляре для всех объектов класса. Часто статические данные называют данными класса, а остальные – данными экземпляра класса, т.е. объекта.

Доступ к некоторым элементам класса (методам) и полям возможен только после создания объекта. Если доступ разрешен, то для обращения к ним используется оператор «точка», например, для некоторого объекта stud доступ к полю name возможен следующим образом stud.name = “Иванов”;.

Аналогичным образом для объекта можно вызывать метод его класса, например, stud.poisk(a);, где poisk(int a) метод класса, для которого создан объект stud**.**

Из синтаксиса следует, что классы могут быть вложенными. Такая ситуация довольно редкая. Ее стоит использовать, когда некоторый класс носит вспомогательный характер, разрабатывается в интересах другого класса, и есть полная уверенность, что внутренний класс никому не понадобится, кроме класса, в который он вложен и, возможно, его потомков.

Внутренние классы обычно имеют модификатор доступа private или protected.

7.3 Методы класса

Метод – это поименованный функциональный элемент класса, предназначенный для работы с данными и методами данного класса.

Методы определяют набор действий, которые доступны классу (часто говорят, что они определяют поведение класса).

Метод описывается один раз, а может вызываться для различных объектов класса столько раз, сколько необходимо.

Общий формат записи методов класса имеет следующий вид:

```
[ атрибуты ] [ спецификаторы ] тип метода     имя метода ( [ параметры ] )
            {тело метода}
```

Например,

```
static void Main(string[] args)
  {  }
```

Наиболее часто встречаемые спецификаторы это private, public и static.

Любые методы класса, объявленные со спецификатором private, доступны только в методах данного класса.

Спецификатор public делает метод доступным в любом месте программы.

Спецификатор static означает, что к методу можно обращаться «на уровне класса» не создавая объект класса – это очень важно, так как в данной дисциплине мы будем очень часто использовать статические методы.

Другими методами доступными программе без создания объекта класса являются конструкторы класса (они и создают объект).

Доступ к остальным методам возможен только после создания объекта класса.

Если спецификатор не указан то (по умолчанию) считается, что данный метод класса имеет спецификатор private.

Тип метода может задаваться любым определенным в программе или стандартным типом языка C# или void – без типа. Например:

```
int kol(int a) { … }
public double sym(out float r)  { … }
public void poisk(ref float s) { … }
public int funkcij( int a, out int b, params int[] c) { … }
```

Если задан тип метода (кроме void), то последним оператором тела метода должен быть оператор return, возвращающий результат работы метода. При этом метод необходимо присваивать некоторой переменной или использовать как выражение в операторах языка C#. Часто такие методы называют функциями.

Если перед методом указан тип void, то метод не должен возвращать результат своей работы с помощью оператора return (оператор return в этом случае отсутствует в теле метода). Часто такой метод называют процедурой – ее не надо присваивать переменной, а можно записывать как отдельную подпрограмму – процедуру (имя метода с указанием в круглых скобках ее параметров).

Имя метода – идентификатор, определяемый программистом. Желательно в имя метода закладывать смысловое назначение метода, например, sym, max, poisk и т.д.

Параметры метода (формальные параметры) предназначены для обмена данными между методом и программой. Часто параметры метода называют средством «настройки» метода на выполнение необходимого алгоритма.

В языке C# различают следующие параметры методов:

– параметры-значения (входные параметры, т.е. получаемые методом);

– выходные-параметры (помечаются служебным словом out);

– параметры-ссылки (помечаются служебным словом ref);

– параметры-массивы (помечаются служебным словом params).

Параметры-значения не имеют помечающего служебного слова.

Параметры метода класса разделяются запятыми. Параметр-массив в методе может быть только один и должен быть последним в списке параметров.

Если в методе объявлены параметры - значения, то это означает, что метод получает в свое распоряжение копии некоторых переменных. Метод может изменять значения этих копий, но их оригинал (в программе) остается неизменным. По окончании работы метода параметры - значения удаляются из памяти компьютера.

Выходные - параметры метода предназначены для передачи результатов работы метода в программу. В теле метода обязательно должны находиться операторы присваивания выходным - параметрам некоторых значений, иначе выдается сообщение об ошибке во время компиляции программы.

Если в методе объявлены параметры - ссылки, то фактически метод получает в свое распоряжение адреса соответствующих переменных и может их использовать по своему алгоритму (читать или писать новые значения).

Объявленные в методе параметры - массивы предназначены для работы с произвольным числом фактических переменных. При этом формальному параметру, стоящему за служебным словом params ставится в соответствие массив произвольной длины данных.

Таким образом, через свои параметры метод может, как получать необходимые значения (параметры - значения и параметры - ссылки), так и возвращать результаты своей работы (выходные-параметры и параметры - ссылки).

Тело метода содержит фрагмент кода программы, реализующий некоторый алгоритм. При этом метод выступает как некоторый шаблон действия с формальными параметрами. В программе вместо формальных параметров необходимо использовать реальные переменные – фактические параметры и шаблон действия метода будет применяться для реальных переменных.

7.4 Структура объекта

Рассматривая классы, мы отмечали, что переменной типа класс является объект. Объект переменная, следовательно, ей выделяется место в памяти компьютера, в которой хранится информация объекта.

Рассмотрим, что конкретно хранится в памяти, соответствующей объекту.

Естественно, то значения всех полей данных класса.

Специальное поле this (параметр по ссылке), который формируется автоматически при создании объекта и содержит адрес объекта.

Фактически связь объекта с методами класса происходит через параметр this. Каждый метод класса может непосредственно обращаться к

параметру this для работы с элементами текущего объекта. Поскольку значение this всегда соответствует текущему объекту (объекту, с которым в текущий момент работает программа), то методы класса будут работать с элементами текущего объекта.

Использование указателя this позволяет не создавать в каждом объекте копии методов класса, для работы с объектом. Таким образом, методы класса не тиражируются для каждого объекта.

7.5 Пример учебной программы

Рассмотрим чисто учебный пример по созданию класса треугольник (на основе примера из первой лекции).

На этапе визуального программирования мы будем использовать те же три стандартных элемента управления из окна Toolbox: статический текст или метка (Label), поле ввода или окно редактирования (TextBox) и командную кнопку (Button), но расположи их на форме в другом порядке.

Исходный код файла Form1.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
  {
   public class treyg
     {
       private int a, b, c, p;
       public string ss;
       public void vvod(int sa, int sb, int sc)
       {
   if (sa > 0 && sb > 0 && sc > 0)
        {
         if (sa + sb > sc && sa + sc > sb && sb + sc > sa)
          {
           a = sa; b = sb; c = sc;
           p = a + b + c;
           ss = "Периметр треугольника = " + p.ToString();
          }
         else
          ss = "Одна из сторон треугольника больше суммы двух
других Повторите ввод ";
        }
        else
```

```
ss = "Одна из сторон треугольника меньше 0! Повторите ввод ";
        }
      }

   public Form1()
    {
      InitializeComponent();
    }
   private void button1_Click(object sender, EventArgs e)
    {
     int A, B, C;
     treyg t = new treyg();
     A = Convert.ToInt32(textBox1.Text);
     B = Convert.ToInt32(textBox2.Text);
     C = Convert.ToInt32(textBox3.Text);
     t.vvod(A,B,C);
     textBox4.Text = t.ss;
   }
 }
}
```

Рассмотрим подробнее некоторые элементы класса и их использование в программе.

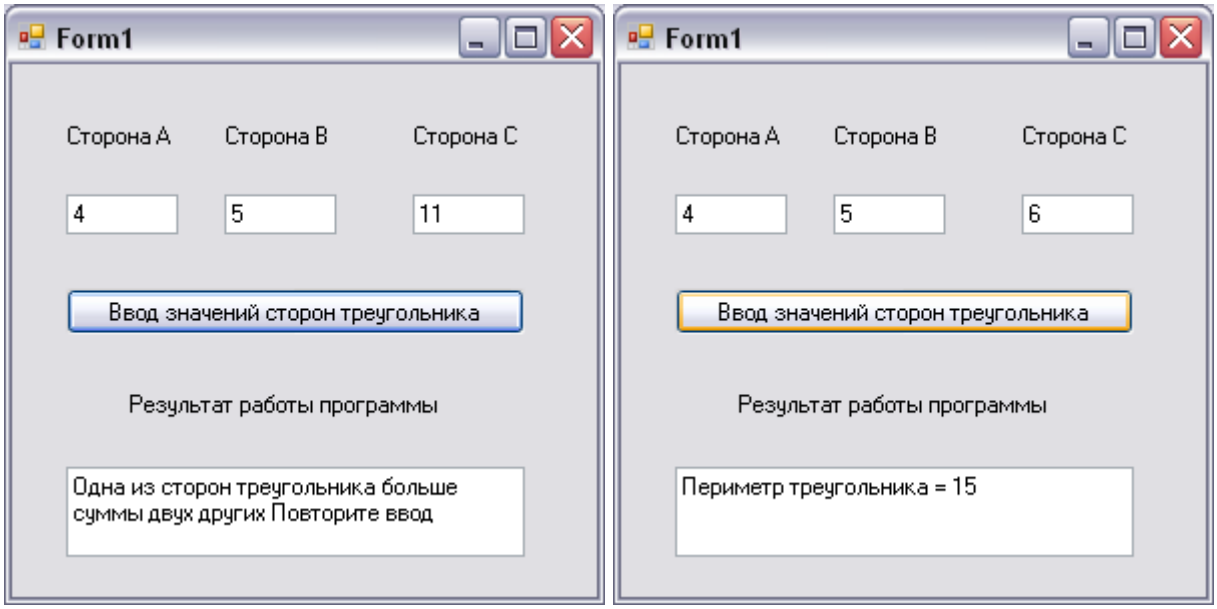


Рисунок 7.1 – Окна программы «Треугольник Класс»

В первую очередь для работы с данными и методами класса class treyg необходимо создать объект этого класса – переменную t

```
treyg t = new treyg();
```

Данные класса private int a, b, c, p; являются закрытыми. Это означает, что доступ к элементам данных класса возможен только с помощью его методов. Например, если после создания объекта t попытаться присвоить новое значение элементу данных b (t.b = 3;), то это

действие вызовет сообщение об ошибке т.к. непосредственное обращение к элементам данных класса запрещено спецификатором доступа private.

В классе treyg использованы два метода – конструктор (по умолчанию) и метод ввода значений сторон треугольника.

Конструктор создает объект с «нулевыми» значениями его полей данных.

Задание значений полям данных класса treyg осуществляется методом: public void vvod(int sa, int sb, int sc), которому в качестве фактических параметров задаются значения переменных A, B и C, введенные в режиме диалога.

В программе рассмотрены варианты «неправильного» задания значений сторон треугольника и печать соответствующих комментарий. Однако в ней не предусмотрена защита от нажатия кнопки «Ввод значений сторон треугольника» с «пустыми» окнами ввода.

7.6 Доступ к полям

Каждое поле имеет модификатор доступа, принимающий одно из четырех значений: public, private, protected, internal. Возможно совместное задание двух атрибутов protected и internal.

Модификатор private. Модификатор private является атрибутом доступа по умолчанию. Он закрывает поля от всех других классов, разрешая прямой доступ к ним (чтение и запись) только методам самого класса. Помните, все поля всегда доступны всем методам класса. Они являются для методов класса глобальной информацией, с которой работают все методы, извлекая из полей нужные им данные и изменяя их значения в ходе работы.

Модификатор protected. Этот модификатор открывает поля классам наследникам. Если класс A объявил некоторое поле с модификатором protected, то методы класса B, который является наследником класса A и, следовательно, наследует поля класса A, могут непосредственно работать с наследуемыми полями.

Модификатор internal. Этот модификатор открывает поля дружественным классам. Два класса A и B называются дружественными, если они принадлежат одной сборке - одному проекту. Если класс A объявил некоторое поле с модификатором internal, то методы дружественного класса B, являющегося клиентом класса A, могут непосредственно работать с таким полем.

Комбинация атрибутов protected и internal. Эта комбинация открывает поле тем классам, которые являются либо наследниками, либо дружественными классами. Если требуется более строгое ограничение доступа к полю, чтобы оно было доступно только тем наследникам, которые являются дружественными классами, то сам класс нужно

объявить с модификатором internal, а соответствующее поле - с модификатором protected.

Если поля доступны только для методов класса, то они имеют модификатор доступа private, который можно опускать. Такие поля считаются закрытыми, но часто желательно, чтобы некоторые из них были доступны в более широком контексте. Если некоторые поля класса A должны быть доступны для методов класса B, являющегося потомком класса A, то эти поля следует снабдить модификатором protected. Такие поля называются защищенными. Если некоторые поля должны быть доступны для методов классов B1, B2, и так далее, дружественных по отношению к классу A, то эти поля следует снабдить модификатором internal, а все дружественные классы B поместить в один проект (assembly). Такие поля называются дружественными. Наконец, если некоторые поля должны быть доступны для методов любого класса B, которому доступен сам класс A, то эти поля следует снабдить модификатором public. Такие поля называются общедоступными или открытыми.

7.7 Вопросы для самопроверки

1 Понятие класса?
2 Понятие свойства класса?
3 Понятие конструктора класса?
4 Понятие деструктора класса?
5 Понятие события класса?
6 Понятие индексатора класса?
7 Назначение поля this объекта?
8 Как называют переменную типа класс?
9 Что означает служебное слово static в описании класса?
10 Что означает служебное слово public в описании данных класса?

8 ЭЛЕМЕНТЫ КЛАССОВ

8.1 Конструкторы

Основное назначение конструктора – создать объект и элементам данных этого объекта присвоить некоторые начальные значения. Иногда этот процесс называют инициализацией объекта.

Создание объекта является основной задачей конструктора.

Присваивание элементам данных объекта некоторых значений может осуществляться различными способами.

В зависимости от способа определения значений элементов данных объекта различают:

– конструкторы с умолчанием;

– конструкторы с заданием параметров;

– множественные конструкторы.

Конструктор с умолчанием вызывается в программе, без каких либо параметров. Например,

```
treyg t1 = new treyg();
```

Это означает, что элементам данных объекта t1 присваиваются некоторые фиксированные значения – обычно нулевые.

Реализация такого конструктора может включать операторы присваивания полям данным класс некоторых фиксированных значений.

Например, в наш класс можно добавить конструктор с умолчанием, который присваивает (по умолчанию) сторонам треугольника фиксированные значения 3, 4 и 5. В этом можно убедиться, если добавить метод печати периметра объекта:

```
public treyg()
      {
       a = 3; b = 4; c = 5;
      }
    public void printO()
      {
         p = a + b + c;
         ss = "Периметр треугольника = " + p.ToString();
  }
```

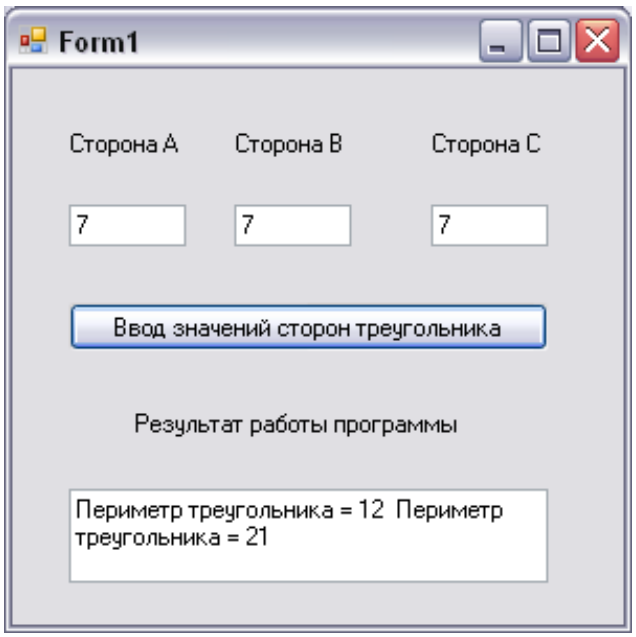


Рисунок 8.1 – Использование в программе конструктора с фиксированными начальными значениями

Конструкторы с заданием параметров позволяют определять начальные значения элементов данных объекта во время создания объекта. Объявление таких конструкторов в классе выглядит следующим образом:

```
public treyg(int sa, int sb, int sc)
       {
        a = sa; b = sb; c = sc;
 }
```

Многие программисты включают в реализацию конструктора проверку на допустимые значения элементов данных объектов, создавая так называемые «умные» конструкторы. Например, в реализацию конструктора класса треугольник, можно включить следующие проверки элементов данных:

– все стороны треугольника данных должны быть больше 0;

– сумма любых двух сторон треугольника должна быть больше третьей стороны.

Если условия не выполняются необходимо выдать сообщение и присвоить элементам данных объекта некоторые фиксированные значения или повторить ввод.

Объявление такого «умного» конструктора ни чем не отличается от объявления обычного конструктора, а реализация включает все необходимые проверки в функции ввода данных класса, например:

```
public treyg(int sa, int sb, int sc)
     {
       vvod(sa,sb,sc);
}
```

Изменим код программы нашего обработчика события следующим образом – умышленно введем в конструкторе неправильные значения сторон треугольника:

```
private void button1_Click(object sender, EventArgs e)
    {
     int A, B, C;
     treyg t = new treyg(3,5,9);
     t.printO();
     A = Convert.ToInt32(textBox1.Text);
     B = Convert.ToInt32(textBox2.Text);
     C = Convert.ToInt32(textBox3.Text);
     t.vvod(A,B,C);
     textBox4.Text = t.ss;
}
```

Работа программы на рисунке 8.2

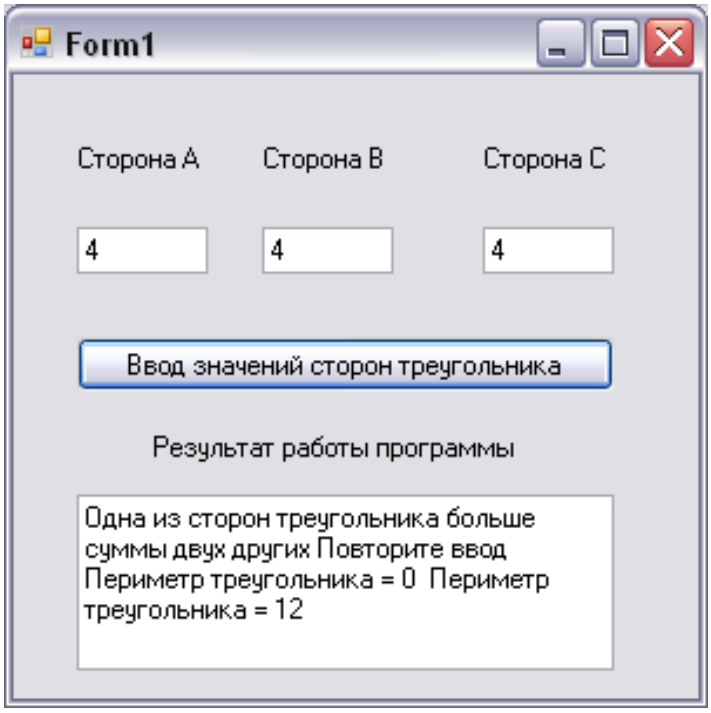


Рисунок 8.2 – Работа программы с заведомо неправильными значениями в конструкторе

Множественные конструкторы используют перегрузку функций при обработке аргументов разного типа.

Такие конструкторы используются, когда значения данных могут задаваться в различных формах, например, целым или вещественным числами или строкой.

Обычно такие конструкторы применяются для обработки дат.

Используя перегрузку функций можно в описание класса включать несколько конструкторов, «понимающих» данные различного типа.

Например, текущую дату можно задавать по умолчанию или вводить разными способами. Возможны, например, следующие три варианта значений ввода даты:

целыми числами (месяц, день и год – 23 12 07);

некоторым текстом (23 декабря 2007 года);

набором с использованием дополнительных символов (20.10.07 или 17/09/2007).

```
class date
 {
   . . .
  date() { . . . }
  date(int mm, int dd, int gg) { . . . }
    date (string tekct) { . . . }
  . . .
};
```

При инициализации объекта возможен любой из предложенных вариантов.

8.2 Деструкторы

В языке C# классы могут содержать специальные методы деструкторы. В функциональном плане они должны осуществлять действия, обратные тем, что реализуют конструкторы. Однако особенностью деструкторов языка C# является то, что они не занимаются освобождением памяти, выделяемой конструктором соответствующим объектом (за этим следит сборщик мусора), а освобождают ресурсы, выделенные объекту, например, закрывают связанные с ним файлы, связь с сервером базы данных, связь с другим компьютером и т.д.

Также как у конструктора имя деструктора совпадает с именем класса, но перед именем деструктора устанавливается символ «~» – тильда. Например, если бы наш класс treyg содержал деструктор, то его запись выглядела бы следующим образом:

```
public ~treyg()
      { тело_деструктора }
```

Деструктор нельзя вызвать непосредственно в программе — он автоматически вызывается сборщиком мусора при удалении объекта из кучи. Освобождение ресурсов, выделяемых объекту, на практике осуществляется автоматически после того, как надобность в них отпадает, поэтому деструкторы конструктивно присутствуют в структуре класса, но, как правило, не создаются – либо используются по умолчанию.

8.3 Свойства

Один из принципов объектно-ориентированного программирования является инкапсуляция. Формально инкапсуляция это объединение полей и методов класса с целью защиты данных от непосредственного доступа из программы. Поля объекта используются через его интерфейс – совокупность правил доступа или свойства. Скрытие полей объекта – их инкапсуляция (от слова «капсула») осуществляется через свойства.

Свойства представляют собой специальные методы, которые на самом деле состоят из двух методов – get() и set() и некоторого поля объекта. Метод get() возвращает текущее значение соответствующего поля объекта, а метод set() помещает в поле объекта новое значение. Эти методы вызываются автоматически всякий раз, когда программа хочет получить значение «свойства» или изменить его. С точки зрения синтаксиса свойства ничем не отличаются от полей: они могут стоять в левой части оператора присваивания либо являться членом выражения в его правой части. Например, закрытому полю int a можно записать следующее свойство целого типа Aa.

