Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра информатики и информационных технологий

Л.Е. Потапова, Т.Г. Алейникова

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C#

Методические рекомендации для лабораторных работ

Витебск
ВГУ имени П.М. Машерова
2016

УДК 004.432(075.8)
ББК 32.973.22я73
П64

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 2 от 24.12.2015 г.

Авторы: доценты кафедры информатики и информационных технологий ВГУ имени П.М. Машерова, кандидаты физико-математических наук, доценты **Л.Е. Потапова, Т.Г. Алейникова**

Р е ц е н з е н т :
доцент кафедры математики и информационных технологий УО «ВГТУ», кандидат физико-математических наук *Т.В. Никонова*

Потапова, Л.Е.

П64 Объектно-ориентированное программирование на языке C# : методические рекомендации для лабораторных работ / Л.Е. Потапова, Т.Г. Алейникова. – Витебск : ВГУ имени П.М. Машерова, 2016. – 50 с.

Учебное издание содержит основной материал по объектно-ориентированному программированию на языке C#, вопросы и индивидуальные задания для лабораторных занятий и самостоятельной работы. Теоретические сведения проиллюстрированы большим количеством примеров с подробными комментариями.

Методические рекомендации предназначены для обучения программированию на языке C# и адресованы студентам специальностей «Математика и информатика» и «Компьютерная безопасность».

УДК 004.432(075.8)
ББК 32.973.22я73

СОДЕРЖАНИЕ

1. ПРОСТЕЙШИЕ КЛАССЫ. ИНКАПСУЛЯЦИЯ И СВОЙСТВА

Данный раздел посвящен усвоению понятия класса, определения членов класса, умению выполнять обработку исключительных ситуаций, овладению приемами разработки простейших программ в объектно-ориентированном стиле программирования.

1.1. Классы: основные понятия

Класс является типом данных, определяемым пользователем. Он должен представлять собой одну логическую сущность, например, являться моделью реального объекта или процесса. Элементами класса являются данные и функции, предназначенные для их обработки.

Описание класса имеет вид:

```
[атрибуты] [спецификаторы] class <имя_класса> [: предки]
{ тело_класса }
```

Обязательными являются только ключевое слово **class**, а также имя и тело класса. *Имя класса* задается программистом по общим правилам C#. *Тело класса* – это список описаний его элементов, заключенный в фигурные скобки. *Спецификаторы* определяют свойства класса, а также доступность класса для других элементов программы. Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. Для классов, которые описываются в пространстве имен непосредственно, допускаются только два спецификатора: **public** и **internal**. По умолчанию подразумевается спецификатор **internal**.

Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов, и используется для их создания. Объекты также называются экземплярами класса. Объекты создаются явным или неявным образом, то есть либо программистом, либо системой.

Класс относится к ссылочным типам данных, память под которые выделяется в хипе. Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные.

В классе могут присутствовать элементы, которые существуют в единственном экземпляре для всех объектов класса – статические элементы.

Функциональные элементы класса не тиражируются, то есть всегда хранятся в единственном экземпляре. Для работы со статическими данными класса используются статические методы, для работы с данными экземпляра – методы экземпляра, или просто методы.

1.1.1. Данные: поля и константы

Данные, содержащиеся в классе, могут быть переменными или константами. Переменные, описанные в классе, называются полями класса.

Синтаксис описания элемента данных:

```
[атрибуты][спецификаторы][const] <тип> <имя>
[= <начальное значение>]
```

По умолчанию элементы класса считаются закрытыми (**private**). Все методы класса имеют непосредственный доступ к его закрытым полям.

Все поля сначала автоматически инициализируются нулем соответствующего типа. После этого полю присваивается значение, заданное при его явной инициализации. Задание начальных значений для статических полей выполняется при инициализации класса, а обычных – при создании экземпляра.

