Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Ильшат Ринатович Мухаметзянов
Должность: директор
Дата подписания: 13.07.2023 12:35:18
Уникальный программный ключ:
aba80b84033c9ef196388e9ea0434f90a83a40954ba270e84bcbe64f02d1d8d0

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования «Казанский национальный исследовательский технический
университет им. А.Н. Туполева-КАИ»
(КНИТУ-КАИ)
Чистопольский филиал «Восток»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ
по дисциплине
ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Индекс по учебному плану: **Б1.О.12.03**
Направление подготовки: **09.03.01 Информатика и вычислительная техника**
Квалификация: **Бакалавр**
Профиль подготовки: **Вычислительные машины, комплексы, системы и сети**
Типы задач профессиональной деятельности: **проектная, производственно-технологическая**

Рекомендовано УМК ЧФ КНИТУ-КАИ

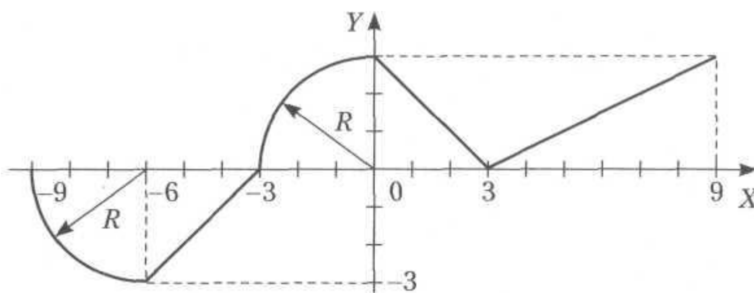
Чистополь
2023 г.

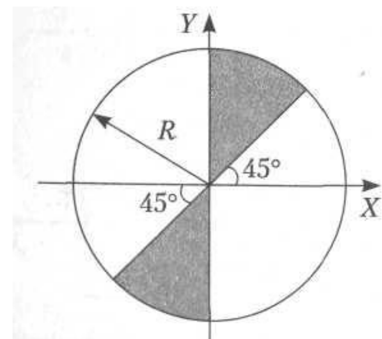
Практическая работа №1

Описание классов в C#

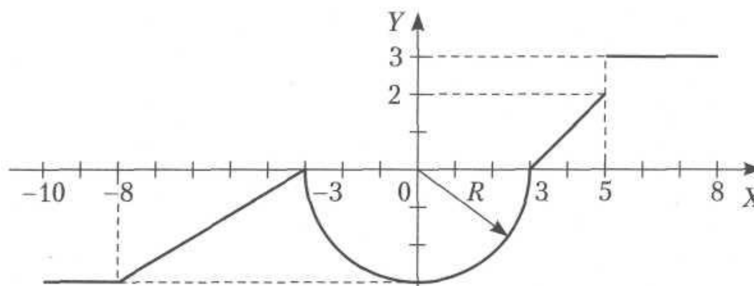
Написать программу на C#, которая определяет, попадает ли точка с заданными координатами в область, закрашенную на рисунке серым цветом. Результаты работы программы вывести в виде текстового сообщения. Разработать классы, описывающие представленные на графиках задания математические объекты.

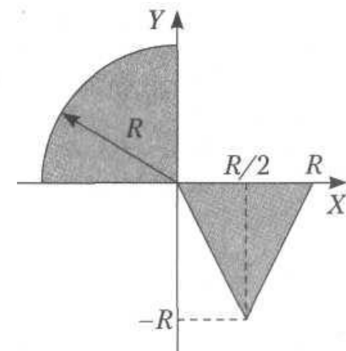
Вариант 1.



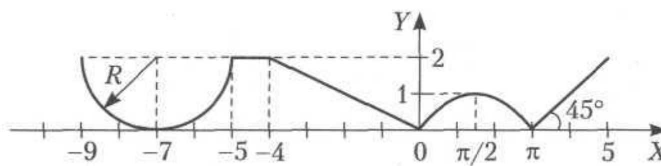


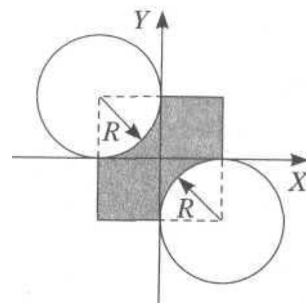
Вариант 2.



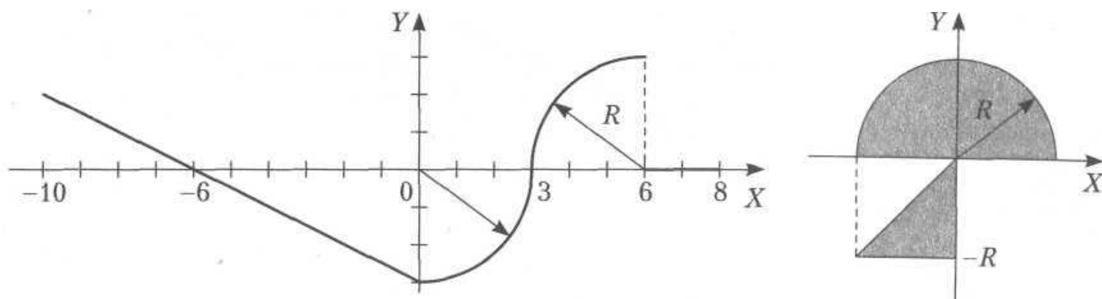


Вариант 3.

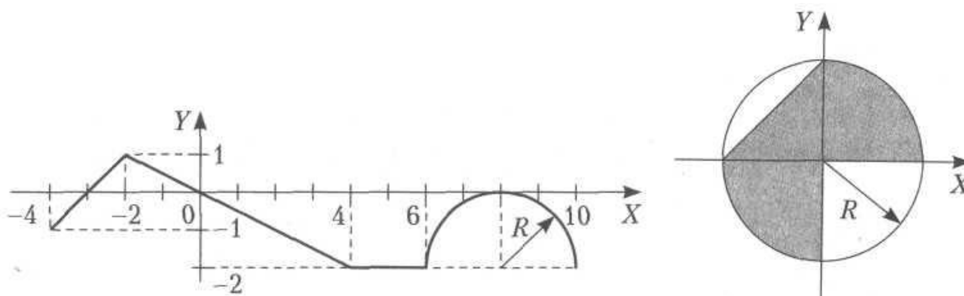




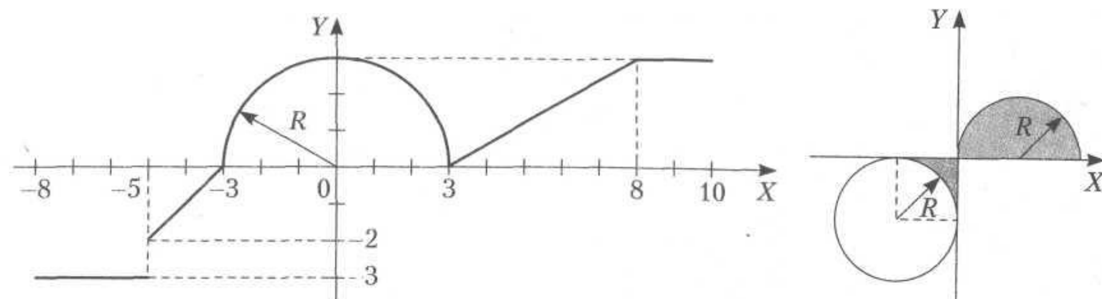
Вариант 4.