```
   public class treyg
     {
       private int a, b, c, p;
       public int Aa
       {
       get { return a;}
       set { a = value;}
       }
       public string ss;
 .   .   .
```

В программе, после создания объекта t полю a можно присвоить новое значение через свойство Aa следующим образом:

```
int A, B, C;
     treyg t = new treyg();
     A = Convert.ToInt32(textBox1.Text);
     B = Convert.ToInt32(textBox2.Text);
     C = Convert.ToInt32(textBox3.Text);
t.Aa = A; ,
```

или переменной программы A присвоить значение поля a объекта t следующим образом:

```
A = t.Aa; ,
```

где t.Aa – это свойство класса treyg.

В приведенном примере свойством t.Aa мы фактически закрытое поле a класса treyg превратили в открытое поле – это можно сделать проще объявив в классе вместо спецификатора private спецификатор public.

Естественно свойства предназначены для реализации более сложной логики доступа к закрытому полю класса. В нашем примере в свойство

установки нового значения полю класса мы можем включить проверку значения на условие > 0 и на реализацию только одной попытки записи, например:

```
set { if (a == 0 && value > 0) a = value;}
```

Любой из методов доступа свойства (но не оба сразу) может отсутствовать. В этом случае мы получаем свойства, предназначенные только для чтения или только для записи.

8.4 Параметр по ссылке this

Рассмотрим специальное поле указатель (this), которое формируется автоматически при создании объекта и содержит адрес созданного объекта.

Фактически связь полей объекта с методами класса происходит через параметр this. Каждый метод класса может непосредственно обращаться к параметру this для работы с элементами текущего объекта. Поскольку значение this всегда соответствует текущему объекту (объекту, с которым в текущий момент работает программа), то методы класса будут работать с элементами текущего объекта.

Использование параметра this применяется во многих конструкторах для инициализации полей объектов. Многие авторы при описании конструкторов классов в качестве имен формальных параметров используют имена полей классов и для того чтобы различать поля классов и имена формальных параметров перед именами полей классов указывается параметр this. Например, наш конструктор treyg в этом случае необходимо было бы записать следующим образом:

```
public treyg(int a, int b, int c)
      {
       this.a = a; this.b = b; this.c = c;
 }
```

Избежать конфликта имен можно более простым способом – просто выбрав другие имена для формальных параметров (что мы и сделали в программе).

Параметр this нельзя использовать при обращении к статическим элементам класса, так как они принадлежат не конкретному объекту, а классу в целом.

В языке C# применяется еще один параметр по ссылке base, который используется для работы с базовым (родительским) объектом. Если Вы внимательно посмотрите исходный код файла Form1.Designer.cs, а именно метод void Dispose(bool disposing), то в последних строках кода этого метода обычно находится запись base.Dispose(disposing);, которая позволяет при удалении объекта из памяти удалять и родительский (базовый) объект.

8.5 События класса

Важной составной частью ООП является реализованный в классах механизм событий. С помощью этого механизма один объект (источник события) может сообщить другому объекту (получателю события) об изменении своего состояния.

Обычно механизм события используется в многопоточных процессах при синхронизации – упорядочения очередности работы этих потоков. Однако этот же механизм можно использовать в Windows-приложениях, в которых такие элементы, как кнопки, флажки, переключатели и т. п., выдают информацию о взаимодействии с ними пользователя. Например, все объекты - кнопки (класса Button) при щелчке мышью возбуждают событие OnClick. Но для одной кнопки это событие приведет, например, к вводу набранных значений, для другой выполнит некоторое преобразование значений (например, их сортировку), а для третьей откроет окно другой формы.

Каждый элемент управления, размещаемый на форме, имеет определенный набор событий, «пустые» обработчики которых можно получить с помощью окна свойств этого элемента. Однако разработчики программ могут разработать свои специальные обработчики событий. Для реализации механизма события – извещение клиентов некоторого класса о факте наступления события класс - источник события должен:

объявить событие как член класса (наряду с полями, методами, свойствами) — для объявления используется зарезервированное слово event;

передать клиентам класса (получателям события) в нужный момент информацию о наступившем событии, сопроводив ее необходимыми параметрами;

получить от клиента (клиентов) ответ и, проанализировав его, выполнить связанное с событием действие.

Две последних операции (обмен информацией с клиентами) обычно реализуются с помощью делегатов, которые мы будем рассматривать в следующих лекциях.

8.6 Перегрузка операций класса

В некоторых классах используются фрагменты кода (блоки), в заголовках которых используется имя класса (но не конструктор) после которого, до круглых скобок формальных параметров, используется служебное слово operator. С помощью этого слова обозначается механизм перегрузки операций, который позволяет использовать в обычных математических выражениях переменные типа класс. Например, если для класса «студент» перегружена операция сложения (operator+) и в

программе созданы два объекта типа «студент» – ct1 и ct2, то можно записать выражение ct1 + ct2.

Что означает сложение двух объектов? Может быть, мы объединяем их имена или суммируем возраст?

Для этих целей и используется перегрузка операций, которая четко прописывает, что должно суммироваться при суммировании двух объектов, например, необходимо суммировать их оценки.

Служебное слово operator указывает, что осуществляется перегрузка операции, а операция « + », что перегружается операция сложения.

Определение собственных операций класса с помощью указания operator называют перегрузкой операций. Перегрузка обычно применяется для классов, описывающих математические или физические понятия, то есть таких классов, для которых семантика операций делает программу более понятной.

Перегрузка операции класса описываются с помощью методов специального вида (функций - операций).

Формат записи перегрузки операции имеет следующий вид:

```
спецификаторы имя класса operator  имя операции
(формальные параметры) {тело }
```

В качестве спецификаторов обычно одновременно используются ключевые слова public и static. Кроме того, операцию можно объявить как внешнюю (extern).

Тело операции определяет действия, которые выполняются при использовании операции в выражении. Тело представляет собой блок, аналогичный телу других методов класса.

Например:

```
public static int operator+ (Stydent S1, Stydent S2)
            {
                return S1.Ocenka + S2.Ocenka;
        }
```

Если мы определили перегружаемую операцию для « + », то это не означает, что мы определили перегрузку операции «++» или «+=». Это другие операции и их необходимо определять самостоятельно.

Если мы вводим перегружаемую операцию для некоторого класса, то ее действия должны быть ограничены только данными этого класса.

Необходимо отметить, что перегрузка операций не создает новые операции, а только адаптирует существующие к данным типа класс.

При описании перегрузки операций необходимо соблюдать следующие правила:

– операция должна быть описана как открытый статический метод класса (спецификаторы public static);

– формальные параметры в операцию должны передаваться по значению (то есть не должны предваряться ключевыми словами ref или out);

– форматы записей всех операций класса должны различаться.

Перегрузка операций обычно используется с целью обеспечения более естественного синтаксиса выражений для типов, определяемых пользователем.

Пример использования перегрузки операции для подсчета суммы оценок первых двух студентов:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public class Stydent
        {
            int Ocenka;
            string Name;
            public string ss;
            public int Aa
              {
                    get { return Ocenka;}
                    set {if (value>=2 && value <=5) Ocenka =
value; else ss="Вводите правильно значение оценки \r\n";}
              }
         public void Vvod(string name)
         {
           Name = name;
         }
         public static int operator+ (Stydent S1, Stydent S2)
            {
                return S1.Ocenka + S2.Ocenka;
            }
        }
        public Stydent[] styd = new Stydent[5];
        public static int n=0;
        public Form1()
        {
            InitializeComponent();
            textBox1.Text = "";
            n = 0;
        }
        private void button1_Click(object sender, EventArgs e)
        {
            int Oc;
            string fio,In;
            Stydent st = new Stydent();
            fio = textBox2.Text;
```

```
            st.Vvod(fio);
            In = textBox3.Text;
            Oc = Convert.ToInt32(In);
            st.Aa = Oc;
            styd[n] = st;
            textBox1.AppendText("Имя студента " + fio + " Его
оценка = " + styd[n].Aa.ToString() +"\r\n");
            if (st.Aa != 0) n++;
            if (n == 2) textBox1.AppendText("Сумма оценок двух
студентов = " + (styd[0] + styd[1]).ToString() + "\r\n");
        }
    }
}
```

Работа программы изображена на рисунке 8.3.

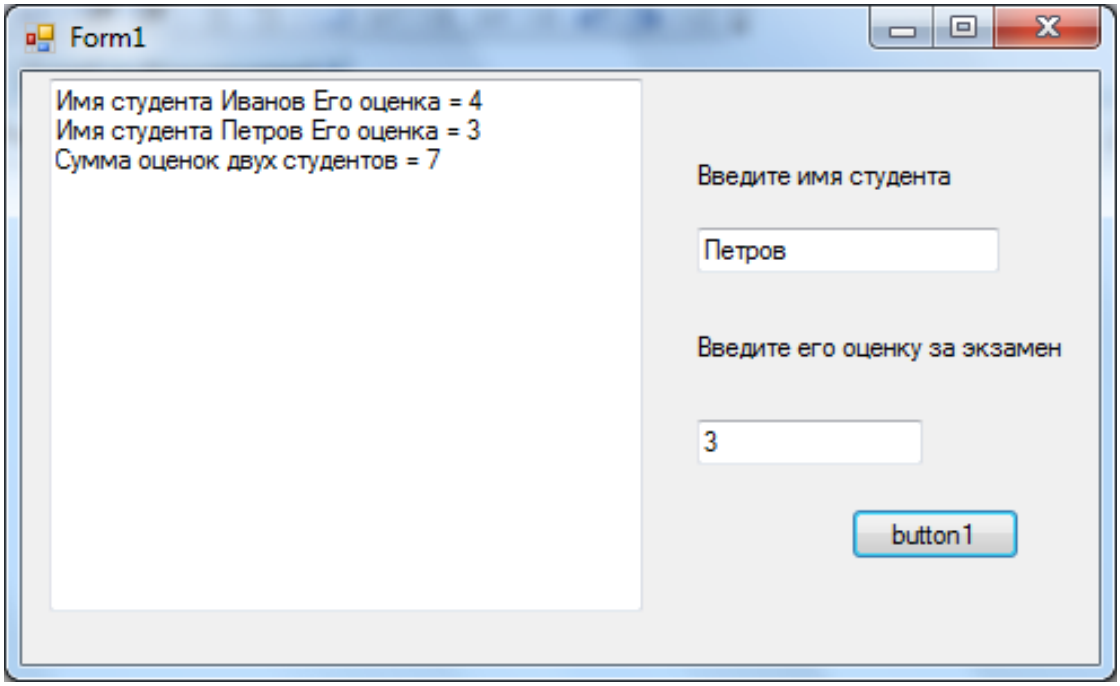


Рисунок 8.3 – Пример перегрузки операции сложения

8.7 Вопросы для самопроверки

1 Для чего предназначен конструктор с умолчанием?

2 Для чего предназначен конструктор с заданием параметров?

3 Как называется процесс определения нескольких методов с одинаковыми именами?

4 Как называются несколько конструкторов одного класса?

5 Когда вызывается деструктор класса tka?

6 Как обычно называется метод, если тип его возвращаемого значения объявлен void?

7 Как должен заканчиваться в языке C# метод если его тип не void?

8 Какие формальные параметры метода C# называются параметры-ссылки?

9 Как называется объединение в одной структуре полей и методов их обработки?

10 Какой механизм используется в классах для доступа к «закрытым» полям класса?

9 ПРИНЦИПЫ ООП

9.1 Понятие инкапсуляции

Технология объектно - ориентированного программирования базируется на трех основных принципах – инкапсуляции, наследовании и полиморфизме.

Принцип инкапсуляции – объединение в одной структуре данных и методов их обработки лежит в основе самой организации класса.

Формально инкапсуляция это объединение полей и методов класса с целью защиты данных от непосредственного доступа из программы.

Поля объекта должны использоваться через его интерфейс – совокупность правил доступа или свойства. Скрытие полей объекта – их инкапсуляция (от слова «капсула») осуществляется через свойства.

Понятие свойства подробно рассмотрено в предыдущей лекции, поэтому ограничимся только рассмотрением примера использования свойства в программе.

Используем класс treyg в качестве учебного примера, для демонстрации роботы со свойствами класса.

Задача 9.1 В классе treyg сторонам треугольника, через свойства, присваиваются случайные целые числа в диапазоне от минус 5 до 5. Свойства выполняют проверку, чтобы вводимые значения были больше 0. После ввода значений сторон треугольника, отдельным методом класса выполняется проверка, что сумма любых двух сторон больше третьей. Каждый шаг работы программы сопровождать комментариями.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public class treyg
  {
   private int a, b, c, p;
   public string ss;
   public treyg()
   {
    a = b = c = 0;
   }
   public int Aa
   {
```

```
     get { return a; }
     set { if (value > 0)  a = value; else a = 0; }
    }
    public int Bb
    {
     get { return b; }
     set { if (value > 0)  b = value; else b = 0; }
    }
    public int Cc
    {
     get { return c; }
     set { if (value > 0)  c = value; else c = 0; }
    }
    public int Pp
    {
     get { return p; }
    }
    public void proverka()
    {
     if (a + b > c && a + c > b && b + c > a)
     {
      p = a + b + c;
      ss = ss + "Периметр треугольника = " + p.ToString();
     }
     else
      MessageBox.Show("Одна из сторон треугольника больше суммы
двух других Повторите ввод ");
    }
   }
   public Form1()
   {
    InitializeComponent();
   }
   private void button1_Click(object sender, EventArgs e)
   {
    Random rnd = new Random();
    int A=1, B=1, C=1;
    bool ok = true;
    treyg t = new treyg();
    while (ok)
    {
     A = rnd.Next() % 11 - 5; t.Aa = A;
     B = rnd.Next() % 11 - 5; t.Bb = B;
     C = rnd.Next() % 11 - 5; t.Cc = C;
     textBox1.Text = Convert.ToString(t.Aa);
     textBox2.Text = Convert.ToString(t.Bb);
     textBox3.Text = Convert.ToString(t.Cc);
     if (A + B + C == t.Aa + t.Bb + t.Cc)
     {
      t.proverka();
      if (t.Pp != 0) ok = false;
     }
     else
```

```
      MessageBox.Show("Одна из сторон треугольника меньше 0!
Повторите ввод ");
    }
    textBox1.Text = Convert.ToString(t.Aa);
    textBox2.Text = Convert.ToString(t.Bb);
    textBox3.Text = Convert.ToString(t.Cc);
    textBox4.Text = t.ss;
   }
  }
}
```

Работа программы представлена на рисунке 9.1.

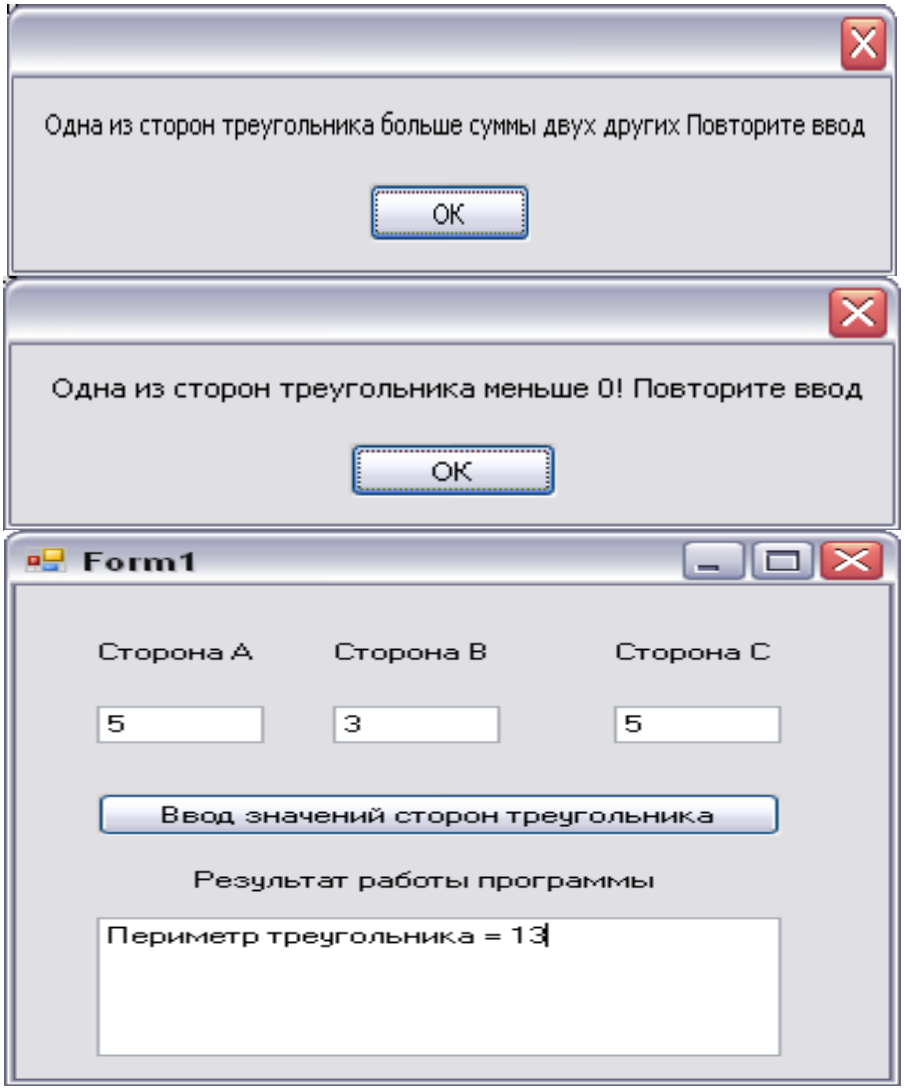


Рисунок 9.1 – Окна работающей программы

При появлении окна MessageBox.Show() одновременно в главном окне программы отображаются значения свойств сторон треугольника.

9.2 Понятие наследования

Принцип наследования является фундаментальным в концепции объектно - ориентированного программирования. Целью наследования является повторное использование уже созданных классов.

Некоторые авторы, объясняя идею наследования классов, приводят пример иерархической связи от общего к частному:

Животные – кошачьи – тигры.

Другие авторы поясняют идею наследования классов на примерах иерархических связей от маленького объекта к большому объекту:

Точка – отрезок – прямоугольник.

При этом достигается единственная цель – полностью “запутывается” читатель в понимании, что может быть базовым – основным классом, а что порожденным - производным классом.

Применение этих двух подходов в изложении материала объясняется разными целями авторов.

Подход «от маленького объекта к большому объекту» позволяет пояснить саму идею наследования – была точка, далее отрезок и т.д. Эта технология применима, когда «с нуля» разрабатывается цепочка наследуемых классов.

Второй подход от общего к частному, точнее к конкретному, применяется в программировании с использованием уже существующих классов, например, VCL в DELPHI, MFC и другие стандартные библиотеки в VISUAL C++ или пространство имен C#.

Процесс программирования в системах визуального программирования включает выделение нужных частей существующих базовых классов с последующим дополнением необходимого кода программы.

Класс, у которого наследуются данные или методы, называется базовым классом.

Класс, который наследует данные или методы у базового класса, называется производным классом.

Наследование представляет собой способность производного класса использовать как свои свойства, данные так и методы базового класса.

Одной из задач программиста сводится к тому, чтобы выбрать нужный класс или часть классов (наследовать их) и дополнить их недостающими фрагментами – адаптировать их для своих задач.

Для пояснения идеи наследования рассмотрим следующий пример. Пусть вам необходимо написать программу, в которой некоторому классу вы решили добавить дополнительные свойства и данные по сравнению с классом предыдущей программы.

Существует два варианта действий.

Первый предполагает копирование класса предыдущей программы в новую программу и внесение в него необходимых изменений.

Второй вариант предполагает создание нового класса, наследующего предыдущий класс.

Из рассмотренных двух вариантов действий по написанию программы, с точки зрения ООП, второй вариант является предпочтительнее, хотя первый вариант иногда проще.

Наследование классов значительно сокращают объёмы программного кода, уменьшается время разработки программы, а так же повышается её надёжность.

Наследование позволяет строить иерархии, в которых производные классы получают в свое распоряжение поля, свойства и методы базовых классов и могут дополнять их или изменять. Таким образом, наследование обеспечивает важную возможность многократного использования кода. Написав и отладив код базового класса, можно, не изменяя его, за счет наследования приспособить класс для работы в различных ситуациях. Это экономит время разработки и повышает надежность программ.

Классы, расположенные ближе к началу иерархии, объединяют в себе общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных особенностей.

Напомню, что базовым классом в иерархической цепочке всех классов C# является класс System.Object.

Итак, наследование применяется для следующих взаимосвязанных целей:

– исключения из программы повторяющихся фрагментов кода;

– упрощения модификации программы;

– упрощения создания новых программ на основе существующих.

Кроме того, наследование является единственной возможностью использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

Формат записи наследования классов включает обычное определение класса, в котором, через двоеточие, добавляется имя базового класса или предка:

```
[ атрибуты ] [ спецификаторы ] class имя_класса [ : базовый класс ]
{  тело класса}
```

Если имя базового класса не указано, то по умолчанию базовым классом является System.Object.

Если у нас есть свой базовый класс, то его необходимо указывать через символ двоеточие, например:

```
public class otr : tka
{ тело класса }
```

В приведенном примере класс отрезок наследует класс точка. Естественно, если данные класса tka закрыты спецификатором доступа privat, то они являются закрытыми и для производного класса.

В некоторых базовых классах данные располагаются после спецификатора доступа protected. Например,

```
protected int x;
protected int y;
```

Необходимость введения спецификатора доступа protected объясняется следующим: производный класс может обращаться к открытым элементам (public) базового класса, как будто они определены в производном классе. С другой стороны производный класс не может обращаться к закрытым элементам (private) базового класса напрямую – для обращения к таким элементам производный класс должен использовать методы базового класса. Поэтому и появился спецификатор доступа protected, который определяет защищенные элементы класса. Защищённые элементы занимают промежуточное место между закрытыми и открытыми элементами. Если элемент является защищённым, объекты производного класса могут обращаться к нему, как будто он является открытым. Для оставшейся части программы защищённые элементы являются закрытыми. И если в программе возникает необходимость обратиться к защищённым элементам, то это можно осуществить только с помощью соответствующих методов класса.

Очень важным вопросом в наследовании классов, является очередность создания объектов при вызове конструктора производного класса.

Так как методы производного класса могут наследовать данные и методы базового класса, то естественно, при создании объекта производного класса все то, что будет наследоваться, уже должно быть.

Поэтому работа конструктора производного класса начинается с вызова конструктора базового класса, по окончании работы которого создается объект производного класса.

Конструктор производного класса должен выполнять инициализацию наследуемых (используемых) элементов данных базового класса.

При вызове деструктора производного класса первым должен удаляться объект производного класса, а затем объект базового класса.

Рассматривая вопросы наследования необходимо отметить, что конструктор и деструктор базового класса не наследуются производным классом.

Задача 9.2 В учебных целях создать цепочку наследуемых классов – точка – отрезок. Предусмотреть просмотр очередности создания объектов классов.

Исходный код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
```

```
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public int ax, ay, bx, by;
  public class tka
  {
   Random rnd = new Random();
   protected int x, y;
   public string ss;
   public tka()
   {
    x = rnd.Next(100);
    y = rnd.Next(100);
   }
   public void gettka(out int ax, out int ay)
   {
    ax = x; ay = y;
   }
   public string printtka()
   {
    return "[" + x.ToString() + "," + y.ToString() + "]";
   }
  }
  public class otr : tka
  {
   public tka b;
   public string s;
   public otr()
   {
    MessageBox.Show("Точка 1 - база отрезока [" + x.ToString()
+ "," + y.ToString() + "]");
    s = "";
   }
   public void getotr(out int ax, out int ay)
   {
    int xx, xy;
    b.gettka(out xx, out xy);
    MessageBox.Show("Точка 2 - поле отрезока [" +
xx.ToString() + "," + xy.ToString() + "]");
    ax = xx; ay = xy;
   }
   public double dlina()
   {
    double dl;
    int k1,k2,k3,k4;
    k1 = x; k2 = y;
    b.gettka(out k3, out k4);
    dl = Math.Sqrt((k1-k3)*(k1-k3)+(k2-k4)*(k2-k4));
    return dl;
```

```
    }
    public void printotr()
    {
     s = " Координата точки поля отрезка = " + b.printtka();
     s = s + " Координата точки базы отрезка = [" +
x.ToString() + "," + y.ToString() + "]";
     s = s + " Длина___отрезка      = " +
dlina().ToString("###.###");
    }
   }
   public Form1()
   {
    InitializeComponent();
   }
   private void button1_Click(object sender, EventArgs e)
   {
    string str;
    otr c = new otr();
    c.gettka(out bx, out by);
    tka bb = new tka();
    c.b = bb;
    c.getotr(out ax, out ay);
    c.printotr();
    str = c.s;
    textBox1.Text = str;
    Invalidate();
   }
   private void Form1_Paint(object sender, PaintEventArgs e)
   {
    Pen myPen = new Pen(Color.Red, 2);
    Graphics g = e.Graphics;
    g.DrawLine(myPen, ax, ay, bx, by);
   }
  }
}
```

Работа программы представлена на рисунках 9.2 и 9.3.

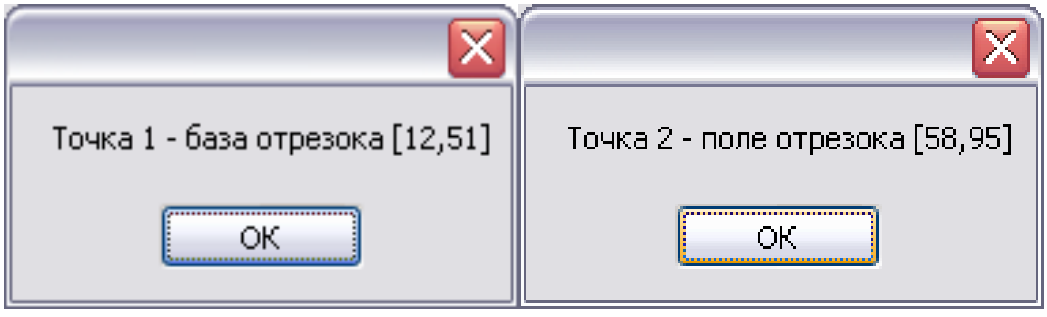


Рисунок 9.2 – Диалоговые окна координат концов отрезка

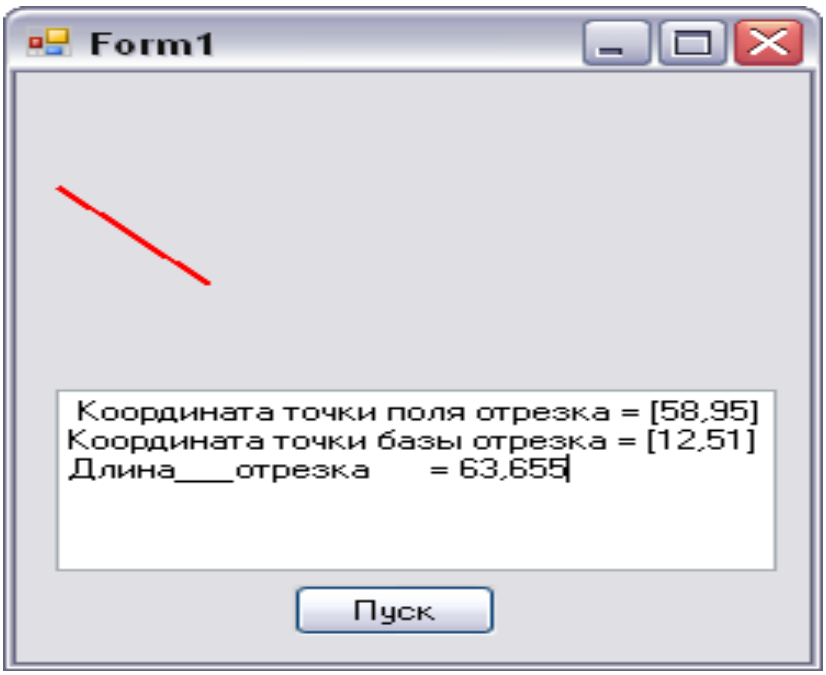


Рисунок 9.3 – Рабочее окно программы

При создании объекта отрезок первоначально формируется объект базового класса отрезка – объект точка (контроль с помощью 1 диалогового окна).

Далее создаем объект точки и присваиваем его полю объекта отрезок (контроль с помощью 2 диалогового окна).

В качестве некоторых действий вычисляем длину отрезка. Все результаты работы выводим с помощью многострочного редактора текстов.

Для гурманов рисуем отрезок в системе координат X,Y (начало системы координат – левый верхний угол окна формы). Обработчик события Form1_Paint формируем с помощью окна Properties – для формы выбираем событие Paint (дважды щелкаем мышкой). Создаем объекты для линий и геометрических фигур и рисуем геометрическую фигуру линию для значений координат отрезка.

Запуск этого обработчика события (перерисовать окно формы) – осуществляем из нашей программы с помощью метода Invalidate().

Этот метод требует от Windows сформировать сообщение WM_PAINT для нашего окна программы. Сообщение поступает в нашу программу (все координаты уже определены) и заставляет «перерисовать» все окно формы нашей программы, добавляя изображение отрезка.

9.3 Вопросы для самопроверки

1 Как один объект может информировать другой объект об изменении своего состояния?

2 Как называется объект, информирующий другой объект об изменении своего состояния?

3 Как формируется «пустой» обработчик некоторого события?

4 Какой обработчик события будет сформирован, если на форме дважды кликнут на кнопку button1?

5 Для чего используется перегрузка операций в классах?

6 Понятие наследования?

7 Что является целью наследования классов?

8 Как называется класс, свойства, данные и методы которого наследуются другим классом?

9 Как называется класс, наследующий свойства, данные и методы базового класса?

10 На чем базируется объектно - ориентированное программирование?

10 ПРИНЦИП ПОЛИМОРФИЗМА

10.1 Понятие полиморфизма

Рассмотренное в предыдущей лекции понятие наследования позволяет использовать методы и данные базовых классов, но при этом различают два вида наследования – статическое и динамическое наследование (статическое и динамическое связывание методов).

Статическое наследование это такое наследование, все связи которого формируются во время компиляции программы и фактически определяются в самой структуре описания классов.

Динамическое наследование, и связанное с ним свойство полиморфизма, предполагают, что некоторые связи формируются в процессе выполнения программы.

Полиморфизм это многообразие форм реализации одноименных методов в цепочке наследуемых классов.

Реализация свойства полиморфизма осуществляется с помощью специальных виртуальных методов и, так называемых, абстрактных базовых классов.

Рассмотрим понятие абстрактных базовых классов.

Для получения преимуществ наследования классов разработчики ООП стали создавать базовые классы, включающих в себя все возможные методы обработки данных определенного множества объектов, но, при этом, базовые классы, как правило, не включали элементы данных.

Например, при создании базового класса «геометрические фигуры» можно включить методы нахождения площади или объема. Естественно, если производным классом является класс «точка» или класс «отрезок», то такие методы для этих объектов лишены смысла.

Базовые классы, для которых создание объектов невозможно или не имеет смысла, стали называть абстрактными базовыми классами. Абстрактные базовые классы служат только для порождения потомков. Как правило, в них задаются только наборы методов, которые каждый из потомков будет реализовывать по-своему. Подобные методы абстрактных базовых классов рассчитаны на несуществующие – виртуальные элементы данных (т.е. на элементы данных будущих классов в цепочке наследования).

Методы, рассчитанные на несуществующие, виртуальные элементы данных будущих классов в цепочке наследования, стали называть виртуальными методами.

Для обозначения виртуальных методов в языке C# используется специальное указание (специальный термин – virtual), означающее, что метод является виртуальным. Например:

```
virtual public double ploc() {return 0.0;}
```

Слово virtual в переводе с английского значит «фактический». Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться по факту его вызова, то есть не на стадии компиляции, а во время выполнения программы. Этот механизм называется поздним или динамическим связыванием методов.

Встретив виртуальные методы, компилятор создаст таблицу виртуальных методов (Virtual Method Table, VMT), в которую поместит названия виртуальных методов и адреса точек входа. Для каждого класса создается одна таблица виртуальных методов.

При этом во время выполнения программы в каждый создаваемый объект дополнительно включается указатель на созданную таблицу VMT.

Вызов виртуального метода выполняется так: из объекта берется адрес его таблицы VMT, из VMT выбирается адрес метода, а затем управление передается этому методу. Таким образом, при использовании виртуальных методов из всех одноименных методов иерархии всегда выбирается тот, который соответствует фактическому типу вызвавшего его объекта.

Если производный класс имеет свою реализацию одноименного виртуального метода, то в нем этот метод должен объявляться как замещающий или перекрывающий метод с атрибутом override. Например,

```
override public double ploc() { . . . }
```

Переопределять виртуальный метод в каждом из производных классов не обязательно. Если он выполняет устраивающие производный класс действия, то метод просто наследуется.

На этапе выполнения программы при обращении любого базового класса к замещенному методу в таблицу виртуальных методов помещается ссылка на соответствующий метод производного класса, который и работает так, как если бы он был изначальной частью родительского класса.

Замещающий виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.

Принцип полиморфизма базируется на «перекрытии» виртуальных методов абстрактного базового класса замещающими методами. При этом каждый производный класс может иметь свою индивидуальную форму реализации наследуемых виртуальных или замещающих методов.

При этом свойство полиморфизма – это возможность для объектов разных классов, связанных наследованием, реагировать различным образом при обращении к одной и той же (по названию) виртуальной функции базового класса.

Полиморфизм, в переводе с греческого языка, означает «много форм», что в данном случае означает «один вызов — много методов».

При описании базовых классов рекомендуется определять в качестве виртуальных те методы, которые в производных классах должны

реализовываться по - другому. Если во всех классах иерархии метод будет выполняться одинаково, его лучше определить как обычный метод.

10.2 Пример статического наследования методов

Разрабатываем цепочку наследуемых классов простейших геометрических фигур с базовыми фигурами точкой и кругом.

В качестве базового возьмем класс, не имеющий полей и содержащий только виртуальную функцию вычисления площади и «чисто виртуальную функцию» печати.

Пример статического наследования – обычное создание объектов, присваивание полям данных (координатам точки или координатам центра круга и радиуса) некоторых случайных значений в диапазоне от 0 до 100 и печать этих значений.

Исходный код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public int x, y, ra;
  public abstract class baseGeo
  {
   public virtual double ploc() {return 0;}
   public abstract string printO();
  }
  public class tka : baseGeo
  {
   protected int x, y;
   public string ss;
   public tka()
   {
    Random rnd = new Random();
    x = rnd.Next(100);
    y = rnd.Next(100);
   }
   public int gettkax() { return x; }
   public int gettkay() { return y; }
   public void settka(int xx,int yy)
   {
    x=xx;
    y=yy;
```

```
    }
    override public string print0()
    {
     return "[" + x.ToString() + "," + y.ToString() + "]";
    }
   }
   public class kryg : tka
   {
    public kryg()
    {
     Random rnd = new Random(10);
     r = rnd.Next(100);
    }
    public void setkryg(int ax,int ay,int rr)
    {
     settka(ax,ay);
     r=rr;
    }
    public void getkryg(out int ax, out int ay, out int rr)
    {
     ax = x;
     ay = y;
     rr = r;
    }
    public int getkrygr() { return r; }
    override public double ploc()   // перекрываем метод
// базового класса
    {
     return 3.14*r*r;
    }
    override public string print0()     // перекрываем метод
// класса tka
    {
     string ss = "";
     ss = "Круг с центром " + base.print0() + "\r\n";
// наследуется функция печати класса tka
     ss = ss + " Радиусом = " + r.ToString() + "\r\n";
     return ss;
    }
    private int r;
   }
   public Form1()
   {
    InitializeComponent();
   }
   private void button1_Click(object sender, EventArgs e)
   {
    int i;
    double pl;
    string s;
    Random rnd = new Random(20);
    x = rnd.Next(100);
    y = rnd.Next(100);
```

```
    //ra = rnd.Next(100);
    // пример статического наследования методов
    tka a = new tka();
    a.settka(x, y);
    s = " Точка с координатами : " + a.printO() + "\r\n";
    textBox1.AppendText(s);
    s = " Площадь объекта = " + a.ploc().ToString() + "\r\n";
    textBox1.AppendText(s);
    kryg c = new kryg();
    x = c.gettkax();
    y = c.gettkay();
    ra = c.getkrygr();
    //c.setkryg(x, y, ra);
    textBox1.AppendText(c.printO());
    s = " Площадь объекта = " + c.ploc().ToString() + "\r\n";
    textBox1.AppendText(s);
    Invalidate();
   }
   private void Form1_Paint_1(object sender, PaintEventArgs e)
   {
    Pen myPen = new Pen(Color.Red, 2);
    Graphics g = e.Graphics;
    g.DrawEllipse(myPen, x, y, ra, ra);
   }
  }
}
```

Работа программы представлена на рисунке 10.1.

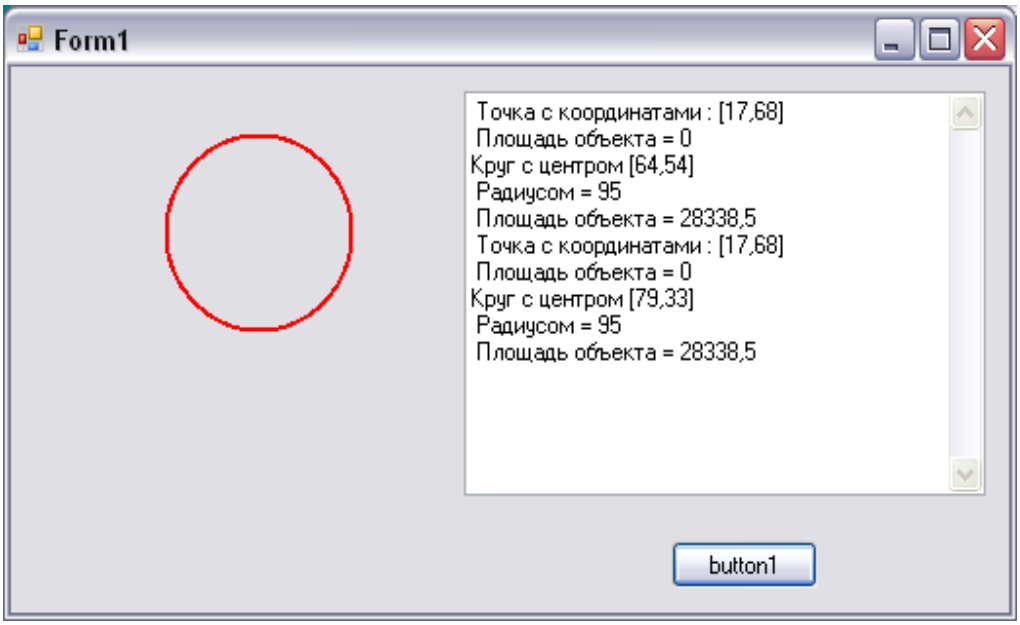


Рисунок 10.1 – Статическое наследование методов

В приведенном примере статического наследования методов присутствует механизм перекрытия методов, но связи между методами фактически определяются во время компиляции программы.

10.3 Пример динамического наследования методов

В качестве примера динамического наследования методов используем ту же цепочку наследуемых классов, но, во время работы программы, поместим 5 созданных объектов в стек.

Исходный код программы:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{   public partial class Form1 : Form
 {
  public int x, y, ra;
  public int[,] masi = new int[6,3];
  public Stack vstek = new Stack();
  public abstract class baseGeo
  {
   public virtual double ploc() { return 0; }
   public abstract string printO();
  }
  public class tka : baseGeo
  {
   protected int x, y;
   public tka()
   {
    Random rnd = new Random();
    x = (int)Math.Round(rnd.NextDouble() * 100);
    y = (int)Math.Round(rnd.NextDouble() * 100);
   }
   public int gettkax() { return x; }
   public int gettkay() { return y; }
   public void settka(int xx, int yy)
   {
    x = xx;
    y = yy;
   }
   override public string printO()
   {
    return "Точка [" + x.ToString() + "," + y.ToString() +
"]";
   }
  }
  public class kryg : tka
  {
   public kryg()
```

```
 {
  Random rnd = new Random(10);
  r = (int)Math.Round(rnd.NextDouble() * 100);
 }
 public void setkryg(int ax, int ay, int rr)
 {
  settka(ax, ay);
  r = rr;
 }
 public void getkryg(out int ax, out int ay, out int rr)
 {
  ax = x;
  ay = y;
  rr = r;
 }
 public int getkrygr() { return r; }
 override public double ploc()        // перекрываем метод
       // базового класса
 {
  return 3.14 * r * r;
 }
 override public string printO()     // перекрываем метод
            // класса tka
 {
  string ss = "";
  ss = "Круг с центром: " + base.printO() + "\r\n";
  // наследуется функция печати класса tka
  ss = ss + " Радиусом = " + r.ToString() + "\r\n";
  return ss;
 }
 private int r;
}
static void vkl(Stack vst, baseGeo n)
{
 vst.Push(n);
}
public Form1()
{
 InitializeComponent();
}
private void button1_Click(object sender, EventArgs e)
{
 int i,k;
 double pl;
 string s;
 Stack vstek = new Stack();
 Random rnd = new Random(20);
 // пример динамического наследования методов
 for (i = 1; i <= 5; i++)
 {
  x = (int)Math.Round(rnd.NextDouble() * 100);
  y = (int)Math.Round(rnd.NextDouble() * 100);
  k = (int)Math.Round(rnd.NextDouble() * 100);
```

```
    if (k % 2 == 0) k = 0; else k = 1;
    if (k == 0)
    {
     tka a = new tka();
     a.settka(x, y);
     s = a.printO() + "\r\n";
     textBox1.AppendText(s);
     s = " Площадь объекта = " + a.ploc().ToString() + "\r\n";
     textBox1.AppendText(s);
     masi[i, 0] = a.gettkax();
     masi[i, 1] = a.gettkay();
     masi[i, 2] = 0;
     vkl(vstek, a);
    }
    else
    {
     kryg c = new kryg();
     x = c.gettkax();
     y = c.gettkay();
     ra = c.getkrygr();
     textBox1.AppendText(c.printO());
     s = " Площадь объекта = " + c.ploc().ToString() + "\r\n";
     textBox1.AppendText(s);
     c.getkryg(out masi[i,0], out masi[i,1], out masi[i,2]);
     vkl(vstek, c);
    }
   }
   s = "Печать содержимого стека: \r\n";
   foreach (baseGeo T in vstek)
   s = s + T.printO() + "\r\n";
   textBox1.AppendText(s);
   Invalidate();
  }
  private void Form1_Paint(object sender, PaintEventArgs e)
  {
   Pen myPen = new Pen(Color.Red, 2);
   Graphics g = e.Graphics;
   for (int i = 1; i <= 5; i++)
   {
    if (masi[i,2]==0)
    {
     x = masi[i, 0]; y = masi[i, 1];
     g.DrawEllipse(myPen, x, y, 2, 2);
    }
    else
    {
     x = masi[i, 0]; y = masi[i, 1]; ra = masi[i, 2];
     g.DrawEllipse(myPen, x, y, ra, ra);
    }
   }
  }
 }
}
```

Работа программы представлена на рисунке 10.2.