Поля, описанные со спецификатором **static**, а также константы существуют в единственном экземпляре для всех объектов класса, поэтому к ним обращаются через имя класса. Обращение к полю класса выполняется с помощью операции доступа - точки:

для обычных полей: `<имя экземпляра>.<имя поля>`

для статических: `<имя класса>.<имя поля>`

Пример 1. Класс Demo, содержащий поля и константу.

```
using System;
namespace ConsoleApplicationl
{class Demo
  {public int a = 1;  //поле данных
   public const double c = 1.66; //константа
   public static string s = "Demo"; //статич. поле класса
  double y;     //закрытое поле данных }
  class Classl
  {static void Main()
  {Demo x = new Demo(); //создание экземпляра класса Demo
   Console.WrteLine(x.a); //обращение к полю класса
   Console.WriteLine(Demo.c); // обращение к константе
   Console.WriteLine(Demo.s); //обращение к статич. полю }}}
```

Поле y вывести на экран аналогичным образом нельзя – оно является закрытым, т.е. недоступно из класса Classl. Поскольку значение этому полю явным образом не присвоено, среда присваивает ему значение ноль.

1.2. Методы

Методы класса имеют непосредственный доступ к его закрытым полям. Метод, описанный со спецификатором **static**, должен обращаться только к статическим полям класса. Статический метод вызывается через имя класса, а обычный – через имя экземпляра.

Например, для статического метода класса `Class1` с сигнатурой `static void Poisk(int a, ref int b, out int c)` вызов может быть следующим: `Poisk(1, ref x, out y);`

Для метода с сигнатурой `void Out()` необходимо создать объект:

```
Class1 myob = new Class1();
```

а затем вызвать метод `myob.Out();`

1.3. Конструкторы

Конструктор предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции **new**. Формат записи конструктора такой:

```
[спецификатор] <имя_класса>()
{тело конструктора}
```

Обычно в качестве элемента *спецификатор* используется модификатор доступа **public**, поскольку конструкторы, как правило, вызываются вне их класса.

Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации. Конструктор, вызываемый без параметров, называется конструктором по умолчанию. Если в классе не указан ни один конструктор или какие-то поля не были инициализированы, полям значимых типов присваивается нуль соответствующего типа, полям ссылочных типов – значение **null**.

Создание объекта выполняется операцией

```
<имя_переменной_типа_класса> = new <имя_класса>();
```

Имя класса вместе со следующей за ним парой круглых скобок – это не что иное, как конструктор реализуемого класса. Если в классе конструктор не определен явным образом, оператор **new** будет использовать конструктор по умолчанию, который предоставляется средствами языка C#.

Пример 2. В программе создаются два объекта с различными значениями полей с помощью конструкторов по умолчанию и с параметрами.

```
using System;
namespace ConsoleApplication1
{class Demo
   {internal int a;
```

```
    internal double y;
     public Demo(int a,double y1) //конструктор с параметр.
     {this.a = a; //полю a присваиваем значение параметра a
       y = y1;}
     public Demo()// конструктор по умолчанию
     {a = 276;
       y = 5.5;}}
  class Class1
   {static void Main()
    {//вызов конструктора с параметрами
     Demo ob1 = new Demo(300, 0.002);
     Console.WriteLine(ob1.a); //результат: 300
     Console.WriteLine(ob1.y); //результат:0,002
     Demo ob2 = new Demo();//вызов конструктора без параметров
     Console.Writeline(ob2.a); //результат: 276
     Console.WriteLine(ob2.y); //результат: 5.5 }}}
```

1.4. Свойства

Свойство – это специальный тип членов класса, который предназначен для организации доступа к закрытым полям класса и определяет методы его получения и установки.

Формат записи свойства:

```
[атрибуты][спецификаторы] <тип> <имя_свойства>
{[ get {<код аксессора чтения поля>//код доступа}]
 [ set {<код аксессора записи поля>// код доступа}]}
```

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором **public**). После определения свойства любое использование его имени означает вызов соответствующего аксессора. *Код доступа* представляет собой блоки операторов, которые выполняются при получении (**get**) или установке (**set**) свойства. Может отсутствовать либо **часть get**, либо **set**, но не обе одновременно. Если отсутствует часть **set**, свойство доступно только для чтения (read-only), если отсутствует часть **get**, свойство доступно только для записи (write-only).