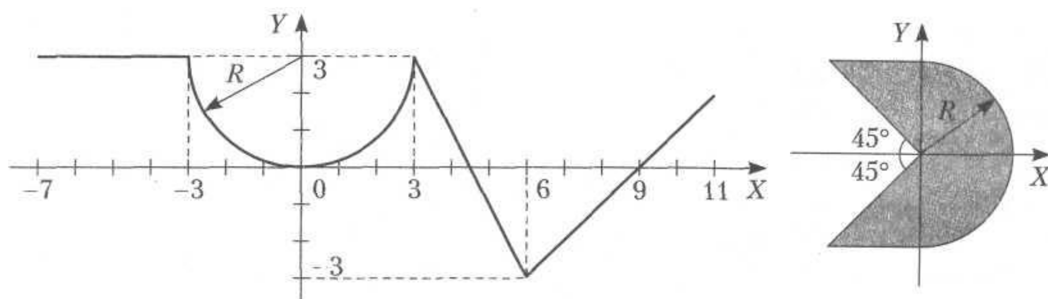
Вариант 5.



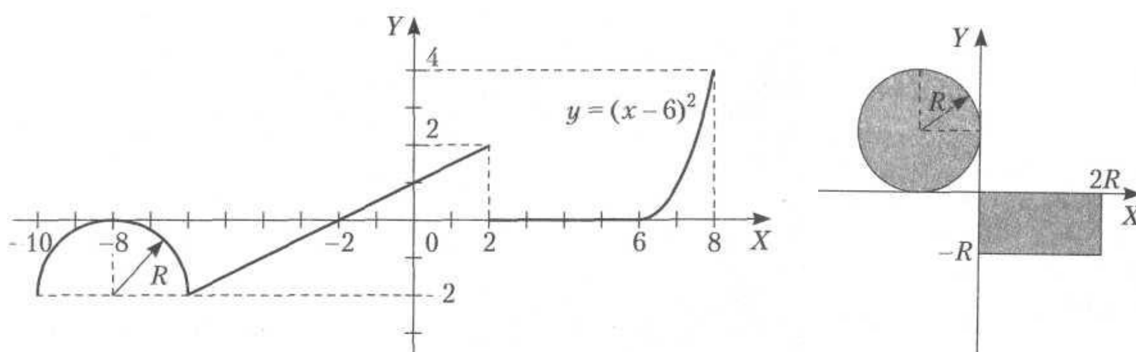
Вариант 6.



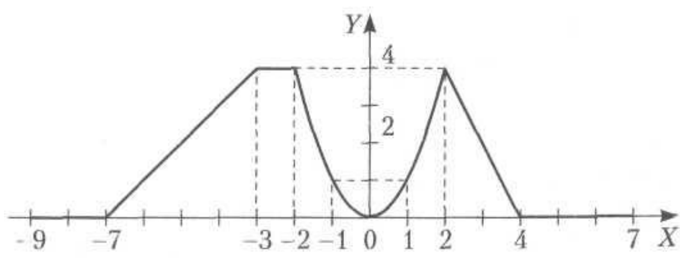
Вариант 7.

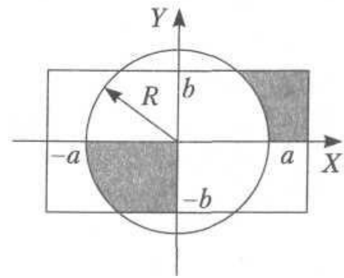


Вариант 8.

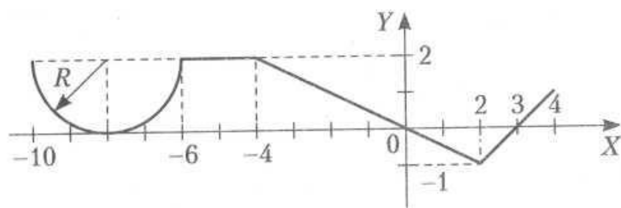


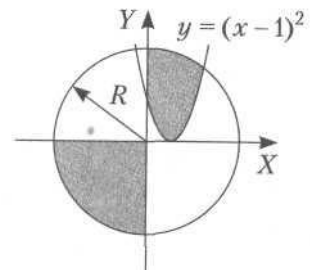
Вариант 9.





Вариант 10.





Практическая работа №2

Наследование

Наследование является одним из принципов ООП.

Основы классификации и реализация механизмов повторного использования и модификации кода. Базовый класс задаёт общие признаки и общее поведение для классов-наследников.

Общие (наиболее общие) свойства и методы наследуются от базового класса, в дополнение к которым добавляются и определяются НОВЫЕ свойства и методы.

Таким образом, прежде всего, наследование реализует механизмы расширения базового класса.

Реализация принципов наследования на примере.

```
using System;
namespace Inheritance_1
{
public class A
{
public int val1_A;
public void fun1_A (String str)
 {
Console.WriteLine("A's fun1_A:" + str);
 }
}

public class B:A
{
public int val1_B;
public void fun1_B (String str)
 {
```

```
Console.WriteLine("B's fun1_B:" + str);
 }
}

class Class1
{
static void Main(string[] args)
 {
B b0 = new B();
// От имени объекта b0 вызвана собственная функция fun1_B.
b0.fun1_B("from B");
// От имени объекта b0 вызвана унаследованная от класса A функция fun1_A.
b0.fun1_A("from B");
 }
}
}
```

Наследование и проблемы доступа

Производный класс наследует от базового класса ВСЁ, что он имеет. Другое дело, что воспользоваться в производном классе можно не всем наследством.