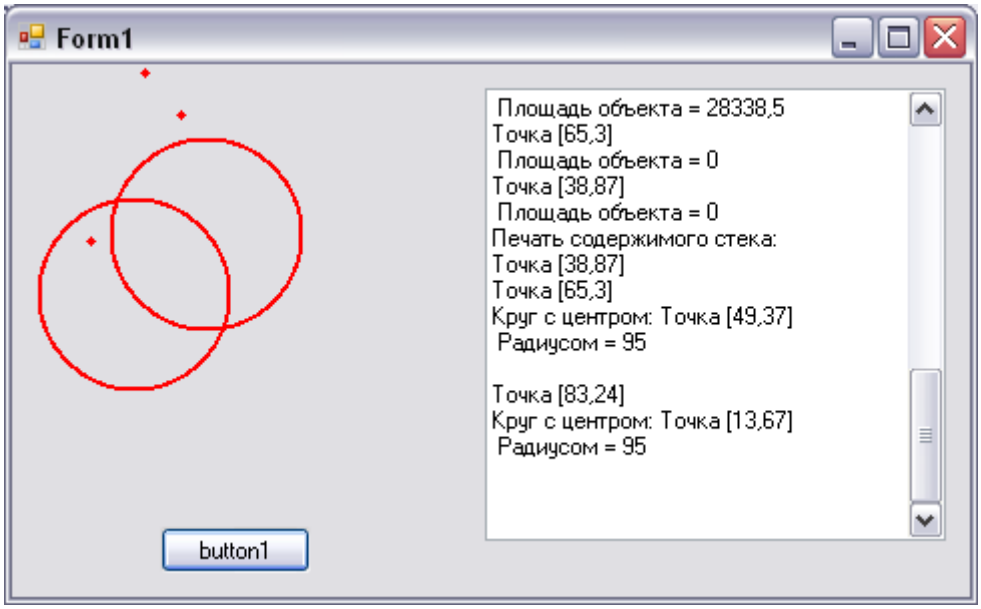


Рисунок 10.2 – Динамическое наследование методов

Печать содержимого элементов стека является примером полиморфизма (динамического наследования методов). Действительно, «статически» печать элемента стека рассчитана на печать элемента простейшей геометрической фигуры. Но в процессе работы программы (в этом суть динамического наследования методов) в стек помещаются объекты классов точка и круг.

10.4 Вопросы для самопроверки

1 Какой класс является базовым классом в иерархической цепочке всех классов языка C#?

2 Какой объект базового или производного классов создается раньше при вызове конструктора производного класса?

3 В чем преимущество цепочек наследуемых классов?

4 Что означает статическое наследование в языке C#?

5 Что означает динамическое наследование в языке C#?

6 Понятие полиморфизма в языке C#?

7 Как реализуется механизм полиморфизма?

8 Как называется метод, объект для которого определяется во время работы программы?

9 Какой класс в языке C# называется абстрактным базовым классом?

10 Что представляет собой таблица виртуальных методов?

11 ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСОВ

11.1 Понятие интерфейса

Слово "интерфейс" - многозначное, и в разных контекстах оно имеет различный смысл. Существует понятие программного или аппаратного интерфейса, но в большинстве случаев слово интерфейс ассоциируется с некоторой связью между объектами или процессами. В данной лекции речь идет о понятии интерфейса, стоящем за ключевым словом interface. В таком понимании интерфейс - это частный случай класса.

Интерфейс представляет собой полностью абстрактный класс, все методы которого абстрактны.

От абстрактного класса интерфейс отличается некоторыми деталями в синтаксисе и поведении.

Синтаксическое отличие состоит в том, что методы интерфейса объявляются без указания модификатора доступа.

Отличие в поведении заключается в более жестких требованиях к потомкам. Класс, наследующий интерфейс (интерфейсный класс), обязан полностью реализовать все методы интерфейса. В этом отличие от класса, наследующего абстрактный класс, где потомок может реализовать лишь некоторые методы родительского абстрактного класса, оставаясь абстрактным классом.

Важное отличие интерфейсного класса от обычного класса заключается в том, что он может наследовать несколько родительских интерфейсов. Таким образом, в C# разрешено множественное наследование, но только в интерфейсных классах.

Родительские интерфейсы перечисляются в списке за именем класса и двоеточием:

```
public interface INewClass: IInt1, IInt2, ..., IIntN
{ . . .}
```

Такого рода интерфейсные классы обязаны содержать реализации всех методов всех родительских интерфейсов.

Замечу, что интерфейсный класс может наследовать не только от интерфейсов, но и от одного (и только одного!) обычного класса, по отношению к которому он ведет себя как обычный наследник, то есть может переопределять его методы, добавлять поля и т. д.

Множественное наследие потенциально связано с возможностью конфликта имен и наличием общего родителя. Конфликт имен проявляется в том, что разные родительские интерфейсы могут содержать одноименные методы с одинаковым синтаксисом.

Поскольку интерфейсный класс обязан реализовывать все методы своих родительских интерфейсов, возникает коллизия, которую можно разрешить одним из следующих способов.

Склеивание методов. В этом случае интерфейсный класс полагает, что у всех одноименных методов должна быть одинаковая программная реализация, и объявляет этот единственный метод для реализации всех одноименных методов своих родителей.

Переименование методов. Если реализация одноименных методов должна быть различной, методы переименовываются.

Отметим еще одно важное назначение интерфейсов, отличающее их от абстрактных классов. Абстрактный класс представляет собой начальный этап проектирования класса, который в будущем получит конкретную реализацию. Интерфейсы задают дополнительные свойства классу. Каждый интерфейс наделяет класс тем или иным новым свойством.

11.2 Синтаксис интерфейса

Общее описание интерфейса, включающее необязательные элементы (они выделены квадратными скобками), имеет следующий формат записи:

```
[ атрибуты ] [ спецификаторы ]   interface имя_класса
[ : родители ]
   { тело_класса  }   ,
```

где

атрибуты – задают дополнительную информацию о классе;

спецификаторы – обычно определяют условие доступа к составляющим класса;

родители – родительские интерфейсные классы, которые наследует наш класс:

тело класса – определяет состав интерфейсного класса.

Если внимательно посмотреть на формат записи интерфейса, то можно заметить, что его формат очень похож на формат записи обычного класса. Это объясняется тем, что интерфейс – частный случай класса.

В библиотеке платформы .NET имеется большое число интерфейсов, наследуя которые, классы получают дополнительные свойства.

Например, интерфейс IComparable задает метод сравнения объектов по принципу больше или меньше, что позволяет выполнять их сортировку.

Реализация интерфейсов IEnumerable и IEnumerator дает возможность просматривать (перебирать) содержимое объекта с помощью конструкции foreach, а реализация интерфейса ICloneable — клонировать объекты.

Каждый интерфейс наделяет класс теми или иными новыми возможностями. В этом смысле поле для разработки новых интерфейсов практически бесконечно.

Например, можно разработать интерфейс для продажи – покупки валюты в соответствии с текущим курсом, интерфейсы для различных начислений коммунальных услуг с учетом льгот и т.д.

В качестве учебного примера опишем интерфейс, реализация методов которого позволит классу проводить некоторые преобразования над музыкальной записью – преобразуя 7 нот и паузу в цифры от 0 до 7 и выполнять обратные преобразования.

Как и любой учебный пример, он немного искусственный, поскольку наша главная задача сейчас состоит в том, чтобы рассмотреть технологию создания и использования интерфейсов.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  interface ITextNoti
  {
   string Codirovanie();
   string Decodirovanie();
  }
    class MyzikText: ITextNoti
  {
   string text;
   static string[] codeTable =
     {
        "до", "ре", "ми", "фа", "соль", "ля", "си", "пауза"
     };
     //Конструктор
   public MyzikText(string txt)
   {
    text = txt;
   }
   //Реализация интерфейсов
   public string Codirovanie()
   {
    Boolean ok;
    string rez = "";
    string[] noti;
    // преобразование к нижнему регистру
    string text1 = text.ToLower();
    //размерность массивов noti устанавливается
    // автоматически в соответствии с размерностью массива,
    //возвращаемого методом Split
    noti = text1.Split(' ');
```

```
    for (int i = 0; i < noti.Length; i++)
    {
     ok = false;
     for (int j = 0; j < 8; j++)
     if (noti[i] == codeTable[j])
        { ok = true; rez = rez + " " + j.ToString(); }
     if (ok == false) rez = rez + " ?";
    }
    return rez;
   }
   // дешифровка поля text
   // с использованием таблицы нот
   public string Decodirovanie()
   {
    Boolean ok;
    string rez = "";
    string[] noti;
    // преобразование к нижнему регистру
    string text1 = text.ToLower();
    noti = text1.Split(' ');
    for (int i = 0; i < noti.Length; i++)
    {
     ok = false;
     for (int j = 0; j < 8; j++)
      if (Convert.ToInt32(noti[i]) == j)
         { ok = true; rez = rez + " " + codeTable[j]; }
     if (ok == false) rez = rez + " ?";
    }
    return rez;
   }
  }
  public Form1()
  {
   InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e)
  {
   string a,b;
   a = textBox1.Text;
   MyzikText IcxodText = new MyzikText(a);
   b = IcxodText.Codirovanie();
   textBox3.AppendText(b + "\r\n");
  }
  private void button2_Click(object sender, EventArgs e)
  {
   string a, b;
   a = textBox2.Text;
   MyzikText IcxodText1 = new MyzikText(a);
   b = IcxodText1.Decodirovanie();
   textBox3.AppendText(b + "\r\n");
  }
 }
}
```

Объявляем интерфейс ItextNoti, содержащий два метода – кодирование (словесный текст нот заменяется цифрами от 0 до 7) и декодирование (текст, представленный цифрами от 0 до 7, заменяется словесным текстом нот).

```
interface ITextNoti
     {
          string Codirovanie();
          string Decodirovanie();
     }
```

Далее объявляем интерфейсный класс, наследующий интерфейс и реализующий его методы. Выполняем общедоступную реализацию методов интерфейса. Алгоритм реализации интерфейсных методов прокомментирован в коде программы и не нуждается в дополнительных пояснениях.

Работа программы изображена на рисунке 11.1.

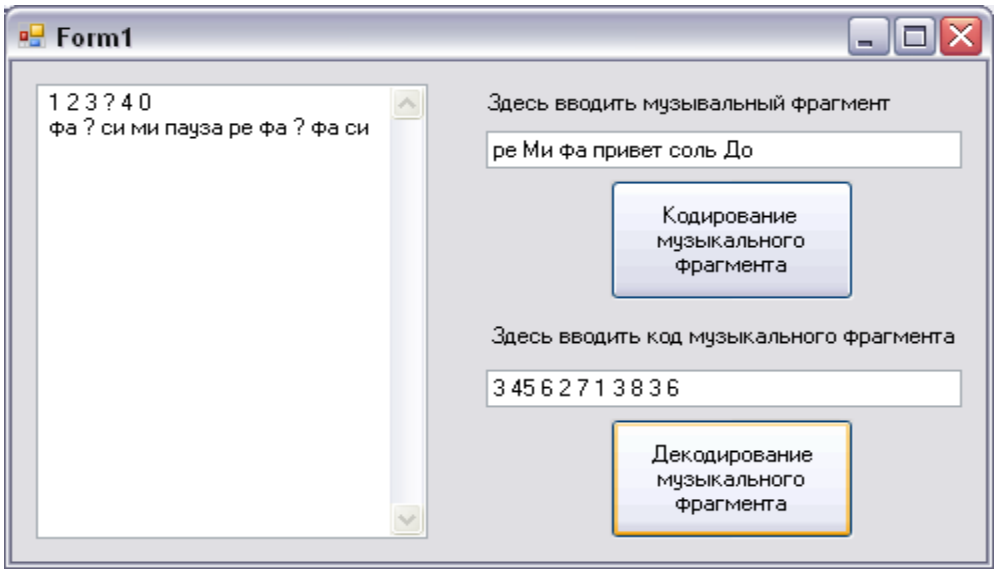


Рисунок 11.1 – Использование интерфейсного класса

В приведенном примере показана технология создания и использование интерфейса и интерфейсного класса.

11.3 Использование стандартного интерфейса IEnumerable

На первый взгляд преимуществ во введении интерфейсного класса нет – методы кодирования и декодирования можно разместить непосредственно в классе MyzikText.

Реально существующие в библиотеке платформы .NET классы включают большое число интерфейсных методов различных интерфейсов, наследуя которые, классы получают дополнительные свойства – через название методов, а не через их реализацию. Реализацию интерфейсных

методов каждый интерфейсный класс, как правило, должен выполнять самостоятельно. Например, если в нашем классе необходимо организовать просмотр с помощью цикла foreach некоторых перечисляемых объектов представленных массивом, то наш класс должен быть наследником интерфейса IEnumerable (перечислимый). У этого интерфейса всего один метод GetEnumerator(), возвращающий объект типа Enumerator (перечислитель). Формат записи метода GetEnumerator() имеет следующий вид:

```
IEnumerator GetEnumerator();
```

Таким образом, наш класс должен быть наследником интерфейсов IEnumerable и IEnumerator.

Интерфейс IEnumerator включает одно свойство Object Current{get;}, возвращающее очередной перечисляемый объект, и два метода – bool MoveNext(), передвигающий перечислитель на следующий перечисляемый объект, и метод void Reset(), устанавливающий перечислитель на первый перечисляемый объект.

В совокупности именно указанное свойство и эти два метода позволяют организовать процесс просмотра объектов массивов с помощью цикла foreach. Методы этих интерфейсов работают с виртуальной коллекцией (набором объектов определяемых в процессе работы программы), что и определяет их универсальность.

Если в классе необходимо выполнять сравнение объектов, например, при их сортировке, то такой класс следует объявить наследником интерфейса IComparable. Этот интерфейс имеет всего один метод CompareTo(object obj), возвращающий целочисленное значение, положительное, отрицательное или равное нулю, в зависимости от выполнения отношения "больше", "меньше" или "равно".

Рассмотрим работу с интерфейсами IEnumerable и IEnumerator на учебном примере, в котором необходимо организовать просмотр товаров некоторого магазина. Для простоты считаем, что класс Tovar имеет два поля название товара и его цена.

```
class Tovar
        {
            public string Naz; // Название и цена товара
            public int Cena;
            public Tovar(string n, int c) // Конструкор товара
            {
                Naz = n;    Cena = c;
            }
        }
```

Для хранения объектов типа Tovar используем класс Cklad, имеющий следующую структуру:

```
        class Cklad
         {
            public Tovar[] tovar;    // Массив товаров
```

```
            public Cklad()  // Конструктор склада
            {
               tovar = new Tovar[4];
            }
        }
```

Максимальное количество объектов, которое может храниться на складе, в учебных целях принято равным 4.

Рассмотрим работу программы без использования интерфейса IEnumerable**.**

Код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public static string s;
  public static int kol;
  class Tovar
  {
   public string Naz; // Название и цена товара
   public int Cena;
   public Tovar(string n, int c) // Конструкор товара
   {
    Naz = n;     Cena = c;
   }
  }
  class Cklad
  {
   public Tovar[] tovar;    // Массив товаров
   public Cklad()  // Конструктор склада
   {
    tovar = new Tovar[4];
   }
  }
  public Form1()
  {
   InitializeComponent();
   kol = 0;
   s = "";
  }
  Cklad ckl = new Cklad();
  private void button2_Click(object sender, EventArgs e)
  {
   if (kol < 4)
```

```
    {
ckl.tovar[kol] = new Tovar(textBox1.Text,
Convert.ToInt32(textBox2.Text));
      s = s + textBox1.Text + textBox2.Text + "\r\n";
    }
    else { s = s + "CKLAD POLHIJ" + "\r\n"; kol--; }
    kol++;
    textBox3.Text = s;
   }
   private void button1_Click(object sender, EventArgs e)
   {
    s = "";
    s = "Работает цикл foreach"+ "\r\n";
    foreach (Tovar t in ckl.tovar)
    {
     s = s + t.Naz + "  " + t.Cena.ToString() + "\r\n";
    }
    s = s + "Работает цикл for" + "\r\n";
    for (int i = 0; i < kol; i++)
    {
s = s + ckl.tovar[i].Naz + "  " + ckl.tovar[i].Cena.ToString()
+ "\r\n";
    }
    textBox3.Text = s;
   }
  }
}
```

Работа программы изображена на рисунке 11.2.

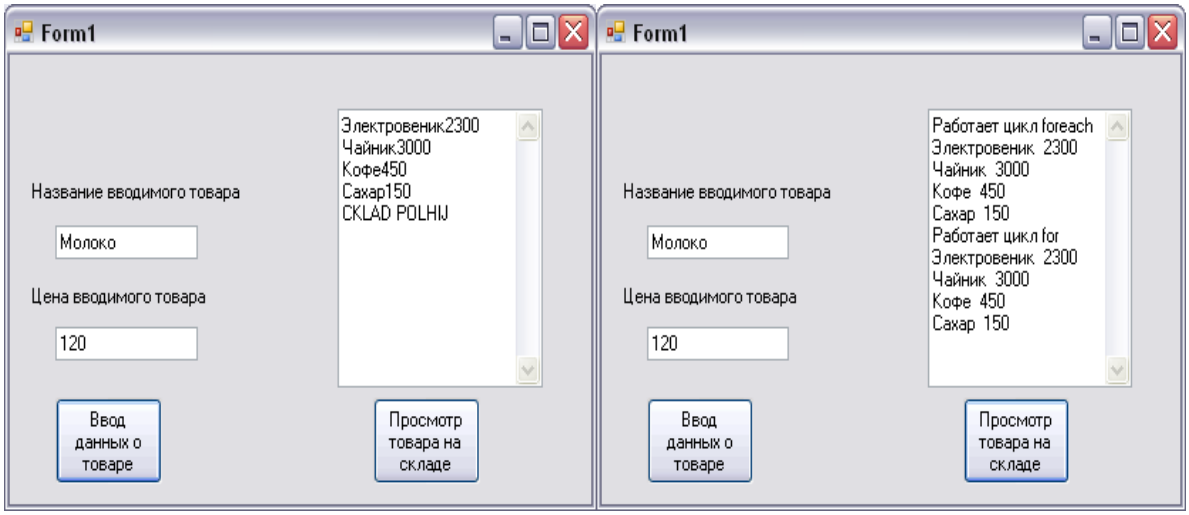


Рисунок 11.2 – Работа программы без интерфейсов

Необходимо отметить, что в программе цикл foreach используется только для переменной типа массив ckl.tovar, который уже имеет встроенный интерфейс – все классы массивы, независимо от типа элементов, реализуют перечисление элементов массива, и для них определен метод GetEnumerator.

Однако, если мы попытаемся использовать цикл foreach для объектов класса Tovar в объекте ckl типа Cklad, а не для массива tovar объекта класса Cklad, например,

```
foreach (Tovar t in ckl)
     {
       s = s + t.Naz + "  " + t.Cena.ToString() + "\r\n";
 } ,
```

то программа выдаст сообщение об ошибке:

«foreach statement cannot operate on variables of type ' WindowsFormsApplication1.Form1.Cklad ' because 'Books' does not contain a public definition for 'GetEnumerator'»
(Оператор **foreach** не может применяться к переменным типа ' WindowsFormsApplication1.Form1.Cklad ' , так как переменные этого класса не содержат открытого определения метода **'GetEnumerator'**).

