

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 работы 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:

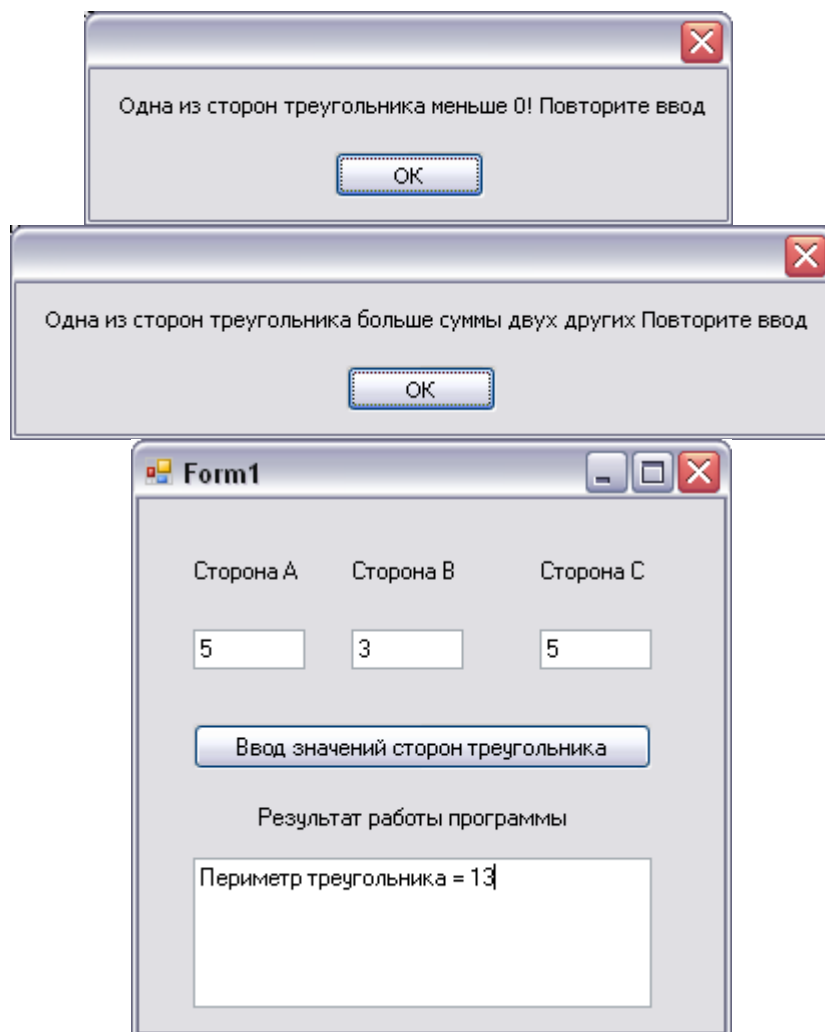


Рисунок 9.1 – Окна работающей программы

При появлении окна `MessageBox.Show()` одновременно в главном окне программы отображались 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 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 сокращаются 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. Защищенные элементы занимают промежуточное место между закрытыми и открытыми элементами. Если элемент является защищенным, объекты производного класса могут обращаться к нему, как будто он является открытым. Для оставшейся части программы защищенные элементы являются закрытыми. И если в программе возникает необходимость обратиться к защищенным 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 отрезка [" + 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 отрезка [" + 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:

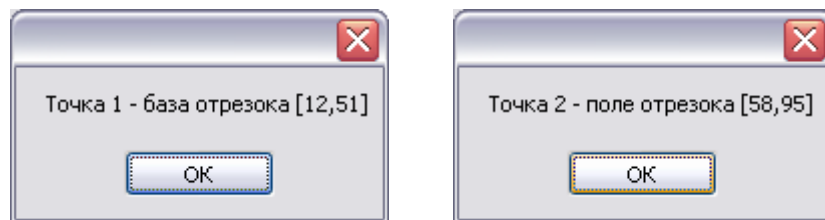


Рисунок 9.2 – Диалоговые окна координат концов отрезка

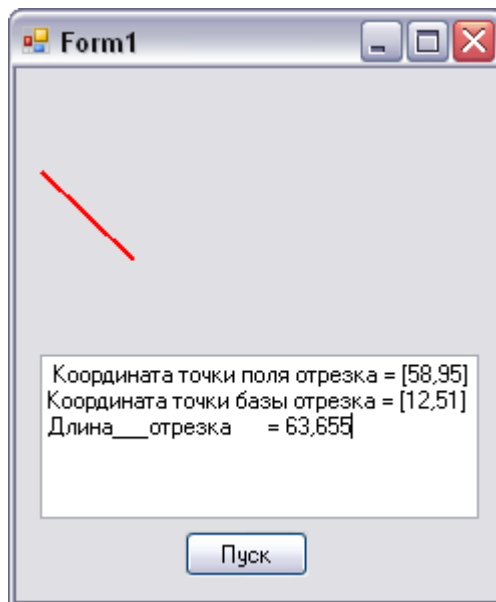


Рисунок 9.3 – Рабочее окно программы

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.