Добавляем в базовый класс частные члены:

```
 public class A
{
public int val1_A = 0;
public void fun1_A (String str)
 {
Console.WriteLine("A's fun1_A:" + str);
this.fun2_A("private function from A:");
 }
```

```
// При определении переменных
// в C# ничего не происходит без конструктора и оператора new.
// Даже если они и не присутствуют явным образом в коде.
private int val2_A = 0;
private void fun2_A (String str)
        {
Console.WriteLine(str + “A’s fun2_A:” + val2_A.ToString());
        }
}
```

И объект-представитель класса B в принципе НЕ может получить доступ к частным данным-членам и функциям-членам класса A. Косвенное влияние на такие данные-члены и функции-члены лишь через публичные функции класса A.

Следует иметь в виду ещё одно важное обстоятельство.

Если упорядочить все (известные) спецификаторы доступа C# по степени их открытости:

Максимальная степень открытости		Минимальная степень открытости
public	…	private

то наследуемый класс не может иметь более открытый спецификатор доступа, чем его предок.

```
protected class A
{
::::::::
```

```
}

public class B:A // Неправильное наследование.
{
:::::::::
}
```

Используем ещё один спецификатор доступа – protected. Этот спецификатор обеспечивает открытый доступ к членам базового класса, но только для производного класса!

```
public class A
{
:::::::::
protected int val3_A = 0;
}

public class B:A
{
:::::::::
public void fun1_B (String str)
 {
:::::::::
this.val3_A = 125;
 }

}

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

:::::::::

//b0.val3_A = 125; // Это член класса закрыт для внешнего использования!

}

Защищённые члены базового класса доступны для ВСЕХ прямых и косвенных наследников данного класса.

И ещё несколько важных замечаний относительно использования спецификаторов доступа: в C# структуры НЕ поддерживают наследования. Поэтому спецификатор доступа protected в объявлении данных-членов и функций-членов структур НЕ ПРИМЕНЯЕТСЯ, спецификаторы доступа действуют и в рамках пространства имён (поэтому и классы в нашем пространстве имён были объявлены со спецификаторами доступа public). Но в пространстве имён явным образом можно использовать лишь один спецификатор – спецификатор public, либо не использовать никаких спецификаторов.

Разработать систему классов, с учетом наследования базовых свойств геометрических объектов, необходимых для реализации программы на C# по вариантам.

Вариант 1.

Два выпуклых многоугольника заданы на плоскости перечислением координат вершин в порядке обхода границы. Определить площади многоугольников и проверить, лежит ли один из них строго внутри другого.

Вариант 2.

Из заданного на плоскости множества точек выбрать три различные точки так, чтобы разность между площадью круга, ограниченного окружностью, проходящей через эти три точки, и площадью треугольника с вершинами в этих точках была минимальной.

Даны два множества точек на плоскости. Выбрать три различные точки первого множества так, чтобы круг, ограниченный окружностью,

проходящей через эти три точки, содержал все точки второго множества и имел минимальную площадь.

Вариант 3.

Даны два множества точек на плоскости. Выбрать четыре различные точки пер-вого множества так, чтобы квадрат с вершинами в этих точках накрывал все точ-ки второго множества и имел минимальную площадь.

Вариант 4.

Даны два множества точек на плоскости. Выбрать три различные точки первого множества так, чтобы треугольник с вершинами в этих точках накрывал все точки второго множества и имел минимальную площадь.

Вариант 5.

Даны два множества точек на плоскости. Найти радиус и центр окружности, проходящей через η (и>=3) точек первого множества и содержащей строго внутри себя равное число точек первого и второго множеств.

Вариант 6.

Даны два множества точек на плоскости. Из первого множества выбрать три различные точки так, чтобы треугольник с вершинами в этих точках содержал (строго внутри себя) равное количество точек первого и второго множеств.

Вариант 7.

На плоскости заданы множество точек М и круг. Выбрать из М две различные точки так, чтобы наименьшим образом различались количества точек в круге, лежащие по разные стороны от прямой, проходящей через эти точки.

Вариант 8.

Дано 3 N точек на плоскости, причем никакие три из них не лежат на одной прямой. Построить множество N треугольников с вершинами в этих точках так, чтобы никакие два треугольника не пересекались и не содержали друг друга.

Вариант 9.

Выбрать три различные точки из заданного множества точек на плоскости так, чтобы была минимальной разность между количествами точек, лежащих внутри и вне треугольника с вершинами в выбранных точках.

Практическая работа №3

Делегаты

В C# существует возможность введения в базовом классе методов, а их реализацию оставить «на потом». Такие методы называют **абстрактными**. Абстрактный метод автоматически является и виртуальным, но писать это нельзя. Класс, в котором имеется хотя бы один абстрактный метод, тоже называется **абстрактным** и такой класс может служить только в качестве базового класса. Создать объекты абстрактных классов невозможно, потому что там нет реализации абстрактных методов. Чтобы класс- наследник абстрактного класса не был, в свою очередь, абстрактным (хотя и это не запрещено), там должны содержаться переопределения всех наследованных абстрактных методов.

Модифицируем приведенный выше пример.

```
namespace Virtual1
{
    abstract class Shape
    {  /* Создается абстрактный класс, наличие abstract обязательно!   */
    public int a,h;
        public Shape (int x,int y)
        {
        a=x;
        h=y;
        }
        public abstract  void Show_area();
                            //  реализация не нужна – метод абстрактный,
                              // фиксируется лишь интерфейс метода
    }
    class Class1
    {
```

```
static void Main(string[] args)
{
 Shape q; //указатель на абстрактный класс
// q=new Shape(4,6);   это ошибка, нельзя создать объект
//                                        абстрактного класса!
 q = new Tri(10,20);
 q.Show_area();
 q = new Square(10,20);
 q.Show_area();
    Console.ReadLine();
}     }     }
```

Указатель на абстрактный класс может указывать на любой класс-наследник.

Делегат — это объект, который может ссылаться на метод. Во время выполнения программы один и тот же делегат можно использовать для вызова различных методов, просто заменив метод, на который ссылается этот делегат. Таким образом, метод, который будет вызван делегатом, определяется не в период компиляции программы, а во время ее работы. Делегат в C# соответствует указателю на функцию в C++.

Общий вид объявления делегата:

delegate тип_возвращаемого_значения имя_делегата (список_формальных_параметров);

Для использования делегата должен быть объявлен указатель на делегата

Имя_делегата указатель_на_делегата;

и этот указатель должен быть инициирован:

указатель_на_делегата = new имя_делегата (имя_функции);

Здесь используется только имя функции (параметры не указываются).

Рассмотрим использование делегата на нескольких примерах.

<u>Пример 1</u>

Дополним программу следующей строкой:

```
namespace Virtual1
{
        delegate void del1();   // объявление  делегата del1
//  далее следует программа из раздела 3.12
}
```

Объявленному делегату del1()могут соответствовать **только** функции без формальных параметров и без возвращаемого значения.

Главная функция будет иметь вид:

```
static void Main(string[] args)
            {
        Shape q, r;
        q = new Tri(10,20);
        del1 f1 = new del1(q.Show_area);
// объявим переменную типа делегат f1 и поставим ей в соответствие
// функцию Show_area из объекта q класса  Tri
//
        r = new Square(10,20);
        del1 f2 = new del1(r.Show_area);
// объявим переменную типа делегат f2 и поставим ей в соответствие
// функцию Show_area из объекта  r класса  Square
        f1();  // идентичен вызову q.Show_area();
        f2();  // идентичен вызову r.Show_area();
```

```
Console.ReadLine();
        }
```

Примечание. В данной функции переменная r лишняя, вместо нее можно было использовать q.

В этом примере переменная типа «делегат» лишь заменяет имя функции, при этом фиксируется объект, к которому относится представляемая функция.

Пример 2

Делегаты позволяют использовать имя функции в качестве формальных параметров.

```
namespace Deleg
{
    delegate double fun1(double x); //объявление делегата
    class Test
    {
    protected double []x;
    protected double y=0;
        protected int n;
        public Test()
        {
            n=5;
            x=new double[n];
            for(int i=0;i<x.Length;i++)
            {
                Console.Write("X["+i+"]=");
                x[i]=Convert.ToDouble(Console.ReadLine());
            }
```

```
        }
            public  void gg(fun1 ff)
                        // формальный параметр – функция
            {
                for(int i=0;i<x.Length;i++)

                y+=(double)ff(x[i]);
// использование формального параметра – функции
            Console.WriteLine("Summa={0:##.###}",y);
            }
        public static double w1(double p)
        {return Math.Sin(p);}
        public  double w2(double p)
        {return Math.Log(p);}
    }
    class Class1
    {
        static void Main(string[] args)
        {
            Test tt=new Test();
                //   объявление переменной типа класс Test
            fun1 f1=new fun1(Test.w1);
// объявление переменной f1 типа делегат fun1, функция  w1
// статическая,  поэтому на нее можно ссылаться через имя класса Test
            tt.gg(f1);
```

// использование функции в качестве фактического параметра

```
                    fun1 f2;
          f2=new fun1(tt.w2);
```

// объявление переменной f1 типа делегат fun1, на функцию w2

// можно ссылаться только через имя объекта tt

```
                    tt.gg(f1);
```

// использование функции в качестве фактического параметра

```
                    Console.ReadLine();
              }             }          }
```

Делегату fun1 могут соответствовать только функции, имеющие тип возвращаемого значения double и один формальный параметр типа double.

Пример 3. Многоадресный делегат. Одному делегату можно ставить в соответствие несколько функций. В таком случае они будут выполнены в такой последовательности, как они были прикреплены к делегату.

```
namespace Deleg_2
{
delegate int Deleg(ref string st);      //  объявим делегат
       class Class1
       {
              public static int met1(ref string x)
              {
              Console.WriteLine("I am Metod 1");
              x+=" 11111";
     return 5;
```

```
        }
        public static int met2(ref string y)
        {
        Console.WriteLine("I am Metod 2");
                y+=" AAAAA";
    return 55;
        }
        static void Main(string[] args)
        {
                Deleg d1;
    int k;
                string r="******";
                Deleg d2=new Deleg(met1);
                   // связываем делегат и функцию
                Deleg d3=new Deleg(met2);
                d1=d2;   // присоединим первый делегат
                d1+=d3;  // добавим второй делегат
                k=d1(ref r);
                Console.WriteLine(r);
      Console.WriteLine("k=" + k);
                Console.ReadLine();
                }                }            }
```

В качестве ответа получим:

I am Metod 1

I am Metod 2

****** 11111 AAAAA

K=55

Как видно из примера, в случае, когда делегат имеет возвращаемое значение (в нашем случае int), значением многоадресного делегата будет значение, возвращенное последней функцией (в нашем случае met2).

Для исключения функции из многоадресного делегата необходимо писать d1-=d3; (вычитать удаляемый делегат, в данном случае d3).

Подведем итоги: делегаты расширяют знакомые нам средства программирования в двух случаях:

делегаты как указатели на функцию. Это позволяет использовать функции в качестве формальных/фактических параметров других функций;

многоадресные делегаты Таким образом, получим возможность одним вызовом обеспечить выполнение ряда функций.

Использование делегата в роли псевдонима функции может иногда уменьшить объем наших записей , но не расширяет наши возможности.

На основе делегатов построено еще одно важное средство C#: **событие** (event). Событие — это автоматическое уведомление о выполнении некоторого действия. События работают следующим образом. Объект, которому необходима информация о некотором событии, регистрирует обработчик для этого события. Когда ожидаемое событие происходит, вызываются все зарегистрированные обработчики. Обработчики событий представляются делегатами. События — это члены класса, которые объявляются с использованием ключевого слова event. Наиболее распространенная форма объявления события имеет следующий вид:

event событийный_делегат объект;

Здесь элемент *событийный_делегат* означает имя делегата, используемого для поддержки объявляемого события, а элемент объект — это имя создаваемого событийного объекта.

Пример 4

```
namespace Event1
{
        delegate void MyEvent(); //объявим делегат события
        class My
        {   //  класс события
        public event MyEvent activate; //объявим событие activate
                public void fire()
                {
                if(activate!=null)activate();
                }
        }
        class Demo
        {
                static void handler()
                { // функция – обработчик события
                Console.WriteLine("Что-то случилось . . . ");
                }
                static void Main(string[] args)
                {
                        My evt=new My();
                        evt.activate+=new MyEvent(handler);
                          // метод handler регистрируется в качестве обработчика
события
                        evt.fire();
                        Console.ReadLine();
                }       }           }
```

Все события активизируются посредством делегата. Следовательно, событийный делегат определяет сигнатуру для события. В данном случае параметры отсутствуют, однако событийные параметры разрешены. Затем создается класс события My. При выполнении следующей строки кода, принадлежащей этому классу, объявляется событийный объект MyEvent. Кроме того, внутри класса My объявляется метод fire(), который в этой программе вызывается, чтобы сигнализировать о событии (другими словами, этот метод вызывается, когда происходит событие). Как показано в следующем фрагменте кода, он вызывает обработчика события посредством делегата if(activate!=null)activate();

Обратите внимание на то, что обработчик события вызывается только в том случае, если делегат activate не равен null-значению. Поскольку другие части программы, чтобы получить уведомление о событии, должны зарегистрироваться, можно сделать так, чтобы метод fire () был вызван до регистрации любого обработчика события. Чтобы предотвратить вызов null-объекта, событийный делегат необходимо протестировать и убедиться в том, что он не равен null-значению. Внутри класса Demo создается обработчик события handler (). В этом примере обработчик события просто отображает сообщение, но ясно, что другие обработчики могли бы выполнять более полезные действия. В методе Main() создается объект класса My, а метод handler() регистрируется в качестве обработчика этого события. Обратите внимание на то, что обработчик добавляется в список с использованием составного оператора "+=". Следует отметить, что события поддерживают только операторы "+=" и " –= ".

Подобно делегатам события могут предназначаться для многоадресной передачи. В этом случае на одно уведомление о событии может отвечать несколько объектов.

<u>Пример 5</u>