Доопределим нашу программу необходимым интерфейсом, для этого в нашей программе необходимо добавить дополнительное пространство имен:

```
using System.Collections;
```

Класс Cklad должен наследовать интерфейс IEnumerable:

```
class Cklad : IEnumerable
```

В тело класса Cklad необходимо включить реализацию метода GetEnumerator:

```
public IEnumerator GetEnumerator()
     {
      for (int i = 0; i < 4; i++) yield return tovar[i];
}
```

Необходимы некоторые комментарии, которые взяты из книги Т.А.Павловской [2].

«Таким образом, если требуется, чтобы для перебора элементов класса мог применяться цикл `foreach`, необходимо реализовать четыре метода: GetEnumerator, Current, MoveNext и Reset. Например, если внутренние элементы класса организованы в массив, потребуется описать закрытое поле класса, хранящее текущий индекс в массиве, в методе MoveNext задавать изменение этого индекса на 1 с проверкой выхода за границу массива, в методе Current – возврат элемента массива по текущему индексу и т.д.

Это не интересная работа, а выполнять ее приходится часто, поэтому в версии 2.0 были введены средства, облегчающие выполнение перебора в объекте – итераторы.

Итератор представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла `foreach` выполняется один шаг итератора, заканчивающийся

выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова yield**.**

. . .

Все, что требуется сделать в версии 2.0 для поддержки перебора — указать, что класс реализует интерфейс `IEnumerable`, и описать итератор. Доступ к нему может быть осуществлен через методы `MoveNext` и `Current` интерфейса `IEnumerator`. За кодом, приведенным в листинге итератора, стоит большая внутренняя работа компилятора.

На каждом шаге цикла `foreach` для итератора создается «оболочка» — служебный объект, который запоминает текущее состояние итератора и выполняет все необходимое для доступа к просматриваемым элементам объекта. Иными словами, код, составляющий итератор, не выполняется так, как он выглядит — в виде непрерывной последовательности, а разбит на отдельные итерации, между которыми состояние итератора сохраняется.»

Приведенные два небольших изменения в программе позволяют использовать цикл foreach, который ранее выдавал сообщение об ошибке.

Необходимо отметить особенность работы цикла foreach, которая может приводить к «зависанию» программ. Если в нашей программе, после ввода нескольких объектов товара, но не до полного заполнения массива, мы включим режим просмотра товаров на складе, то программа «повиснет» при попытке вывода несуществующих значений – необходимо контролировать «перебираемые» значения цикла foreach**.**

Цикл for не имеет этих недостатков, потому что его конечное значение определяется текущим значением глобальной переменной kol.

```
for (int i = 0; i < kol; i++)
{
  s = s + ckl.tovar[i].Naz + "  " +
  ckl.tovar[i].Cena.ToString() + "\r\n";
}
```

11.4 Вопросы для самопроверки

1 Понятие интерфейса?

2 В чем отличие наследования интерфейсного класса от обычного наследования?

3 В чем отличие интерфейсного класса от обычного класса?

4 Почему в интерфейсных класса возможно появление конфликтов в названиях имен?

5 Как можно решать конфликт имен при множественном наследовании?

6 В чем суть склеивания методов?

7 В чем суть переименования методов?

8 В чем преимущество интерфейсов?

9 Какой тип могут иметь интерфейсы?

10 Какое служебное слово используется при объявлении интерфейса?

12 КОМПОЗИЦИЯ И КОЛЛЕКЦИЯ КЛАССОВ

12.1 Понятие композиции и коллекции класса

Если некоторый класс, в своих полях данных, использует объекты другого класса, то такое объединение классов называется композицией.

Композиция классов это один из способов повторного использования ранее написанных фрагментов программ. Например, класс «аптека» можно рассматривать как композицию класса «лекарство» – массив различных объектов одного класса «лекарство». Примером композиции является объединение объектов «автомобиль» в классе «гараж» и т.д.

Эти примеры показывают, что композиции классов соответствует множество отношений из реальной жизни.

Объединение однотипных объектов в одной структуре данных называется коллекцией.

Коллекция должна обеспечивать многие функции обработки объектов - сохранять и удалять объекты, предоставлять операции доступа по обновлению и добавлению объектов и т.д.

Обычно коллекция представлена отдельным классом.

Класс, описывающий всю коллекцию, называется классом коллекции.

В приведенных примерах композиции классов второй класс часто выступает в роли коллекции объектов первого класса. Например, класс «гараж» представляет собой коллекцию объектов класса «автомобиль». Класс «аптека» – коллекцию объектов класса «лекарство».

Все классы коллекций условно можно разделить на линейные и нелинейные коллекции.

Линейные коллекции образуют следующие классы коллекций:

– с индексированным доступом, например, различные словари и справочники (телефонный справочник, в котором поиск выполняется по буквам фамилии абонента);

– с прямым доступом, например, массивы. Многие «списки» представлены динамическими массивами;

– с последовательным доступом, например, стеки, очереди, списки.

Нелинейные коллекции включают следующие классы коллекций:

– иерархические, например, различные древовидные структуры. Иерархические системы классификации - УДК или организация поисковых массивов документов в некоторых поисковых системах;

– групповые, например, различные наборы, сетевые структуры, графы.

Еще раз отметим, что композиция классов является мощным инструментом программирования при использовании ранее разработанных классов или даже фрагментов программ.

В среде программирования C# существует множество различных классов, предназначенных для коллекционирования однотипных объектов других классов, например, различные списки, стеки, очереди, словари, деревья и многие другие коллекции.

12.2 Пример использования композиции и коллекции класса

Из всего разнообразия классов коллекций рассмотрим самую простую коллекцию классов – стек.

Задача 12.1 Пусть элементом стека является объект класса КНИГА. Предполагается, что книги складываются «стопкой» и брать и добавлять книги можно только сверху.

Для простоты будем считать, что все данные класса открыты и ограничены только автором книги, ее названием и ценой. Из методов класса используем только конструктор с заданием параметров.

Для организации коллекции в виде списочной структуры используем стандартную структуру Stack.

Исходный код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        public class Kniga
        {
            public string Naz;
            public string Avtor;
            public int Ctoimoct;
            public Kniga(string sa, string sb, int sc)
            {
                Avtor = sa; Naz = sb; Ctoimoct = sc;
            }
        };
        public Stack<Kniga> vstek = new Stack<Kniga>();
        public string ss = "";
        private void button1_Click(object sender, EventArgs e)
        {
```

```
            string a, b;
            a = textBox1.Text;
            b = textBox2.Text;
            c = Convert.ToInt32(textBox3.Text);
            Kniga Tom = new Kniga(a, b, c);
            vstek.Push(Tom);
        }
        private void button2_Click(object sender, EventArgs e)
        {
            Kniga Tom = new Kniga("", "", 0);
            foreach (Kniga T in vstek )
            {
              ss = T.Avtor + "  " + T.Naz + "  " +
              Convert.ToString(T.Ctoimoct) + " \n";
              textBox4.AppendText(ss);
            }
        }
    }
}
```

Работа программы изображена на рисунке 12.1

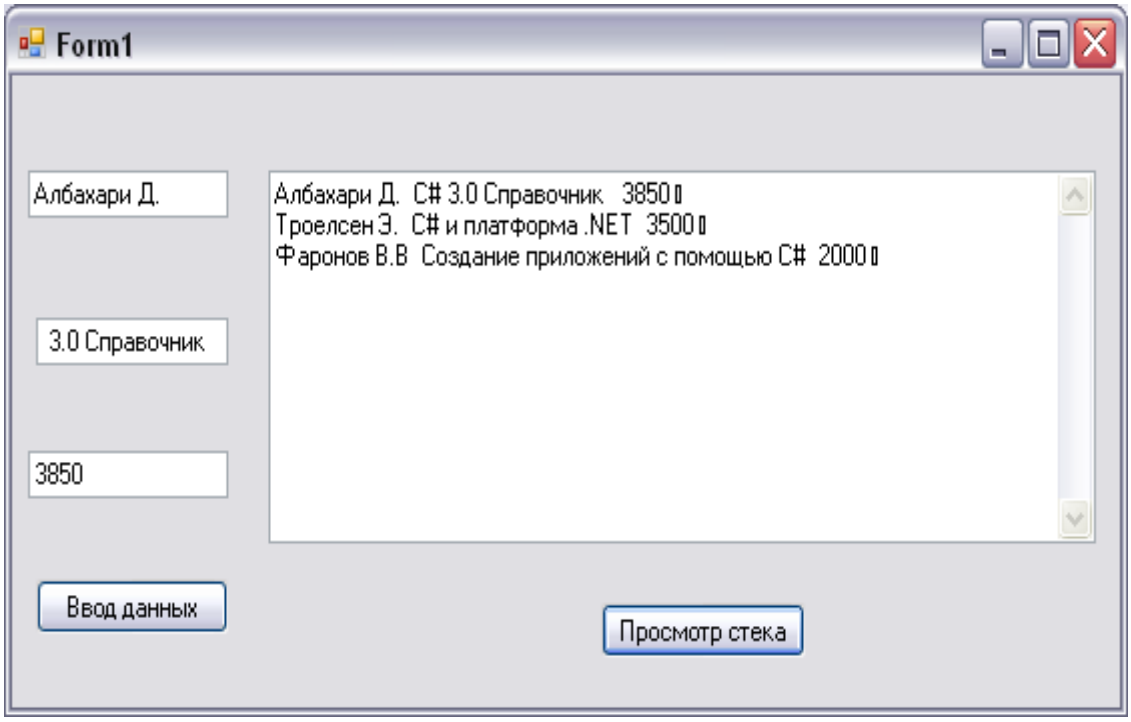


Рисунок 12.1 – Работа программы

Необходимо отметить, что в языке C# имеется несколько классов, реализующих коллекцию объектов, например, массива, стеки и очереди и т.д. Считается, что только массивы являются конструкцией языка C#, а остальные коллекции – это просто классы платформы .NET (точнее ее библиотеки Framework). Поэтому, имеет смысл, ознакомится с составом коллекций этой библиотеки.

12.3 Некоторые коллекции Framework

Реализация классов коллекций осуществляется с помощью интерфейсов. Платформа .NET, а именно библиотека Framework, предоставляет разработчика коллекций дополнительные интерфейсы и их обобщенные версии для организации коллекций, доступа к ее элементам по индексам, поиск в коллекции и т.д.

Интерфейс ICollection являются стандартным для счетных коллекций объектов. Он допускает перебор своих элементов с использованием интерфейсов IEnumerable и IEnumerator, позволяет определять размер коллекции, копировать ее в массив для более сложной обработки и т.д.

Интерфейс IList является стандартным для создания коллекций типа массив. Он также допускает перебор своих элементов с использованием интерфейсов IEnumerable и IEnumerator, но и обеспечивает возможность прямого доступа к элементу с помощью перегружаемого индексатора. Интерфейс IList обеспечивает добавление, удаление и редактирование элемента коллекции по его индексу.

Существует множество интерфейсов для работы с нестандартными коллекциями, например, интерфейс IDictioaryEnumerator позволяет работать с коллекциями типа словари и т.д.

Использование интерфейсов позволило в библиотеке Framework создать несколько классов коллекций, например, Array**,** ArrayList**,** LinkedList**,** Queue**,** Stack и другие.

Для визуального представления коллекций (фактически это тоже классы коллекций) в языке C# имеется несколько управляющих элементов, например, известный нам DataGridView и другие подобные ему элементы, TreeView – предназначен для отображения иерархических коллекций и т.д.

Более подробно Вы будете изучать коллекции в дисциплинах «Прикладное программирование» и «Базы данных» в нашей дисциплине мы рассмотрим работу с классом коллекций на примере коллекции ArrayList**.**

12.4 Коллекция ArrayList

Класс ArrayList принадлежит пространству имен System**.**Collections**.**ArrayList и предназначен для хранения объектов произвольного типа. Основные свойства и методы класса ArrayList приведены в таблице 12.1

Таблица 12.1 – Основные свойства и методы класса ArrayList

Свойства и методы	Описание
public static ArrayList (IList: List);	На основе коллекции List создает объект ArrayList
public virtual int Add(Oblect Value);	Добавляет в конец списка новый объект и возвращает его индекс
public virtual viod AddRange (ICollection coll)l	Добавляет в конец списка несколько объектов
public virtual int BinsrySearch(Qbject Value);	Отыскивает объект Value в отсортированном списке и возвращает его индекс или отрицательно число, если объект не найден
public virtuai int Capacity {get;}	Это свойство, предназначенное только для чтения, хранит текущую емкость коллекции
public virtual void Clean();	Удаляет все элементы коллекции
public virtual bool Containse (Object Value);	Возвращает true, если коллекция содержит элемент Value
public virtal int Count {get;};	Это свойство, предназначенное только для чтения, хранит текущую длину коллекции
public static ArrayList FixedSize(ArrayList AL);	Метод возвращает объект, элементы которого можно изменять, но нельзя добавлять или удалять
public virtual IEnumerator GetEnumerator();	Возвращает итератор для объекта
public virtual ArrayList GetRange(int Indx, int Count);	Возвращает диапазон элементов
public virtual int IndexOf(Object Value);	Возвращает индекс элемента со значением Value
public virtual void Insert (int Indx, Object Value);	Вставляет элемент Value на нужное место Indx коллекции
public virtual void InsertRange)int Indx, IColliction col);	Вставляет диапазон элементов
public virtual bool IsFixedSize{get;}	Возвращает true, если объект имеет фиксированный размер
public virtual bool IsReadOnly{get;}	Возвращает true, если объект имеет элементы, предназначенные только для чтения
public virtual Object this[int Indx]{set; get;}	Это свойство позволяет обратиться к элементам по индексу

Продолжение таблицы 12.1

public virtual int LastIndexOf(Object Value);	Возвращает индекс последнего вхождения в коллекцию значения Valiue
public static ArrayList ReadOnly(ArrayList AL);	Устанавливает для элементов коллекции режим только чтение
public virtual void Remove (Object Value);	Удаляет из коллекции первое вхождение элемента со значением Value
public virtual void RemoveAt (int Indx);	Удаляет из коллекции элемент с индексом Indx
public virtual void RemoveRange(int Indx, int count);	Удаляет из коллекции count элементов, начиная с элемента с индексом Indx
public static ArrayList Repeat(Object Value, int count);	Возвращает коллекцию, в которой элемент Value повторен count раз
public virtual void Reverse();	Изменяет порядок следования элементов на обратный
public virtual void SetRange (int Indx, ICollection col);	Вставляет коллекцию col, начиная с индекса Indx
public virtual void Sort();	Сортирует коллекцию
public virtual void TrimToSize();	Устанавливает емкость коллекции равной количеству ее элементов

По умолчанию начальная емкость коллекции ArrayList составляет 16 элементов. При расширении коллекции ее емкость удваивается, составляя 32, 64, 128 и т. д. элементов. Метод TrimToSize() отсекает неиспользуемые элементы.

Задача 12.1 Использовать класс ArrayList для организации коллекции «Гараж», объединяющей объекты «Автомобиль». Для работы с коллекцией использовать свойства и методы таблицы 12.1.

Это чисто учебная программа, поэтому количество полей у объекта «Автомобиль» всего два – марка автомобиля и его цена.

В учебных целях использован элемент управления TabControl – набор вкладок, с помощью которого реализовано как меню программы, так и «дополнительные формы» с элементами управления каждого режима меню.

В программе использованы элементы управления Panel – панель как для размещения на них результатов работы программы, так и для «скрытия» или «открытия» результатов работы программы «на форме» – режим «Просмотр» команда «Просмотр графика цены».

Особенность коллекции ArrayList является то, что коллекция предназначена для хранения объектов произвольного типа, поэтому некоторые методы из таблицы 12.1 требуют дополнительной настройки интерфейсных классов.

Во время проверки программы, если число объектов коллекции не превышает 4, то по умолчанию начальная емкость коллекции равна 4, а не 16 как утверждается во многих учебника. При расширении коллекции емкость удваивается – 8 (если число объектов больше 4, но меньше 9), 16 и т.д.

Исходный код программы:

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
   InitializeComponent();
   textBox3.Text = "";
   panel2.Visible = true;
   panel3.Visible = false;
   panel1.Visible = false;
  }
  public class Avto
  {
   public string Marka;
   public int Cena;
   public Avto(string sm, int sc)
   {
    Marka = sm; Cena = sc;
   }
  };
  public ArrayList Garaj = new ArrayList();
  private void button1_Click(object sender, EventArgs e)
  {
   panel2.Visible = true;
   panel3.Visible = false;
   panel1.Visible = false;
   string n;
   int c;
   n = textBox1.Text;
   c = Convert.ToInt32(textBox2.Text);
   Avto at = new Avto(n, c);
   Garaj.Add(at);
  }
  private void button2_Click(object sender, EventArgs e)
  {
   panel2.Visible = false;
```

```
   panel3.Visible = false;
   panel1.Visible = true;
   int i = 0;
   dataGridView1.Rows.Clear();
   foreach (Avto at in Garaj)
   {
    dataGridView1.Rows.Add();
    dataGridView1.Rows[i].Cells[0].Value = at.Marka;
    dataGridView1.Rows[i].Cells[1].Value = at.Cena;
    i++;
   }
  }
  private void button3_Click(object sender, EventArgs e)
  {
   panel2.Visible = false;
   panel3.Visible = false;
   panel1.Visible = false;
   this.Invalidate();
  }
  private void button4_Click(object sender, EventArgs e)
  {
   panel2.Visible = false;
   panel3.Visible = true;
   panel1.Visible = false;
   int i = Garaj.Capacity;
   int j = Garaj.Count;
   textBox3.AppendText("Коллекция состоит из " + j.ToString()+
" элементов" + "\r\n");
   textBox3.AppendText("Текущая размерность коллекции = " +
 i.ToString() +" элементов" + "\r\n");
  }
  private void button5_Click(object sender, EventArgs e)
  {
   panel2.Visible = false;
   panel3.Visible = true;
   panel1.Visible = false;
   Avto at1 = new Avto("", 0);
   Avto at2 = new Avto("", 0);
   int i=0;
   foreach (Avto at in Garaj)
   {
    if (i==0) at1=at;
    at2 = at;
    i++;
   }
   int j = Garaj.Count;
   Garaj.RemoveAt(j-1);
   Garaj.RemoveAt(0);
   Garaj.Insert(0, at2);
   Garaj.Insert(j-1, at1);
   textBox3.AppendText("Обмен закончен" + "\r\n");
  }
  private void button6_Click(object sender, EventArgs e)
```

```
  {
   Garaj.RemoveAt(0);
   panel2.Visible = false;
   panel3.Visible = true;
   panel1.Visible = false;
   textBox3.AppendText("Из коллекции удален 0
элемент"+"\r\n");
  }
  private void Form1_Paint(object sender, PaintEventArgs e)
  {
   string ss;
   Graphics g = e.Graphics;
   Pen myPen = new Pen(Color.Red, 2);
   g.DrawLine(myPen, 50, 150, 50, 50);
   g.DrawLine(myPen, 50, 150, 250, 150);
   int h,i = 0;
   foreach (Avto at in Garaj)
   {
    h = at.Cena/100;
    if (i < 8)
    {
   g.FillRectangle(Brushes.Blue, i * 25 + 55, 150 - h, 20, h);
     ss = i.ToString();
     g.DrawString(ss, new Font("10_IC_1", 12), Brushes.Black,
 (i + 1) * 25 + 30, 160);
    }
    i++;
   }
  }
 }
}
```

Работа программы изображена на рисунках 12.2 –12.6.

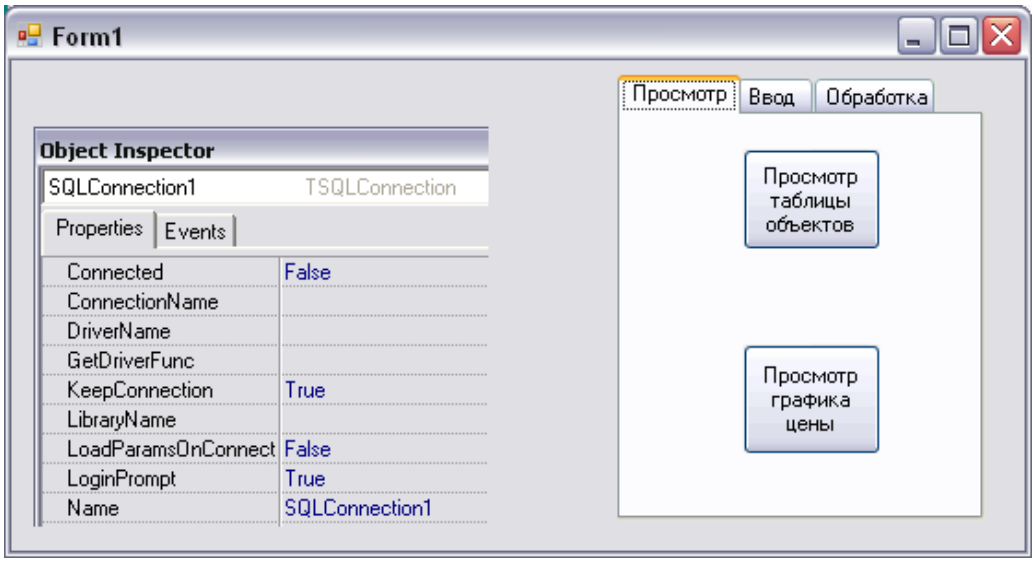


Рисунок 12.2 – Окно запуска программы с рисунком

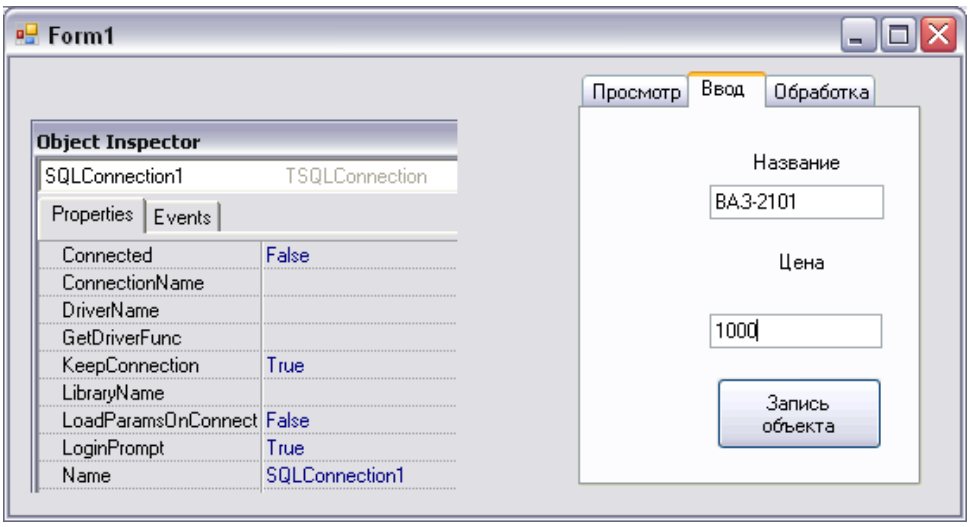


Рисунок 12.3 – Режим ввода данных коллекции

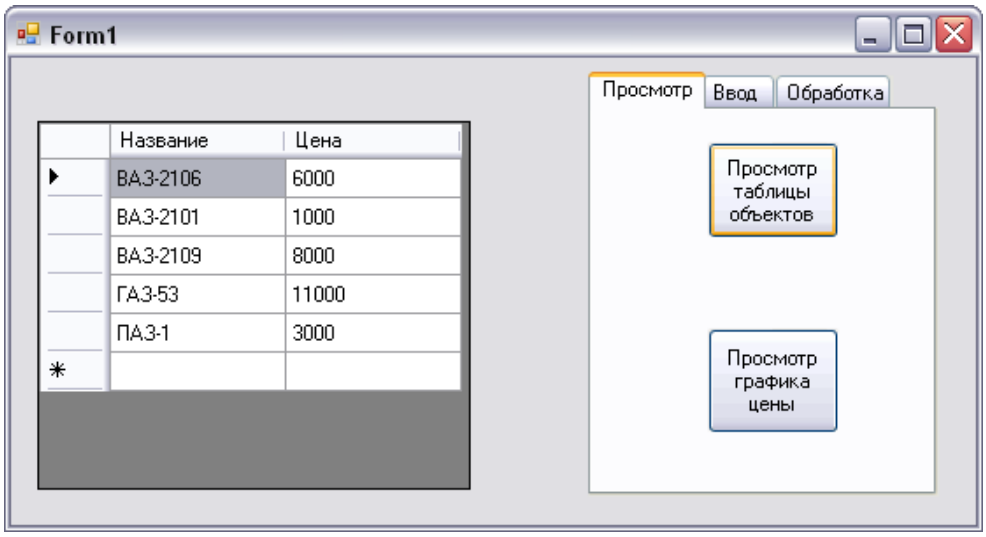


Рисунок 12.4 – Режим просмотра таблицы объектов коллекции

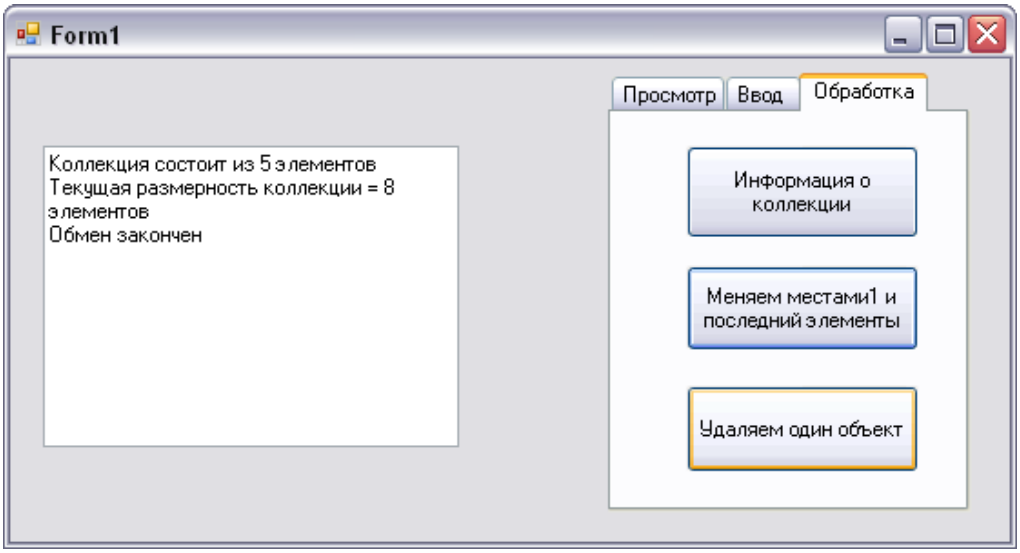


Рисунок 12.5 – Режим просмотра информации о коллекции

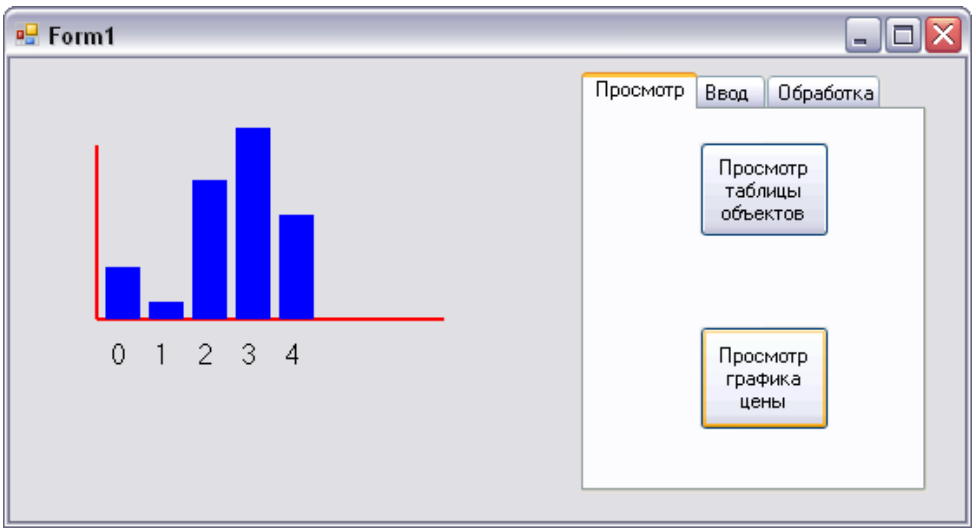


Рисунок 12.6 – Просмотр графика цен объектов коллекции

Как и во всех учебных программах в данном примере основное внимание уделялось технологии использования новых элементов управления и классов, а не «содержательной» стороне объектов этих классов. Некоторые значения цены или марки автомобилей взяты, что называется «с потолка».

12.5 Вопросы для самопроверки

1 Понятие композиции классов.

2 В чем основная цель композиции классов?

3 Понятие коллекции классов.

4 Какая объединяющая структура обычно используется для коллекционирования однотипных объектов?

5 Какие коллекции относятся к коллекциям с прямым доступом?

6 Какие коллекции относятся к коллекциям с последовательным доступом?

7 Какие коллекции относятся к иерархическим коллекциям?

8 Объекты какого типа могут включаться в коллекцию класса ArrayList?

9 Какое свойство класса ArrayList хранит текущую емкость коллекции?

10 Какое свойство класса ArrayList хранит текущую длину коллекции?

11 Какой метод класса ArrayList позволяет включать объект в коллекцию на нужное место?

13 ДЕЛЕГАТЫ

13.1 Понятие делегата

Существует множество задач, в которых необходимо выполнять вычисления с помощью «однотипных» методов, например, с помощью математических функций вещественного типа, имеющих вещественный аргумент (это все тригонометрические функции, логарифмы, экспонента и т.д.). В таких задачах очень хочется получить в свое распоряжение метод, в котором формальным параметром является имя вычисляемой функции или ссылка на имя.

В языке C# методы сами по себе не «гуляют», а могут определяться только внутри некоторых классов.

Поэтому в язык C# был включен специальный класс, позволяющий хранить ссылки на «однотипные» методы. Этот класс получил название Делегат.

Делегат это специальный класс, предназначенный для хранения ссылок на методы.

По определению – переменная типа класс является объектом. Класс, позволяющий описать некоторое множество объектов, каждый из которых является функцией (или ссылкой на функцию), называется функциональным типом.

Таким образом, делегаты в языке C# предназначены для описания функциональных типов. Экземплярами такого класса являются ссылки на функции (методы) – им также как переменным выделяются места в памяти компьютера, начальные адреса которых являются «точками» входа в функции и передаются ссылками.

Эта особенность делегата определила две основные области его применения – самостоятельно для решения некоторых задач (подобно приведенной выше) или для поддержки событий (смотри следующую лекцию).

В этой лекции мы будем рассматривать самостоятельное использование делегатов для решения некоторых задач.

13.2 Описание делегата

Формат записи делегата фактически задает сигнатуру (описание) методов, которые могут быть вызваны с его помощью:

```
[спецификаторы] delegate  <тип > <имя> (<параметры>);,
```

где

спецификаторы определяют условия доступа к делегату;

`delegate` —зарезервированное слово;

<тип> — тип возвращаемого результата;

<имя> — имя делегата (уникальный идентификатор);

<параметры> — формальные параметры вызова.

Например, описание всех функций вещественного типа, имеющих вещественный аргумент, имеет следующий вид:

```
public delegate double Funk(double argym);
```

Любая функция, соответствующая этому описанию, может использоваться в качестве параметра вызова умалчиваемого конструктора класса - делегата, который и возвращает конкретный экземпляр делегата – ссылку на функцию. Отличительной особенность работы делегата является формирование ссылок во время работы программы (динамически), а не на этапе ее компиляции.

13.3 Пример использования делегата

Для лучшего понимания механизма работы делегата рассмотрим решение следующей задачи: разработать приложение для вычисления пяти функций вещественного типа, имеющих вещественный аргумент: Sin(x), Log(x), Cos(x), Exp(x) и Round(x). Для выбора вычисляемой функции использовать делегат.

Исходный код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
   InitializeComponent();
  }
  public delegate double Funk(double x);//Объявление делегата
  public double rab(Funk f, double x) { return f(x); }
          //Объявление функции, использующей делегат
  private void button1_Click(object sender, EventArgs e)
  {
   string st;
   double x, y;
   textBox2.Text = "";
   x = Convert.ToDouble(textBox1.Text);
   st="Значение x=" +textBox1.Text+ "\r\n";
```

```
    textBox2.AppendText(st);
    y = rab(Math.Sin, x);             // Применение делегата
    st = "Sin(x)=" + y.ToString() + "\r\n";
    textBox2.AppendText(st);
    y = rab(Math.Log, x);           // Применение делегата
    st = "Log(x)=" + y.ToString() + "\r\n";
    textBox2.AppendText(st);
    y = rab(Math.Cos, x);           // Применение делегата
    st = "Cos(x)=" + y.ToString() + "\r\n";
    textBox2.AppendText(st);
    y = rab(Math.Exp, x);           // Применение делегата
    st = "Exp(x)=" + y.ToString() + "\r\n";
    textBox2.AppendText(st);
    y = rab(Math.Round, x);         // Применение делегата
    st = "Round(x)=" + y.ToString() + "\r\n";
    textBox2.AppendText(st);
   }
  }
}
```

Работа программы изображена на рисунке 13.1.

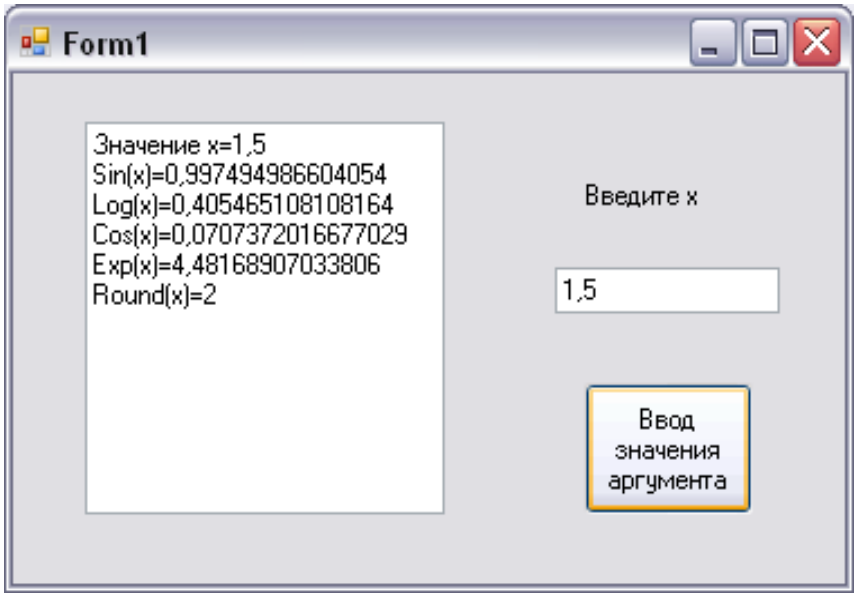


Рисунок 13.1 – Использование делегата

Естественно можно навести «красоту» в код программы, например,

```
private void button1_Click(object sender, EventArgs e)
{
 string st;
 Funk[] ff={Math.Sin, Math.Log, Math.Cos,
Math.Exp,Math.Round};
 string[] sfu = { "Sin", "Log", "Cos", "Exp", "Round" };
 double x, y;
 textBox2.Text = "";
 x = Convert.ToDouble(textBox1.Text);
```

```
 st="Значение x=" +textBox1.Text+ "\r\n";
 textBox2.AppendText(st);
 for (int i = 0; i < 5; i++)
 {
  y = rab(ff[i], x);
  st = sfu[i]+"=" + y.ToString() + "\r\n";
  textBox2.AppendText(st);
 }
}
```

Идея объявления и использования делегата в программе полностью рассмотрена в коде программы этого учебного примера.

Еще раз отметим особенность делегатов – динамически (во время выполнения программы) обращаться к вызываемым методам. Эта особенность делегатов часто используется при создании базовых конструкций программ, в которые можно добавлять различные создаваемые фрагменты, например, фрагменты меню, включающие «однотипные» методы.

13.4 Совместимость делегатов

Рассматривая «совместимость» делегатов необходимо рассмотреть два вопроса – совместимость типов делегатов и совместимость экземпляров объектов делегатов.

Типы делегатов всегда несовместимы друг с другом, даже если их форматы записей совпадают, например,

```
public delegate void Funk1();
public delegate void Funk2();
. . .
Funk1 d1 = Metod1;
Funk2 d2 = d1;   //Ошибка о несовместимости типов
```

Если для объявления делегатов использован один и тот же тип, то имеет смысл говорить о совместимости (равенстве) экземпляров делегатов. Экземпляры однотипных делегатов считаются равными, если они получают ссылку на один и тот же метод, например,

```
public delegate void Funk1();
. . .
Funk1 d1 = Metod1;
Funk1 d2 = Metod1;
```

В данном примере можно говорить о равенстве экземпляров делегатов d1 и d2.

13.5 Методы базовых классов делегатов

Необходимо отметить, что в CTS платформы .NET есть абстрактные классы System.Delegate и System.MulticastDelegate из которых (как из базовых) создаваемые делегаты могут наследовать некоторые методы.

Перечень свойств и методов, наиболее часто наследуемых создаваемыми делегатами от System.MulticastDelegate, можно представить следующим списком:

public MethodInfo Method {get:}– это свойство возвращает имя метода, на который указывает делегат;

public object Target {get:} – если делегат указывает на метод – член класса, то Target возвращает имя класса метода; если метод статический, то Target возвращает null;

public static Delegate Combine(Delegate[]) – добавляет новые методы к группе методов, обрабатываемых делегатом;

public static Delegate Remove(Delegate source, Delegate value) – удаляет метод value из списка методов делегата source;

public sealed override Delegate[] GetInvocationList() – возвращает список всех связанных с делегатом методов.

Для делегатов существует понятие многоадресный делегат – делегат, способный указывать на любое количество функций. Поскольку все делегаты языка C# являются производными классами от System.MulticastDelegate, то любой делегат C# потенциально является многоадресным.

Применим свойство многоадресности для делегата нашей программы вычисления вещественных функций. Изменения связаны только с обработчиком кнопки, поэтому приведен только фрагмент этого кода:

```
private void button1_Click(object sender, EventArgs e)
{
 string st;
 Funk[] ff={Math.Sin, Math.Log, Math.Cos, Math.Exp,
Math.Round};
 string[] sfu = { "Sin", "Log", "Cos", "Exp", "Round" };
 double x, y;
 textBox2.Text = "";
 x = Convert.ToDouble(textBox1.Text);
 st="Значение x=" +textBox1.Text+ "\r\n";
 textBox2.AppendText(st);
   Funk df = Math.Sin;
   for (int i = 1; i < 5; i++)
                df += ff[i];   //df = df + ff[i];
   for (int i = 4; i >= 0; i--)
   {
    y = rab(df, x);
    df -=ff[i];  //df = df - ff[i];
    st = sfu[i]+"=" + y.ToString() + "\r\n";
```

```
        textBox2.AppendText(st);
    }
}
```

Работа программы приведена на рисунке 13.2.

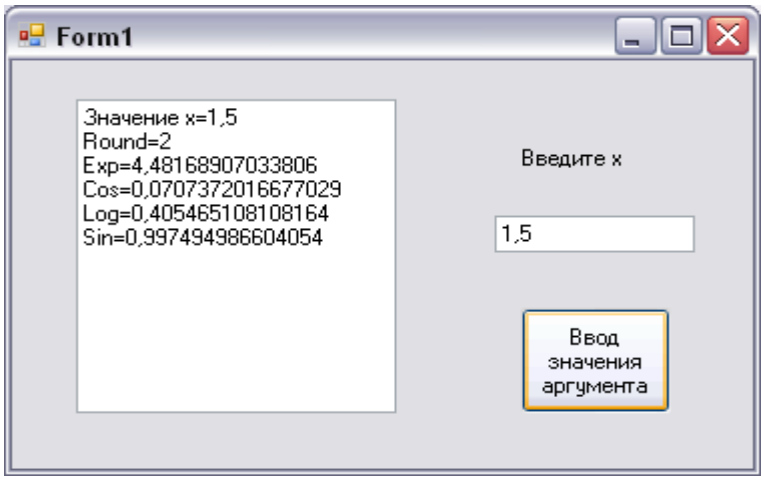


Рисунок 13.2 – Использование многоадресного делегата

Из рисунка 13.2 видно, что работа многоадресного делегата организована по принципу стека.

Заканчивая рассмотрения делегатов как специальный класс, предназначенный для хранения ссылок на методы, необходимо отметить, что любой объект представляет собой ссылку (в том числе и объект делегата) и может быть передан в некоторые методы в качестве параметра для дальнейшей работы.

Таким образом обеспечивается функциональная параметризация: в методы можно передавать не только данные, но и различные методы их обработки.

13.6 Вопросы для самопроверки

1 Понятие делегата.

2 Какой класс называется функциональным типом?

3 Что является экземпляром класса типа делегат?

4 Какое служебное слово используется при объявлении делегата?

5 Какой метод или функция может использоваться в качестве параметра вызова делегата?

6 Когда формируются ссылки на параметры вызова делегата?

7 Когда совместимы типы делегатов?

8 Когда совместимы экземпляры делегатов?

9 Какие абстрактные классы платформы .NET можно наследовать при создании делегатов?

10 Понятие многоадресного делегата?

14 СОБЫТИЯ

14.1 Понятие события

В основе работы операционной системы Windows лежит принцип событийного управления. Это означает, что и сама система и все приложения, написанные для Windows, после запуска ожидают действий пользователя или сообщений операционной системы и реагируют на них определенным образом – через очередь сообщений приложения формируются события, на которые приложение должно реагировать.

Каждый элемент управления (кнопка или строка меню) имеет свой идентификатор. Когда Вы нажимаете на кнопку или выбираете строку меню, в очередь сообщений приложения Windows заносит сообщение, содержащее идентификатор использованного элемента управления. Таким образом операционная система Windows направляет сообщение от использованного элемента управления в очередь того приложения, к которому принадлежит данный элемент управления.

Если в окне формы была нажата кнопка, то факт нажатия кнопки (сообщение) через операционную систему Windows вернется в очередь сообщений окна нашей формы и у кнопки возникнет событие Click.

Итак, механизм появления события понятен, но что это событие?

Из лекции 7 (состав класса) мы определили события класса как специальные методы, позволяющие классу реагировать на действия пользователя или на определенные изменения в программе.

Из этого определения событие и метод – обработчик события являются синонимами.

Уточним эти понятия. Каждый объект является экземпляром некоторого класса. Свойства класса определяют его состояние, а методы класса определяют его поведение.

У всех объектов один и тот же набор свойств, но значения свойств объектов различны, так что объекты одного класса могут находиться в разных состояниях. Например, значение свойства «доход», у объектов класса «Служащий», может очень сильно различаться, и эти объекты могут находиться в разных состояниях. Например, «Хочу купить автомобиль, но не имею возможности …».

Все объекты обладают одними и теми же методами и набор событий у всех объектов одного класса один и тот же, но вот методы, обрабатывающие возникшие события, могут быть разные. Например, у двух кнопок окна формы («Ввод данных», «Переход на форму 2») одинаковые события Click, но обработчики событий (их коды) разные.

Таким образом, событие как свойство класса, являясь единым для всех объектов, имеет различную реализацию для каждого конкретного экземпляра класса.

События позволяют задать индивидуальное поведение объекта в специфических ситуациях, когда возникает некоторое сообщение.

Таким образом, события класса это специальные методы, позволяющие классу реагировать на действия пользователя или на определенные изменения в программе.

Для многих действий пользователя (они предсказуемы) разработаны заготовки для обработчиков событий – смотри Properties ->Events.

14.2 Некоторые часто используемые события среды Visual Studio.NET

Для каждого элемента управления Windows - приложений определен свой набор событий, на которые он может реагировать. Заготовка шаблона обработчика события формируется двойным щелчком на поле, расположенном справа от имени соответствующего события на странице Events окна Properties для выбранного элемента управления.

Наиболее часто используемыми событиями для элементов управления являются следующие:

Click, DoubleClick – одинарный или двойной щелчок мышки;

KeyDown, KeyUp – нажатие и отпускание любой клавиши и их сочетаний;

KeyPress – нажатие клавиши, имеющей ASCII-код;

MouseDown, MouseUp – нажатие и отпускание кнопки мыши;

MouseMove – перемещение мыши;

Paint – возникает при необходимости прорисовки формы.

Например, если «Кликнуть» на кнопку, расположенную в окне нашей формы, то будет сформирован обработчик этого события со следующим заголовком:

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

Если нам необходимо обрабатывать событие, связанное с двойным нажатием кнопки мышки в пределах формы, то в «событиях» формы необходимо дважды «Кликнуть» на поле, расположенное справа от события MouseDoubleClick при этом появится шаблон обработчика этого события со следующим заголовком: private void Form1_MouseDoubleClick(object sender, MouseEventArgs e).

В качестве примеров приводим шаблоны еще двух обработчиков событий:

```
private void textBox1_KeyDown(object sender,
KeyEventArgs e)
private void Form1_MouseClick(object sender,
MouseEventArgs e)
```

Попробуйте определить сами, когда они формируются?

Анализируя шаблоны обработчиков событий можно отметить общую закономерность – у всех шаблонов два формальных параметра, причем первый (object sender) у всех одинаковый.

Во многих учебниках по программированию на языке C# при пояснении механизма работы события используется модель «публикация – подписка», в которой один класс, являющийся отправителем сообщения (sender), публикует сообщение, а другие классы, являющиеся получателями сообщения (receivers), подписываются на получение этих сообщений.

В соответствии с этой терминологией первый формальный параметр всех обработчиков сообщений sender типа object (класс object является базовым любого класса языка C#) является отправителем сообщения (иногда говорят – источником сообщения).

Второй формальный параметр шаблонов обработчиков событий является переменной типа объект точнее ссылкой на объект класса XXXEventArgs, который содержит связанные с событием параметры (фактически сообщение), например, для обработчиков мышки можно получить координаты e.X e.Y курсора мышки. Класс XXXEventArgs (XXX имя события) во всех стандартных событиях является наследником системного класса EventArgs.

При описании модели «публикация – подписка» отмечалось, что получатели сообщений (receivers), подписываются на получение этих сообщений. То есть для элементов управления и формы необходимо указывать (подписывать) сообщения, на которые должен реагировать этот элемент. Поскольку все методы обработчиков событий имеют единый формат записи, то их можно объединять (подписывать) с помощью многоадресных делегатов – делегатов, способных указывать на любое количество функций. Для каждого элемента в файле Form1.Designer.cs, в разделе инициализации помимо определения свойств элементов управления, определяется перечень событий, на которые должен реагировать этот элемент управления. Например, фрагмент инициализации формы с определением перечня событий, на которые она должна реагировать, имеет следующий вид:

```
private void InitializeComponent()
{
 . . .
this.MouseDoubleClick += new
System.Windows.Forms.MouseEventHandler
 (this.Form1_MouseDoubleClick);
this.Paint += new
System.Windows.Forms.PaintEventHandler(this.Form1_Paint);
this.MouseClick += new System.Windows.Forms.MouseEventHandler
 (this.Form1_MouseClick);
this.MouseMove += new System.Windows.Forms.MouseEventHandler
(this.Form1_MouseMove);
     . . ..
}
```

Необходимо отметить, что все визуальные элементы управления произошли от класса Control, в котором представлены наиболее важные для работы события.

14.3 Пример использования стандартных событий классов

Рассмотрим чисто учебную программу, использующую некоторые «стандартные» события классов.

В программе использован обработчик события одинарного нажатия левой кнопки мышки на форме, при этом из всех запусков обработчика программа реагирует только на нажатие кнопки мышки, когда ее курсор находится в левом верхнем углу формы:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
  {
if ((e.X < 20 && e.X > 0 && e.Y < 20 & e.Y > 0) && p == 0)
```

Этот прием часто используется программистами для формирования реакции программы на «клик» мышки в определенных местах экрана, например, карты города или фото группы студентов – легко написать необходимые комментарии при «попадании» мышки в определенный участок формы.

Обработчик события перемещения мышки по форме (дополнительное условие p==1) позволяет записать положения курсора мышки в массив 100 точек.

Просмотр массива в графическом режиме осуществляется с помощью обработчика двойного нажатия левой кнопки мышки.

С помощью обработчика – нажатия клавиши в окне редактора осуществляется начальная установка программы.

Исходный код программы:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public int p,i,j;
  int[,] tk = new int[100, 2];
  public Form1()
  {
   InitializeComponent();
   textBox1.Text = "";
```

```
   p = 0; i = 0; j = 0;
  }
  private void Form1_MouseClick(object sender, MouseEventArgs
e)
  {
   if ((e.X < 20 && e.X > 0 && e.Y < 20 & e.Y > 0) && p == 0)
   {
    textBox1.AppendText("Вы кликнули мышкой в верхнем левом
углу
\r\n");
    p++;
   }
  }
  private void Form1_MouseMove(object sender, MouseEventArgs
e)
  {
   if (p == 1)
    if (i < 100) { tk[i, 0] = e.X; tk[i, 1] = e.Y; i++; }
     else
     {
      textBox1.AppendText("Заполнение массива закончено
\r\n");
      p++;
     }
  }
  private void Form1_MouseDoubleClick(object sender,
MouseEventArgs e)
  {
   this.Invalidate();
  }
  private void Form1_Paint(object sender, PaintEventArgs e)
  {
   Pen myPen = new Pen(Color.Red, 2);
   Graphics g = e.Graphics;
   for (int n = 0; n < 100; n++)
     g.DrawEllipse(myPen, tk[n, 0], tk[n, 1], 2, 2);
  }
  private void textBox1_KeyDown(object sender, KeyEventArgs e)
  {
   p = 0; i = 0;
  }
 }
}
```

Работа программы представлена на рисунках 14.1 и 14.2.

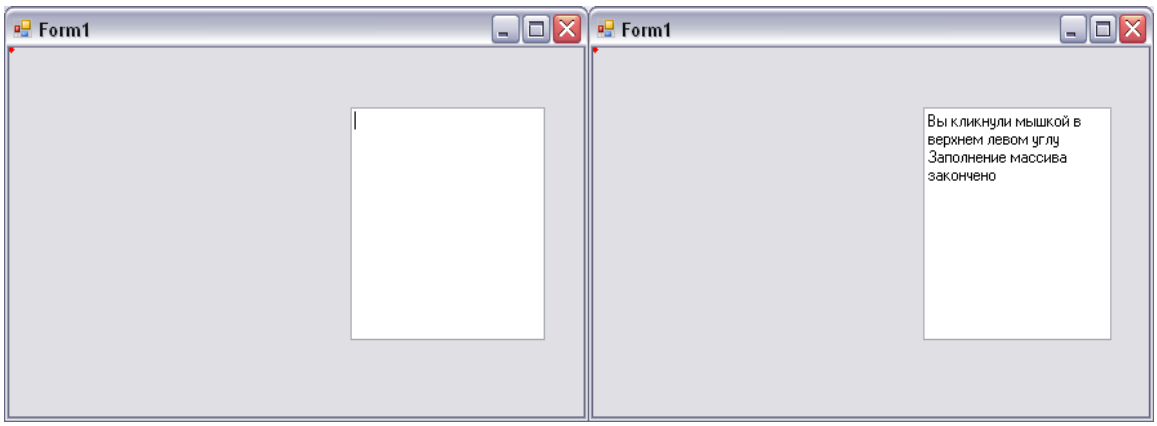


Рисунок 14.1 – Запуск программы и заполнение массива

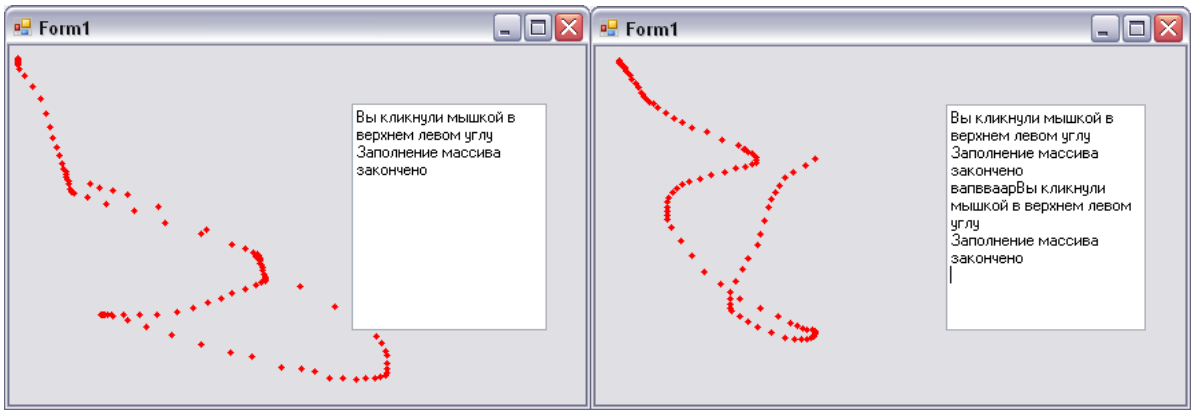


Рисунок 14.2 – Просмотр массива, сброс программы и ее повторный запуск

14.4 Нестандартные события классов

Помимо стандартных событий – как реакция программы на действия пользователя, события могут формироваться и внутри Windows-приложений с помощью специальных методов.

События, формируемые самим приложением, позволяют организовать динамическую (во время работы приложения) связь между различными объектами, например, продажа некоторого товара в магазине должна автоматически изменять содержимое многих документов – общий доход от продажи, документ по продаже конкретного товара, наличие этого товара на складе и рекомендации по пополнению товара и т.д.

Таким образом, возникает необходимость информирования некоторых объектов приложения о событии, происшедшем в объекте источнике сообщения.

Механизм взаимодействия источника события с его получателями (иногда их называют клиентами) основан на использовании делегата. В приложении объявляется экземпляр делегата, которому соответствуют стандартные методы обработчиков событий.

Далее необходимо определить класс, являющегося источником события (sender), и в нем метод описывающий событие, и метод инициирующий событие.

В процессе работы приложения объекты обработчиков событий (клиентов), которые хотят получать уведомление об изменении состояния источника, необходимо включить в список методов, обрабатываемых делегатом. Этот процесс называется регистрацией обработчиков событий.

При появлении события все зарегистрированные методы поочередно, с помощью делегата, вызываются на выполнение.

Работу механизма взаимодействия источника события с его получателями рассмотрим на следующем учебном примере.

Задача 14.1 Сформировать массив из 10 случайных целых чисел в диапазоне от минус 5 до 10. Предположим, что отрицательные значения недопустимы и для них мы будем формировать события**.** Необходимо напечатать массив, вычислить его сумму и нарисовать график изменения значений.

Если значение массива число отрицательное, то необходимо формировать событие, на которое должны реагировать два обработчика. Первый должен поменять знак у отрицательного числа, а второй должен изменить сумму чисел в соответствии с новым значением элемента массива.

Для наглядности предусмотрим вывод графика измененных значений массива.

Итак, для создания и использования события нужно, прежде всего, объявить делегата, с помощью которого реализуется связь между источником события и его клиентами.

В библиотеке .NET описано огромное количество стандартных делегатов, предназначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам:

– имя делегата включает название события и заканчивается суффиксом EventHandler;

– делегат имеет два формальных параметра. Первый параметр задает источник события и имеет тип object. Второй параметр задает аргументы события и имеет тип EventArgs или производный от него.

Рекомендуется при объявлении делегата придерживаться этих правил, например:

```
public delegate void ZamenaEventHandler(object
sender, ZamenaEventArgs arge);
```

В этом примере мы создаем делегата для обработки события Zamena (замена), связанного с изменениями, которые должны произойти в объекте - источнике события. В описании делегата указаны два аргумента: объект sender, создавший событие, и объект arge типа ZamenaEventArgs, содержащий связанные с событием параметры.

Поскольку нам понадобится передавать только индекс массива, то лучше определить класс ZamenaEventArgs следующим образом:

```
public class ZamenaEventArgs : EventArgs
   {
    public int item;
       }
```

Следующим этапом в реализации механизма обработки событий является определение класса, являющегося источником события (sender), и в нем метода, инициирующего событие. Этот метод имеет специальный формат записи и во многом определяется форматом записи соответствующего ему делегата – после определения спецификатора доступа к методу (обычно это public) записывается служебное слово event, после которого указывается тип, заданный делегатом, и имя события. Например:

```
public event ZamenaEventHandler  Zamena;
```

Полное описание класса выглядит следующим образом:

```
class sobit
  {
      public event ZamenaEventHandler  Zamena;
      public void prov(ZamenaEventArgs arge)
        {
          if (masi[arge.item] < 0)
            {
                Zamena(this,arge);
             }
      }
 }
```

На следующем этапе реализации механизма обработки событий необходимо определиться с классами получателями событий. Особенность этих классов является то, что в них должны находиться методы обработчиков событий, формат записи которых должен совпадать с форматом записи соответствующего делегата. Например:

```
 class zam1
  {
     public void OnZam1(object sender, ZamenaEventArgs e)
     {
         masi[e.item] = masi[e.item] * (-1);
     }
 }
```

Регистрацию обработчиков событий необходимо производить для объектов (не классов) классов обработчиков событий. Это возможно только во время работы приложения – объекты (переменные) создаются во время работы приложения, например, во время инициализации формы:

```
     public Form1()
 {
```

```
      InitializeComponent();
      zam1 z1 = new zam1();
      zam2 z2 = new zam2();
      so.Zamena += z1.OnZam1;
      so.Zamena += z2.OnZam2;
 }
```

Для демонстрации работы приложения использованы две кнопки – «Формирование массива» и «Проверка массива и запуск событий».

Исходный код программы имеет следующий вид:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public delegate void ZamenaEventHandler(object sender,
ZamenaEventArgs arge);
  public int cob=0;
  public static int sum = 0;
  public static int[] masi = new int[10];
  public class ZamenaEventArgs : EventArgs
  {
   public int item;
  }
  class sobit
  {
   public event ZamenaEventHandler  Zamena;
   public void prov(ZamenaEventArgs arge)
   {
    if (masi[arge.item] < 0)
    {
     Zamena(this,arge);
    }
   }
  }
  sobit so = new sobit();
  class zam1
  {
   public void OnZam1(object sender, ZamenaEventArgs e)
   {
    masi[e.item] = masi[e.item] * (-1);
   }
  }
  class zam2
  {
   public void OnZam2(object sender, ZamenaEventArgs e)
```

```
    {
        sum = sum + 2 * masi[e.item];
    }
   }
  public Form1()
  {
   InitializeComponent();
   zam1 z1 = new zam1();
   zam2 z2 = new zam2();
   so.Zamena += z1.OnZam1;
   so.Zamena += z2.OnZam2;
  }
  private void button1_Click(object sender, EventArgs e)
  {
   cob = 0; sum = 0;
   string ss="";
   Random rnd = new Random();
   for(int i=0;i<10;i++)
   {
    masi[i] = rnd.Next() % 15 - 5;
    sum = sum + masi[i];
    ss = ss + masi[i].ToString() + " ";
   }
   textBox1.AppendText("Исходный массив: \r\n");
   textBox1.AppendText(ss + "\r\n");
   textBox1.AppendText("Сумма
элементов="+sum.ToString()+"\r\n");
   this.Invalidate();
  }
  private void button2_Click(object sender, EventArgs e)
  {
   cob=1;
   ZamenaEventArgs zz = new ZamenaEventArgs();
   for (int i = 0; i < 10; i++)
   {
    zz.item = i;
    so.prov(zz);
   }
   textBox1.AppendText("Сумма
элементов="+sum.ToString()+"\r\n");
   this.Invalidate();
  }
  private void Form1_Paint(object sender, PaintEventArgs e)
  {
   Graphics g = e.Graphics;
   g.DrawLine(new Pen(Brushes.Blue, 2), 10, 200, 250, 200);
   Pen myPen = new Pen(Color.Red, 2);
   if (cob == 0)
   {
    for (int i = 1; i < 10; i++)
     g.DrawLine(myPen,i*10,(200-masi[i-1]*10),(i+1)*10,(200-
masi[i]*10));
   }
```

```
  if (cob == 1)
  {
   for (int i = 1; i < 10; i++)
    g.DrawLine(myPen,i*10+110,(200-masi[i-1]*10),(i+1)*10+110,
(200-masi[i]*10));
  }
 }
}
}
```

Работа программы представлена на рисунке 14.3.

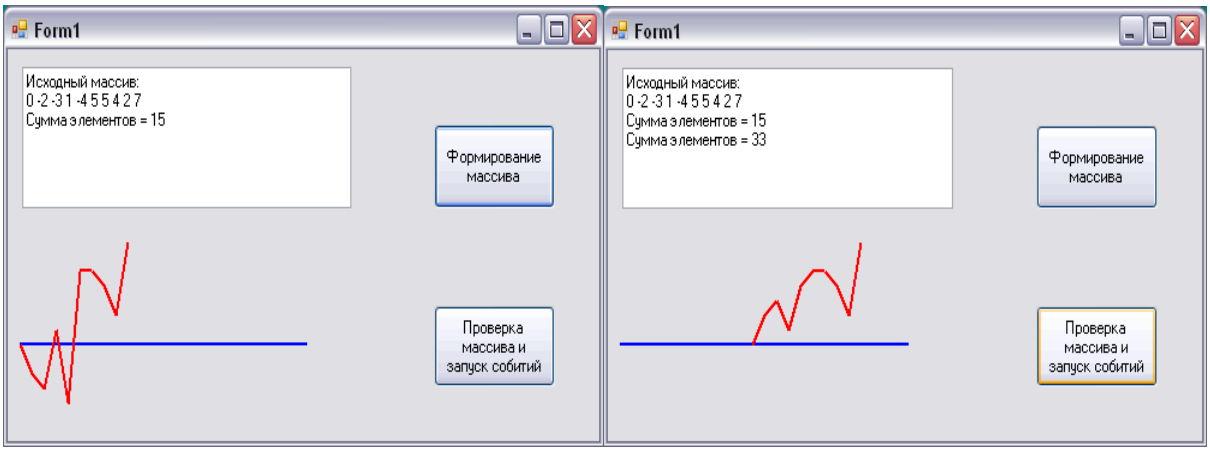


Рисунок 14.3 Работа приложения с обработкой событий

14.5 Вопросы для самопроверки

1 Понятие события.

2 Где находятся заготовки обработчиков событий на действия пользователя для элементов управления?

3 Как обычно называются события класса?

4 Что произойдет, если дважды «Кликнуть» на кнопку, расположенную в окне нашей формы?

5 Что определяет первый формальный параметр всех обработчиков сообщений?

6 Что определяет второй формальный параметр обработчиков сообщений?

7 Где перечислены события, на которые должны реагировать элементы управления формы?

8 Как можно объединить все события класса?

9 Где происходит добавление событий в многоадресный делегат формы?

10 Где объявляется многоадресный делегат формы?

Приложение А

Ответы на вопросы для самопроверки

СРЕДА ВИЗУАЛЬНОГО ПРОГРАММИРОВАНИЯ VISUAL STUDIO.NET

1 Понятие события в работе компьютера?

Событие это появление любой «нестандартной» ситуации в работе компьютера, например, нажатии клавиши на клавиатуре, перемещении курсора мыши, деления на ноль и т.д.

2 Как система Windows «различает» события?

Каждое событие имеет свой индивидуальный номер – «вектор прерывания».

3 Что делает система Windows, получив информацию о появлении события?

Определив номер события система Windows «запускает» соответствующий драйвер.

4 Понятие драйвера.

Это специальная программа, корректно реагирующая на соответствующее событие.