```
namespace Events2
{
    delegate void MyEvent();
//      определение делегата, на основе которого будет определено событие
    class My
    {
```

```
        public event MyEvent activate;  //определение события
        public void fire()
        {
                if (activate!=null) activate();
        }
}
class X
{ // первый обработчик
        public void Xhandler()
{
Console.WriteLine("I am X");
}
}
class Y
{
        public void Yhandler()
        {    // второй обработчик
        Console.WriteLine("I am Y");
        }
}
class Class1
{
        static void handler()
        {   // третий обработчик, функция статическая
        Console.WriteLine("I am base");
```

```
        }
        static void Main(string[] args)
        {
            My evt =new My();
            X  x1= new X();
            Y  y1=new Y();
            evt.activate+=new MyEvent(handler);
// так можно писать, если функция-обработчик статическая
            evt.activate+=new MyEvent(x1.Xhandler);
            evt.activate+=new MyEvent(y1.Yhandler);
//если функция обычная, то ссылка на нее только через объект соответствующего класса
            evt.fire();
            Console.ReadLine();
        }        }        }
```

Практическая работа №4

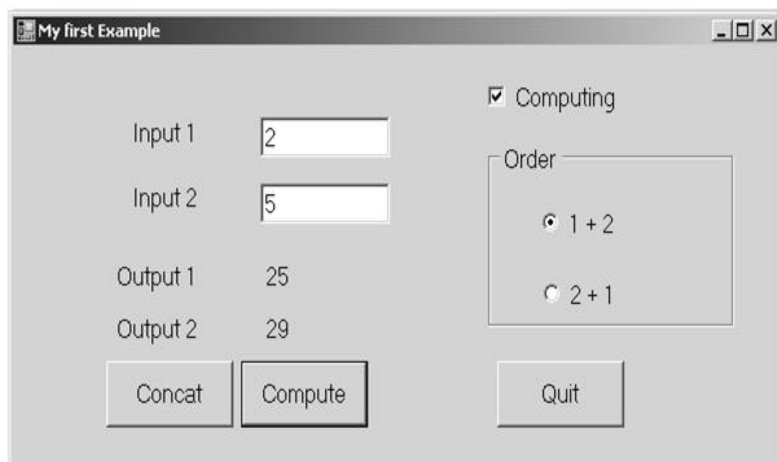
Простое приложение с обработкой сообщений

Рассмотрим как создавать интерфейсы пользователя, что такое свойства объектов и события. Цель – создать простейший пример, главная форма которого приведена на рис.

Для начала поменяем свойство формы Text на My first Example и увидим, что изменился заголовок формы.

Сначала ограничимся рассмотрением компонентов Label, TextBox и Button. Label (метка) предназначена для нанесения на форму пояснительных текстов и вывода результатов. Во время работы приложения невозможно редактировать содержимое меток, поэтому их рекомендуют использовать для вывода пояснительного текста и результатов. TextBox (строка редактирования) предназначена для ввода/вывода, тип данных в нем – всегда String и все преобразования должны выполняться программистом. Компоненту Button (командная кнопка) можно ставить

в соответствие функцию, которая будет выполнена при нажатии на кнопку. Все названные компоненты имеют среди других свойство Name – имя, по которому можно в программе ссылаться на него и Text, которое задает изображаемый на экране текст. Какие значения присвоены свойствам Text, видно из рисунка, значения свойства Name по умолчанию видны из приведенных текстов функций.



Реализации командных кнопок:

```
private void button3_Click(object sender, EventArgs e)
        { //   завершение работы приложения   Quit
           Close();
        }
private void button1_Click(object sender, EventArgs e)
        { //   сцепление введенных строк   Concat
           label5.Text = textBox1.Text + textBox2.Text;
        }
private void button2_Click(object sender, EventArgs e)
        { //  вычисления, нужны явные преобразования типов   Compute
           int i, j, k;
           i = Convert.ToInt32(textBox1.Text);
           j = Convert.ToInt32(textBox2.Text);
           k = 2 * i + 5 * j;
           label6.Text = Convert.ToString(k);  // 1
        }
```

Примечание. Здесь и в дальнейшем: жирным шрифтом показано то, что вставит программист; обычным то, что вставит среда.

Вопрос читателю: можно ли вместо строки // 1 писать

label6.Text = "" + k;   Почему?

4.2. Средства управления работой программы

Добавим на нашу форму кнопку выбора (CheckBox) и две радиокнопки (RadioButton). Напомним, что все кнопки выбора независимы друг от друга. Радиокнопки обычно объединяют в радиогруппы и из каждой группы в любой момент времени может быть выбрана одна и только одна радиокнопка.

Пусть наша единственная кнопка выбора имеет имя cB1 и показанное на рис. значение свойства Text. Назначение этой кнопки: вычисления

возможны лишь в случае, если она выбрана (свойство Checked имеет значение True).

Для создания радиогруппы необходимо сначала занести на форму компонент рамка – GroupBox (находится среди элементов управления Containers) и лишь **после этого** на него – требуемое количество (в нашем случае 2) радиокнопок. Пусть свойство Text для GroupBox имеет значение Order, свойство Name сохраняет значение по умолчанию. Для двух радиокнопок дадим свойству Name значения rB1 и rB2 соответственно. Значения их свойств Text видны на рис. 4.1. В нашем случае от выбора радиокнопки зависит очередность соединения введенных строк.

Поменяем реализацию двух командных кнопок:

```
private void button1_Click(object sender, EventArgs e)
        {  //   сцепление введенных строк   Concat
           if (rB1.Checked)
               label5.Text = textBox1.Text + textBox2.Text;
           if(rB2.Checked)
               label5.Text = textBox2.Text + textBox1.Text;
        }

        private void button2_Click(object sender, EventArgs e)
        {   //  вычисления Compute
           int i, j, k;
           if (cB1.Checked)
           {
              i = Convert.ToInt32(textBox1.Text);
              j = Convert.ToInt32(textBox2.Text);
              k = 2 * i + 5 * j;
              label6.Text = Convert.ToString(k);
              // label6.Text = "" + k;
           }
```

```
else
    MessageBox.Show("Режим вычислений не установлен "); // таким образом можно вывести сообщения пользователю
}
```

Практическая работа №5

Графика и перерисовка

В пространстве имен System.Drawing − определены основные структуры для представления:

- точки (координат) − Point и PointF
- размера − Size и SizeF
- прямоугольных областей − Rectangle и RectangleF.

Буква F в конце названия структуры означает, что, в отличие от обычных структур (без F), поля структуры имеют не целочисленные значения, а значения вещественного типа (float).

Структура Size

Структура Size предназначена для хранения ширины и высоты объекта и имеет, для этого, соответствующие открытые свойства Width и Height, доступные как для записи, так и для чтения. При создании объекта Size, с помощью конструктора по умолчанию:

```
Size sz = new Size();
```

свойства Width и Height устанавливаются в ноль.

Изменить значения свойств в последствии можно, например, следующим образом:

```
sz.Width = 40;
sz.Height = 60;
```

Структура содержит два конструктора с аргументами:

```
public Size(int, int);
public Size(Point);
```

Конструкторы с аргументами позволяют, в момент создания, инициализировать разные экземпляры структуры по-разному:

```
Size sz1= new Size(10,20);      // sz1.Width = 10, sz1.Height = 20
Size sz2 = new Size(15,50);     // sz2.Width = 15, sz2.Height = 50
```

Структура Point

Структура Point содержит открытые свойства X и Y целого типа, доступные, как для записи, так и для чтения.

Для создания точки “pt” можно использовать конструктор по умолчанию:

```
Point pt   = new Point();
```

Конструктор по умолчанию при создании точки обнуляет значения свойств X и Y.

В дальнейшем координаты точки можно изменить, например, следующим образом:

```
pt.X =25;
pt.Y=75;
```

Инициализировать новую точку класса Point, можно используя, конструкторы с аргументами:

```
public Point(Size);
public Point(int, int);
```

Например:

```
Point pt1 = new Point(10,20); // pt1.X =10, pt1.Y=20
Size sz = new  Size(27,45);
Point pt2 = new Point(sz);    // pt2.X=27, pt2.Y=45
```

Открытый метод структуры:

public void **Offset(** int *dx*, int *dy*)**;**

изменяет текущие координаты точки по формулам:

X=X+dx и Y=Y+dy;

Структура Rectangle

Структура предназначена для определения координат и размера прямоугольника. Структура содержит открытые свойства, часть из которых доступна только для чтения, а часть, как для чтения, так и для записи.

В структуре определены два конструктора с аргументами:

```
public Rectangle(
  int x,            // x-координата левого верхнего угла прямоугольника
  int y,            // y-координата левого верхнего угла прямоугольника
  int width,        // ширина прямоугольника
  int height        // высота   прямоугольника
);
public Rectangle(
  Point location,   // координата левого верхнего угла прямоугольника
  Size size         // размер прямоугольника
);
```

С помощью этих конструкторов можно определять размеры и местоположение прямоугольников при их создании:

```
Point pt =  new Point(10,15);
Size   sz = new Size (50,70);
Rectangle rct = new Rectangle(pt,sz);
Rectangle rect = new Rectangle(20,20,50,30);
```

Структура Rectangle содержит ряд доступных методов. Рассмотрим некоторые из них.

Метод:

```
public void Intersect(Rectangle);
```

Возвращает структуру, которая описывает прямоугольник, представляющий пересечение двух прямоугольников. Если не имеется никакого пересечения, все свойства структуры обнуляются.

Например:

```
Rectangle rect,rct;
rect = new Rectangle(20,25,50,55);
rct = new Rectangle(10,10,30,40);
rect.Intersect(rct);
```

выполнение, приведенного фрагмента кода установит значения свойства структуры прямоугольника rect следующим образом:

X=20, Y=25, Width=20, Height=25.

Метод:

public static Rectangle Union(Rectangle a, Rectangle b);

Возвращает структуру, описывающий минимальный по размерам прямоугольник, включающий в себя прямоугольники a и b.

Методы public void Offset(Point pos) и public void Offset(int x, int y) смещают координаты левой верхней точки прямоугольника по обеим осям на величину, задаваемую параметрами методов.

Представление цвета

Представление цвета осуществляется с помощью экземпляров структуры System.Drawing.Color.

Для задания цвета используется статический метод класса:

public static Color.FromArgb(int red, int green, int blue);

Параметры метода red, green и blue задают интенсивность красной, зеленой и синей составляющей цвета. Значение каждой компоненты цвета может изменяться в диапазоне от 0 до 255. Это позволяет отобразить 2^{24} различных цветов.

Для задания цвета можно также использовать один из перегруженных методов FromArgb:

```
public static Color FromArgb(int alpha, Color cr);
public static Color FromArgb(int alpha,  int red, int green, int blue);
```

Дополнительный параметр alpha задает степень прозрачности цвета. Чем меньше это число, тем меньше насыщенность цвета и тем более прозрачным является этот цвет. Значение параметра alpha может изменяться в диапазоне от 0 до 255. Значение 0 определяет полностью прозрачный (бесцветный), а значение 255 − полностью насыщенный (непрозрачный) цвет.

Структура Color содержит более 200 статических свойств, доступных только для чтения. Эти свойства задают именованные или, так называемые, Интернет − цвета, которые правильно воспроизводятся всеми WEB браузерами.

Например:

```
Color clr2 = Color.Beige;       // бежевый
Color clr3 = Color.Magenta;     // сиреневый
Color clr4 = Color.Orange;      // оранжевый
```

Кисти и перья

Графические объекты рисуются с помощью перьев и кистей.

Сплошные кисти создаются как экземпляры класса System.Drawing.SolidBrush, например:

```
Brush br2 = new SolidBrush(Color.Magenta);
Brush br3 = new SolidBrush(Color.FromArgb(200,10,120));
```

Параметр color конструктора public SolidBrush(Color color)

класса SolidBrush задает цвет сплошной кисти.

В классе System.Drawing.Brushes определено большое количество статических свойств, возвращающих кисть Интернет цветов. Создание таких кистей выглядит следующим образом:

```
Brush brr = Brushes.Orange;
```

В классе System.Drawing.Drawing2D.HatchBrush определены штриховые кисти.

Конструкторы класса:

public HatchBrush(HatchStyle hatchstyle, Color foreColor, Color backColor);

public HatchBrush(HatchStyle hatchstyle, Color foreColor);

Параметры:

foreColor − цвет штриха кисти;

backColor − цвет фонового штриха кисти (по умолчанию − черный цвет);

hatchstyle − стиль штриховой кисти.

Существует большое количество доступных стилей, наиболее типичными являются:

Cross − решетчатая кисть;

DiagonalCross − диагональная решетчатая кисть;

Horizontal − горизонтальная штриховка;

Vertical − вертикальная штриховка.

Например, создание кисти с вертикальной штриховкой синего цвета и фоновым штрихом бежевого цвета будет выглядеть следующим образом:

```
Brush br1 = new HatchBrush(HatchStyle.Vertical,Color.Blue,Color.Beige);
```

Перья описываются классом System.Drawing.Pen.

Конструкторы класса:

```
public Pen(Color color);
public Pen(Color color, float width);
public Pen( Brush brush);
public Pen(Brush brush, float width);
```

Параметры:

color − цвет пера;

width − толщина пера;

brush −кисть.

Примеры создания перьев:

```
Pen pn   = new Pen(Color. Magenta);
Pen pn1 = new Pen(Color.Orange,5);
Pen pn2 = new Pen(Brushes.Orange);
Pen pn3 = new Pen(Brushes.Magenta,10);
Pen pn4 = new Pen(Color.FromArgb(125,155, 0));
Pen pn5 = new Pen(Color.FromArgb(25,155,200),10);
```

В классе System.Drawing.Pens содержится множество статических свойств, описывающих перья с интернет цветом и толщиной в один пиксель. Создание таких перьев выглядит следующим образом:

```
Pen pn6 = Pens.Brown;
Pen pn7 = Pens.Magenta;
```

Рисование линий и фигур

Для вывода текстовой и графической информации в окно приложения необходимо использовать контекст устройства.

Контекст устройства в среде .NET инкапсулирован («завернут») в базовом классе System.Drawing.Graphics. Для создания объекта класса Graphics необходимо использовать метод CreateGraphics(), возвращающий ссылку на объект класса Graphics:

Graphics dc = CreateGraphics();

Полученный объект dc содержит контекст устройства, позволяющий осуществлять вывод информации в окно приложения.

Класс Graphics содержит множество методов, позволяющих рисовать различные графические фигуры. Рассмотрим некоторые из них.

Рисование контуров прямоугольников осуществляется с помощью методов:

public void DrawRectangle(Pen pen, Rectangle rect);

public void DrawRectangle(Pen pen, int x, int y, int width, int height);

public void DrawRectangle(Pen pen, float x, float y, float width, float height);

Рисование контуров эллипсов осуществляется с помощью методов:

public void DrawEllipse (Pen pen, Rectangle rect);

public void DrawEllipse (Pen pen, int x, int y, int width, int height);

public void DrawEllipse (Pen pen, float x, float y, float width, float height);

Рисование закрашенных эллипсов и прямоугольников осуществляется с помощью методов:

public void FillEllipse(Brush brush, Rectangle rect);

public void FillEllipse(Brush brush, int x, int y, int width, int height);

public void FillEllipse(Brush brush, float x, float y, float width, float height);

public void FillRectangle(Brush brush, Rectangle rect);

public void FillRectangle(Brush brush, int x, int y, int width, int height);

public void FillRectangle(Brush brush, float x, float y, float width, float height);

Параметры методов означают следующее:

pen – перо;

brush – кисть;

rect – прямоугольник;

x – координата x левого верхнего угла прямоугольника;

y – координата y левого верхнего угла прямоугольника;

width – ширина прямоугольника;

height –высота прямоугольника;

Рисование линий осуществляется с помощью перегруженных методов:

```
public void DrawLine(Pen pen, Point pt1, Point pt2);
public void DrawLine(Pen pen, PointF pt1, PointF pt2);
public void DrawLine(Pen pen, int x1, int y1, int x2, int y2);
public void DrawLine(Pen pen, float x1, float y1, float x2, float y2);
```

Параметры методов означают следующее:

pen – перо;

pt1 – начальная точка рисования;

pt2 – конечная точка рисования;

x1 и y1 – координаты начальной точки рисования;

x2 и y2 – координаты конечной точки рисования;

Примеры использоания функций:

```
dc.DrawRectangle(Pens.OrangeRed,5,10,25,45);
dc.DrawEllipse(Pens.Magenta,100,125,20,15);
dc.FillEllipse(Brushes.BlueViolet,45,50,20,15);
dc.DrawLine(Pens.Green,20,40,60,70);
```

Рисование текста

Для рисования текста используют перегруженный метод DrawString. Рассмотрим два из шести перегруженных методов DrawString:

```
public void DrawString(string s, Font fnt, Brush br, PointF pt);
```

public void DrawString(string s, Font fnt, Brush br, RectangleF ltRct);

Параметры:

s –строка символов,

fnt – шрифт текста,

br – кисть,

pt – точка, определяющая координаты вывода текста,

ltRct – прямоугольник, внутри которого выводится текст, если же текст не вмещается в область прямоугольника, то он (текст) обрезается.

Например:

Font fnt = new Font("Arial",10); //Шрифт Arial, размер 10

dc.DrawString("Привет!",fnt, Brushes.Green,10,20);

Перерисовка окна приложения

Если свернуть окно приложения, затем вновь развернуть его, то мы, к сожалению, заметим, что изображение на поверхности окна исчезло. Операционная система не восстанавливает содержимого окна. Восстановлением графики и текста должно заниматься само приложение. Операционная система в необходимых случаях вырабатывает сообщение (событие Paint), которое «говорит», что окно приложения не корректно и его необходимо перерисовать. Перерисовка окна должна происходить по событию Paint. Метод-обработчик этого события имеет заголовок:

private void Form_Paint(object sender, System.Windows.Forms.PaintEventArgs e)

Для этого метода нет необходимости создавать контекст устройства, он передается методу с помощью параметра e. Для получения контекста устройства необходимо выполнить следующую операцию:

Graphics dc = e.Graphics;

В теле этой функции необходимо выполнить все действия для перерисовки окна.

Очень часто перерисовка окна должна происходить в определенные моменты времени по инициативе приложения. Это бывает необходимо при выводе на экран анимации.

«Заставить» операционную систему выработать событие Paint можно путем вызова метода Invalidate(), который является членом класса System.Windows.Forms.Form. Существуют несколько перегруженных версий этого метода. Одна из них принимает в качестве параметра прямоугольник, который определяет область окна для перерисовки. Используемая нами версия без параметров перерисовывает все окно.

Практическая работа №6

Дочерние окно и элементы интерфейса

Вариант 1

Написать Windows-приложение, которое выполняет анимацию изображения.

Создать меню с командами Show picture, Choose, Animate, Stop, Quit.

Команда Quit завершает работу приложения. При выборе команды Show picture в центре экрана рисуется объект, состоящий из нескольких графических примитивов.

При выборе команды Choose открывается диалоговое окно, содержащее:

- поле типа TextBox с меткой Speed для ввода скорости движения объекта;
- группу Direction из двух переключателей (Up-Down, Left-Right) типа RadioButton для выбора направления движения;
- кнопку типа Button.

По команде Animate объект начинает перемещаться в выбранном направлении до края окна и обратно с заданной скоростью, по команде Stop — прекращает движение.

Вариант 2

Написать Windows-приложение, которое по заданным в файле исходным данным строит график или столбиковую диаграмму.

Создать меню с командами Input data, Choose, Line, Bar, Quit.

Команды Line и Bar недоступны. Команда Quit завершает работу приложения.

При выборе команды Input data из файла читаются исходные данные (файл сформировать самостоятельно).

По команде Choose открывается диалоговое окно, содержащее:

- список для выбора цвета графика типа ListBox;
- группу из двух переключателей (Line, Bar) типа RadioButton;

- кнопку типа Button.

Обеспечить возможность ввода цвета и выбора режима: построение графика (Line) или столбиковой диаграммы (Bar). После указания параметров становится доступной соответствующая команда меню.

По команде Line или Bar в главном окне приложения выбранным цветом строится график или диаграмма. Окно должно содержать заголовок графика или диаграммы, наименование и градацию осей. Изображение должно занимать все окно и масштабироваться при изменении размеров окна.

Вариант 3

Написать Windows-приложение, которое строит графики четырех заданных функций.

Создать меню с командами Chart, Build, Clear, About, Quit.

Команда Quit завершает работу приложения. При выборе команды About открывается окно с информацией о разработчике.

Команда Chart открывает диалоговое окно, содержащее:

- список для выбора цвета графика типа ListBox;
- список для выбора типа графика типа ListBox, содержащий четыре пункта: sin(x), sin(x+π/4), cos(x), cos(x–π/4);
- кнопку типа Button.

Обеспечить возможность выбора цвета и вида графика. После щелчка на кнопке OK в главном окне приложения строится график выбранной функции на интервале от -π/2 до +π/2. Окно должно содержать заголовок графика, наименование и градацию осей. Изображение должно занимать все окно и масштабироваться при изменении размеров окна.

Команда Clear очищает окно.

Вариант 4

Написать Windows-приложение — графическую иллюстрацию сортировки одномерного массива методом «выбора».

Создать меню с командами File, Animate, About, Exit.

Команда Animate недоступна. Команда Exit завершает работу приложение. Команда About открывает окно с информацией о разработчике. Для выбора файла исходных данных (команда File) использовать объект класса OpenFileDialog.

Из выбранного файла читаются исходные данные для сортировки (сформировать самостоятельно не менее трех файлов различной длины с данными целого типа).

После чтения данных становится доступной команда Animate.

При выборе команды Animate в главном окне приложения отображается процесс сортировки в виде столбиковой диаграммы. Каждый элемент представляется столбиком соответствующего размера. На каждом шаге алгоритма два элемента меняются местами. Окно должно содержать заголовок. Изображение должно занимать все окно.

Вариант 5

Написать Windows-приложение — графическую иллюстрацию сортировки одномерного массива методом пузырька.

Создать меню с командами File, Animate, About, Exit.

Команда Animate недоступна. Команда Exit завершает работу приложение. Команда About открывает окно с информацией о разработчике. Для выбора файла исходных данных (команда File) использовать объект класса OpenFileDialog.

Из выбранного файла читаются исходные данные для сортировки (сформировать самостоятельно не менее трех файлов различной длины с данными целого типа).

После чтения данных становится доступной команда Animate.

При выборе команды Animate в главном окне приложения отображается процесс сортировки в виде столбиковой диаграммы. Каждый элемент представляется столбиком соответствующего размера. На каждом шаге алгоритма два элемента меняются местами. Окно должно содержать заголовок. Изображение должно занимать все окно.

Литература:

1. Гуриков, С. Р. Введение в программирование на языке Visual C# : учеб. пособие / С.Р. Гуриков. — Москва : ФОРУМ : ИНФРА-М, 2019. — 447 с. — (Высшее образование: Бакалавриат). - ISBN 978-5-00091-458-8. - Текст : элек-тронный. - URL: https://znanium.com/catalog/product/1017998 (дата обращения: 29.06.2023). – Режим доступа: по подписке.

2. Павловская, Т. А. C#. Программирование на языке высокого уровня : учебник для вузов / Т. А. Павловская. - Санкт-Петербург : Питер, 2020. - 432 с. - (Серия «Учебник для вузов»). - ISBN 978-5-4461-0913-5. - Текст : электронный. - URL: https://znanium.com/catalog/product/1733745 (дата обращения: 29.06.2023). – Режим доступа: по подписке.

3. Хорев, П. Б. Объектно-ориентированное программирование с примерами на C# : учебное пособие / П.Б. Хорев. — Москва : ФОРУМ : ИНФРА-М, 2023. — 200 с. — (Высшее образование: Бакалавриат). - ISBN 978-5-00091-680-3. - Текст : электронный. - URL: https://znanium.com/catalog/product/1926392 (дата обращения: 29.06.2023). – Режим доступа: по подписке.

4. Гагарина Л.Г. Введение в архитектуру программного обеспечения : учеб. пособие / Л.Г. Гагарина, А.Р. Федоров, П.А. Федоров. — М. : ФОРУМ : ИНФРА-М, 2017. — 320 с. — Режим доступа: http://znanium.com/bookread2.php?book=615207

5. Разработка приложений на C# с использованием СУБД PostgreSQL / Ва-сюткина И.А., Трошина Г.В., Бычков М.И. - Новосиб.:НГТУ, 2015. - 143 с. — Ре-жим доступа: http://znanium.com/bookread2.php?book=556925