5 Понятие сообщения.

В общем случае сообщения – это реакция операционной системы Windows на происходящие в системе события.

6 Что содержит сообщение?

Сообщение Windows является записью, которая содержит информацию о том, что произошло и дополнительную информацию (параметры) о произошедшем событии.

7 Куда поступают сообщения, получаемые системой Windows от драйверов?

Все сообщения, получаемые Windows, помещаются в системную очередь сообщений, которая существует в единственном варианте. Далее из системной очереди сообщения распределяются в очереди сообщений отдельных Windows-приложений.

8 Зачем в каждом приложении имеется цикл обработки сообщений поступающих от Windows?

С помощью этого цикла приложение извлекает «свои» сообщения и передает их соответствующим обработчикам сообщений окна приложения.

9 Какой метод приложения реализует цикл обработки сообщений поступающих от Windows?

Application.Run(new Form1());

10 Для чего предназначен класс Form?

Класс Form определяет пользовательский интерфейс приложения.

ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

1 Для чего предназначен Windows.Forms.Designer платформы .NET?

Продолжение приложения А

Это конструктор или дизайнер форм Windows платформы .NET. Он позволяет в интерактивном режиме выполнять визуальное проектирование формы приложения.

2 Что означает служебное слово “partal” в описании класса формы?

Это служебное слово означает, что в приведенном описании находится только часть всего кода описания класса формы.

3 Какая часть описания класса формы содержит обработчики сообщений?

Описание обработчиков сообщений находится в файле с именем Form1.cs.

4 С какого метода начинается выполнение Windows приложении?

Работа Windows приложения начинается выполнением метода Main().

5 Какой метод создает объект класса Form1 при запуске программы?

Создание и инициализацию объекта класса Form1 при запуске программы осуществляет метод Application.Run().

6 Для чего предназначен управляющий элемент Label?

Этот управляющий элемент предназначен для размещения информации на форме.

7 Какое свойство управляющего элемента Label позволяет выводить информацию в окно формы?

Для вывода информации в управляющем элементе Label используется свойство Text.

8 Каким свойством управляющего элемента Label можно задать его «прозрачность»?

«Прозрачность» элемента Label можно задать с помощью свойства BackColor.

9 С помощью какого свойства можно нанести изображение на кнопку в окне формы?

Изображение на кнопку можно наносить с помощью свойства Image.

10 Как называется диалоговое окно, блокирующее дальнейшие действия с приложением до того момента, пока это окно не будет закрыто?

Такое диалоговое окно называется модальным диалоговым окном.

ГРАФИЧЕСКИЙ ИНТЕРФЕЙС ЯЗЫКА C#

1 Что означает GDI+ в языке C#?

Это сокращенное обозначение графического интерфейса приложений.

2 Что определяет объект класса Brush?

Объект класса Brush определяет, как и чем будет заполняться пространство внутри геометрических фигур.

3 Что определяет объект класса Pen?

Объект класса Pen определяет цвет и тип линий, при помощи которых рисуются контуры геометрических фигур.

4 Что определяет класс Graphics?

Он определяет набор свойств и методов для вывода текста, изображений и геометрических фигур на экран монитора.

Продолжение приложения А

5 Какое сообщение в системе Windows «следит»за перемещением и изменением размера окон?
За перемещением и изменением размера окон системе Windows «следит» сообщение WM_PAINT
6 Как называется обработчик, связанный с перерисовкой содержимого окна формы нашего приложения?
Обработчик, связанный с перерисовкой содержимого окна формы нашего приложения называется Form1_Paint.
7 Что определяет первый формальный параметр обработчика Form1_Paint?
Он передает ссылку на объект, вызвавший событие.
8 Что определяет второй формальный параметр обработчика Form1_Paint?
Через него передается ссылка на объект класса PaintEventArgs.
9 Понятие контекста устройства монитора.
Это специальные программы системы Windows, связывающие приложение с драйвером видеокарты компьютера.
10 Какой метод требует от операционной системы Windows сформировать для формы сообщение WM_PAINT?
Это метод this.Invalidate();.
ИСПОЛЬЗОВАНИЕ МЕНЮ В ПРИЛОЖЕНИИ
1 Для чего создается меню приложения?
Меню приложения позволяет в удобной форме использовать основные режимы работы приложения.
2 Чему должны соответствовать команды меню приложения?
Команды меню приложения должны соответствовать основным режимам работы приложения.
3 Какой элемент управления окна ToolBox позволяет создавать меню приложения?
Меню приложения позволяет создавать элемент управления MenuStrip.
4 Какое поле редактора меню приложения используется для ввода команды?
Для ввода команды в редакторе меню необходимо использовать поле Type Here.
5 С помощью какой команды редактора меню приложения можно вставить новую строку меню приложения между уже существующих строк?
Новую строку меню приложения между уже существующими строками можно вставить командой Insert New.
6 Зачем при создании меню приложения желательно использовать клавиатурные акселераторы?
Применение клавиатурных акселераторов является альтернативным мышки способом выбора команд.
7 Для чего предназначена команда Insert Separator редактора меню приложения?

Продолжение приложения А

С помощью этой команды редактора меню можно вставить разделительную линию между строками меню.

8 В чем принципиальное отличие элемента управления RichTextBox от TextBox?

Элемент RichTextBox позволяет работать с файлами разного типа.

9 Какой элемент управления используется для хранения изображений «иконок», дублирующих основные команды меню приложения?

Для хранения многих изображений используется элемент управления ImageList.

10 Какое свойство кнопки инструментальной панели, позволит снабдить каждую кнопку окном с поясняющим сообщением, которое появляется при «наведении» на кнопку курсора мыши?

Чтобы снабдить кнопки инструментальной панели окнами с «всплывающими» поясняющим сообщением необходимо использовать свойство ToolTipText.

ИСПОЛЬЗОВАНИЕ ДИАЛОГОВЫХ МЕНЮ

1 Какие элементы управления относятся к диалоговым категориям?

Это элементы, позволяющие использовать диалоговые ресурсы системы Windows для обращения к дискам, папкам и файлам компьютера.

2 Какой элементы управления предназначен для создания и обслуживания диалогового окна открытия файла?

Для создания и обслуживания диалогового окна открытия файла необходимо использовать элемент управления OpenFileDialog.

3 Какой метод отображает на экране стандартное диалоговое окно системы Windows для выбора файла при его открытии?

Это метод openFileDialog1.ShowDialog.

4 Как настроить элемент управления OpenFileDialog на открытие только текстовых файлов?

Необходимо свойству Filter элемента OpenFileDialog определить значение "Text files|*.txt".

5 Что произойдет, если в диалоговом окне открытия файла пользователь кликнул на кнопку «Открыть»?

Метод ShowDialog элемента OpenFileDialog возвращает значение DialogResult.OK.

6 Что означает часть условия openFileDialog1.FileName.Length > 0
в записи if (openFileDialog1.ShowDialog() == DialogResult.OK &&
openFileDialog1.FileName.Length > 0)?

Это часть условия означает, что «длина строки полного пути к выбранному пользователем файлу» должна быть больше нуля.

7 Какой метод элемента управления RichTextBox1 используется для открытия файла?

Это метод richTextBox1.LoadFile

8 Для чего предназначен элементы управления SaveFileDialog?

Продолжение приложения А

Он предназначен для создания и обслуживания диалогового окна сохранения файла.
9 Что определяет свойство FileName элемента управления SaveFileDialog?
Это свойство позволяет определить шаблон имени сохраняемого файла.
10 Что определяет параметр RichTextBoxStreamType.PlainText метода richTextBox1.SaveFile?
Что документ будет сохранен в формате Text.
ИСПОЛЬЗОВАНИЕ ДИАЛОГОВЫХ МЕНЮ
1 Что означает сокращение MDI?
Многооконный интерфейс документов.
2 В каких ситуациях имеет смысл проектировать главную форму как главную кнопочную форму?
Когда каждый режим приложения создаваемого проект предоставлен различными сложными сервисами, требующими собственного интерфейса.
3 Какой элемент управления обычно используется для размещения рисунков на форме?
Это элемент управления PictureBox.
4 Как добавить в проект новую форму с помощью окна Solution Explorer?
Необходимо кликнуть правой клавишей мыши строку названия проекта WindowsFormsAplication1 и в появившемся меню режимов работы выберите режим Add и в нем команду Add Windows Form
5 Как добавить в проект новую форму с помощью режима Project?
Необходимо выбрать режим Project, а в нем команду Add Windows Form и подтвердить название предлагаемой формы нажатием кнопки Add.
6 Чем отличается диалоговое (модальное) окно от обычного окна формы?
Из диалогового окна нельзя выйти, не закончив диалог и не закрыв форму.
7 Каким методом открывается обычное (не модальное) окно формы?
Методом Show().
8 Каким методом открывается модальное окно формы?
Методом ShowDialog().
9 Что делает следующий фрагмент программы:

```
private void button3_Click(object sender, EventArgs e)
    {
            Form5 f5 = new Form5();
            if (f5.ShowDialog() == DialogResult.OK) k = 0;
    }  ?
```

Создается новая форма, которая открывается в виде модального диалогового окна.
10 Какой элемент управления часто используется для табличной формы представления информации?
Элемент управления DataGridView.
ПОНЯТИЕ КЛАССА
1 Понятие класса?

Продолжение приложения А

Класс это тип данных, содержащий поля, методы и события.
2 Понятие свойства класса?
Это совокупность методов, позволяющих классу обмениваться (читать или записывать) значениями полей класса с другими классами программы.
3 Понятие конструктора класса?
Это специальные методы класса, которые предназначены для создания объектов класса и присваивания начальных значений полям класса.
4 Понятие деструктора класса?
Это специальные методы, определяющие порядок действий при освобождении ресурсов, выделенных объекту.
5 Понятие события класса?
Это специальные методы, позволяющие классу реагировать на действия пользователя или на определенные изменения в программе.
6 Понятие индексатора класса?
Это средство доступа к элементам данных класса (обычно массивам) по их порядковому номеру.
7 Назначение поля this объекта?
Это ссылка на адрес текущего объекта, которая позволяет методам класса работать с полями текущего объекта.
8 Как называют переменную типа класс?
Переменную типа класс обычно называют объектом.
9 Что означает служебное слово static в описании класса?
Это указание, что программа может использовать класс и его элементы, не создавая переменной этого класса (объекта класса).
10 Что означает служебное слово public в описании данных класса?
Начало описания данных открытых для доступа из программы.
ЭЛЕМЕНТЫ КЛАССОВ
1 Для чего предназначен конструктор с умолчанием?
Он создает объект и полям объекта присваивает некоторые фиксированные значения.
2 Для чего предназначен конструктор с заданием параметров?
Он создает объект и полям объекта присваивает задаваемые значения.
3 Как называется процесс определения нескольких методов с одинаковыми именами?
Перегрузкой методов.
4 Как называются несколько конструкторов одного класса?
Множественные конструкторы.
5 Когда вызывается деструктор класса tka?
Он вызывается автоматически сборщиком мусора при удалении объекта из кучи.
6 Как обычно называется метод, если тип его возвращаемого значения объявлен void?
Процедура.

Продолжение приложения А

7 Как должен заканчиваться в языке C# метод если его тип не void?
Работа метода должна заканчиваться выполнением Оператора return.
8 Какие формальные параметры метода C# называются параметры-ссылки?
Формальные параметры, перед которыми стоит слово ref.
9 Как называется объединение в одной структуре полей и методов их обработки?
Инкапсуляция.
10 Какой механизм используется в классах для доступа к «закрытым» полям класса?
Свойства.
ПРИНЦИПЫ ООП
1 Как один объект может информировать другой объект об изменении своего состояния?
С помощью механизма событий.
2 Как называется объект, информирующий другой объект об изменении своего состояния?
Источник события.
3 Как формируется «пустой» обработчик некоторого события?
В окне свойств на странице событий необходимо дважды кликнут в соответствующей строке.
4 Какой обработчик события будет сформирован, если на форме дважды кликнут на кнопку button1?
private void button1_Click(object sender, EventArgs e)
5 Для чего используется перегрузка операций в классах?
Она позволяет использовать в обычных математических выражениях переменный типа класс.
6 Понятие наследования?
Наследование это способность производного класса использовать некоторые свойства, данные и методы базового класса.
7 Что является целью наследования классов?
Целью наследования является повторное использование уже созданных классов.
8 Как называется класс, свойства, данные и методы которого наследуются другим классом?
Базовый класс.
9 Как называется класс, наследующий свойства, данные и методы базового класса?
Производный класс.
10 На чем базируется объектно-ориентированное программирование?
На принципах инкапсуляции, наследовании и полиморфизме.
ПРИНЦИП ПОЛИМОРФИЗМА

Продолжение приложения А

1 Какой класс является базовым классом в иерархической цепочке всех классов языка C#?
System.Object.
2 Какой объект базового или производного классов создается раньше при вызове конструктора производного класса?
Объект базового класса.
3 В чем преимущество цепочек наследуемых классов?
Наследование последнего класса в такой цепочке позволяет получить в свое распоряжение поля, свойства и методы всех классов цепочки.
4 Что означает статическое наследование в языке C#?
Наследование, в котором связи между объектами определяются в процессе компиляции программы.
5 Что означает динамическое наследование в языке C#?
Наследование, в котором связи между объектами определяются в процессе выполнения программы.
6 Понятие полиморфизма в языке C#?
Полиморфизм это многообразие форм реализации одноименных методов в цепочке наследуемых классов.
7 Как реализуется механизм полиморфизма?
Механизм полиморфизма реализуется за счет перекрытие одноименных методы базовых классов.
8 Как называется метод, объект для которой определяется во время работы программы?
Такой метод называется виртуальным.
9 Какой класс в языке C# называется абстрактным базовым классом?
Класс, для которых создание объектов невозможно.
10 Что представляет собой таблица виртуальных методов?
Специальная таблица, в которой находятся все виртуальные методы и адреса их точек входа.
ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСОВ
1 Понятие интерфейса?
Интерфейс представляет собой полностью абстрактный класс, все методы которого абстрактны.
2 В чем отличие наследования интерфейсного класса от обычного наследования?
Класс, наследующий интерфейс (интерфейсный класс), обязан полностью реализовать все методы интерфейса.
3 В чем отличие интерфейсного класса от обычного класса?
Для интерфейсного класса в C# разрешено множественное наследование.
4 Почему в интерфейсных класса возможно появление конфликтов в названиях имен?

Продолжение приложения А

Конфликт имен может проявляться при множественном наследовании, потому что разные родительские интерфейсы могут содержать одноименные методы с одинаковым синтаксисом.
5 Как можно решать конфликт имен при множественном наследовании?
Один из способов решения конфликта имен при множественном наследовании заключается в склеивание методов .
6 В чем суть склеивание методов?
В интерфейсном классе у всех одноименных методов должна быть одинаковая программная реализация, которая объявляется как единственная для всех одноименных методов своих родителей.
7 В чем суть переименования методов?
Это один из способов решения конфликта имен при множественном наследовании, при этом одноименных методов родительских интерфейсов переименовываются. Для каждого «нового» метода записывается своя реализация.
8 В чем преимущество интерфейсов?
Интерфейсы задают дополнительные свойства классу. Каждый интерфейс наделяет класс тем или иным новым свойством.
9 Какой тип могут иметь интерфейсы?
У интерфейсов нет типов.
10 Какое служебное слово используется при объявлении интерфейса?
При объявлении интерфейса перед именем класса необходимо использовать служебное слово interface.
КОМПОЗИЦИЯ И КОЛЛЕКЦИЯ КЛАССОВ
1 Понятие композиции классов?
Если некоторый класс, в своих полях данных, использует объекты другого класса, то такое объединение классов называется композицией.
2 В чем основная цель композиции классов?
Композиция классов это один из способов повторного использования ранее написанных фрагментов программ.
3 Понятие коллекции классов?
Объединение однотипных объектов в одной структуре данных называется коллекцией классов.
4 Какая объединяющая структура обычно используется для коллекционирования однотипных объектов?
Класс, который называется коллекцией.
5 Какие коллекции относятся к коллекциям с прямым доступом?
Это коллекции, доступ к элементам которой возможен по номеру элемента, например, массивы.
5 Какие коллекции относятся к коллекциям с последовательным доступом?

Это коллекции, для доступа к элементам которой необходимо выполнять последовательный поиск, например, стеки, очереди, списки.
6 Какие коллекции относятся к иерархическим коллекциям?
В иерархических коллекциях данные организованы в виде различных «древовидных» структур, например, в виде бинарного дерева или иерархической системы классификации.
7 Объекты какого типа могут включаться в коллекцию класса ArrayList?
Коллекция класса ArrayList предназначена для хранения объектов произвольного типа.
8 Какое свойство класса ArrayList хранит текущую емкость коллекции?
Это свойство целого типа Capacity.
9 Какое свойство класса ArrayList хранит текущую длину коллекции?
Это свойство целого типа Count.
10 Какой метод класса ArrayList позволяет включать объект в коллекцию на нужное место?
Это метод Insert (int Indx, Object Value);, который вставляет элемент Value на нужное место Indx коллекции.
ДЕЛЕГАТЫ
1 Понятие делегата?
Делегат это специальный класс, предназначенный для хранения ссылок на методы.
2 Какой класс называется функциональным типом?
Класс, позволяющий описать некоторое множество объектов, каждый из которых является функцией (или ссылкой на функцию), называется функциональным типом.
3 Что является экземпляром класса типа делегат?
Экземплярами такого класса являются ссылки на функции (методы) – им также как переменным выделяются места в памяти компьютера, начальный адрес которых являются «точками» входа в функции и передаются ссылками.
4 Какое служебное слово используется при объявлении делегата?
При объявлении интерфейса перед именем класса необходимо использовать служебное слово delegate.
5 Какой метод или функция может использоваться в качестве параметра вызова делегата?
Любая функция или метод, описание которой соответствует описанию делегата, может использоваться в качестве параметра вызова делегата.
6 Когда формируются ссылки на параметры вызова делегата?
Ссылки на параметры вызова делегата формируются во время работы программы (динамически).
7 Когда совместимы типы делегатов?
Типы делегатов всегда несовместимы друг с другом, даже если их форматы записей совпадают.

Продолжение приложения А

8 Когда совместимы экземпляры делегатов?
Если для объявления делегатов использован один и тот же тип, то имеет смысл говорить о совместимосте (равенстве) экземпляров делегатов.
9 Какие абстрактные классы платформы .NET можно наследовать при создании делегатов?
В стандартной системе типов (CTS) платформы .NET есть абстрактные классы System.Delegate и System.MulticastDelegate, которые можно использовать в качестве базовых при создании делегатов.
10 Понятие многоадресного делегата?
Для делегатов существует понятие многоадресный делегат – делегат, способный указывать на любое количество функций. Поскольку все делегаты языка C# являются производными классами от System.MulticastDelegate, то любой делегат C# потенциально является многоадресным.
СОБЫТИЯ
1 Понятие события?
События класса это специальные методы, позволяющие классу реагировать на действия пользователя или на определенные изменения в программе.
2 Где находятся заготовки обработчиков событий на действия пользователя для элементов управления?
Для многих действий пользователя разработаны заготовки обработчиков событий, которые можно посмотреть в окне Properties на странице Events.
3 Как обычно называются события класса?
Обработчик сообщений.
4 Что произойдет, если дважды «Кликнуть» на кнопку, расположенную в окне нашей формы?
Будет сформирован обработчик этого события со следующим заголовком: private void button1_Click(object sender, EventArgs e).
5 Что определяет первый формальный параметр всех обработчиков сообщений?
Он определяет отправителя сообщения, иногда говорят источник сообщения.
6 Что определяет второй формальный параметр обработчиков сообщений?
Второй формальный параметр шаблонов обработчиков событий является переменная типа объект точнее ссылкой на объект класса XXXEventArgs, который содержит связанные с событием параметры фактически это сообщение.
7 Где перечислены события, на которые должны реагировать элементы управления формы?
Для каждого элемента в файле Form1.Designer.cs, в разделе инициализации помимо определения свойств элементов управления, определяются

перечень событий, на которые должен реагировать этот элемент управления.

8 Как можно объединить все события класса?

Поскольку все методы обработчиков событий имеют единый формат записи, то их можно объединять (подписывать) с помощью многоадресных делегатов.

9 Где происходит добавление событий в многоадресный делегат формы?

Это происходит в методе InitializeComponent() с помощью операций присваивания, например,

```
this.MouseClick += new System.Windows.Forms.MouseEventHandler
 (this.Form1_MouseClick);
```

10 Где объявляется многоадресный делегат формы?

В файле Form1.Designer.cs, он находится «по умолчанию».

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

СПИСОК ЛИТЕРАТУРЫ

1 В.В. Фаронов Создание приложений с помощью C# Руководство программиста. - М.: “Эксмо”, 2008г.

2 Т.А. Павловская C#, Программирование на языке высокого уровня. Учебник для вузов, СПб,: Питер, 2009г.

3 А.В. Фролов, Г.В. Фролов Визуальное проектирование приложений C#. Книга вышла в издательстве Кудиц-Образ

4 Д. Кнут. Искусство программирования для ЭВМ. Т.1./ Основные алгоритмы / - М.:Мир,1976.

5 А.Л. Фукс Лекции по дисциплинам «Основы алгоритмизации» и «ООП» для студентов Томского государственного университета