Аксессор **set** автоматически принимает параметр с именем **value**, который содержит значение, присваиваемое свойству.

Свойства не определяют область памяти. Следовательно, свойство управляет доступом к полю, но самого поля не обеспечивает.

Имя свойства можно использовать в выражениях и инструкциях присваивания подобно обычной переменной, хотя в действительности здесь будут автоматически вызываться **get**- и **set**-аксессоры.

Пример 3. Использование свойства. Это свойство позволяет присваивать полю только положительные числа.

```
using System;
class SimpProp {
  int prop; // Это закрытое поле управляется свойством myp.
  public SimpProp() { prop = 0 ; }
  public int myp   /* Это свойство поддерживает доступ к закрытому
  полю класса prop. Оно позволяет присваивать ему только положительные
  числа. */
   {get {return prop; } // способ получения свойства
    set {if(value >= 0) prop = value; }// способ установки
                                        //свойства }}
// Демонстрируем использование свойства
class PropertyDemo {
  public static void Main()
  {SimpProp ob = new SimpProp ();
  Console.WriteLine("Исходное значение ob.myp:"+ob.myp);
  ob.myp = 100; // вызывается метод установки свойства
  Console.WriteLine("Значение ob.myp: " + ob.myp);
  Console.WriteLine("Попытка присвоить -10 свойству
  ob.myp");ob.myp = -10;
  Console.WriteLine("Значение ob.myp: " + ob.myp);}}
```

Контрольные вопросы:

1. Как описываются классы в C#?
2. Что относится к членам класса?
3. Что такое статические члены класса?
4. Как используются статические и экземплярные методы?
5. Что представляет собой конструктор? Для чего он используется?
6. Какие бывают конструкторы?
7. Для чего предназначены свойства?

ЛАБОРАТОРНАЯ РАБОТА № 1. **Классы**

Задание 1. Создайте проект, в котором опишите класс для решения задачи Вашего варианта.

Указания. Разрабатываемый класс должен содержать следующие элементы: скрытые поля, конструкторы без параметров и с параметрами (имена некоторых полей должны совпадать с идентификаторами параметров), свойства, метод вывода полей и указанный в таблице метод.

Составьте тестирующую программу с выдачей результатов. В программе должна выполняться проверка всех разработанных элементов класса, вывод состояния объекта.

Варианты заданий:

№	**Класс**	**Метод**
1.	Сотрудник (поля: имя, p – минимальная зарплата)	Доход: k *p, где k – повышающий коэффициент
2.	Квартира (поля: номер, стоимость 1м^2, площадь)	Стоимость
3.	Агрополе (поля: название, вес r посеянных семян на единицу площади, S – площадь в га)	Урожай: k*r, где k- коэффициент, зависящий от культуры
4.	Стол (поля: название, площадь S в см)	Стоимость: $S^2/3+500000$
5.	Автобус (поля: количество пассажиров, стоимость билета)	Выручка
6.	Транспортное средство (поля: название, расстояние, цена за 1 км)	Стоимость проезда
7.	Пособие (поля: повышающий коэффициент k, минимальное пособие r)	Размер пособия: k*r
8.	Телефон (поля: марка, количество функций – k)	Стоимость: 40ln(k)
9.	Автомобиль (поля: марка, расход горючего на 100км N, расстояние R в км)	Объем горючего: N*R
10.	Одежда (поле – модель, ширина ткани H м, норма расхода – L м)	Расход ткани: (2-H)*L
11.	Постройка (поля: название, высота здания V м)	Высота фундамента: 0,03*V
12.	Аудитория (поля: номер, площадь S м^2)	Количество мест: [S/1,2]

2. МАССИВЫ. ИНДЕКСАТОРЫ

Цель: данный раздел посвящен усвоению основных приемов создания классов с полями структурированного типа - массив; формированию знаний о возможности использования индексаторов для доступа к элементам закрытых массивов.

2.1. Структура массива в C#

Массив – это коллекция переменных одинакового типа, обращение к которым происходит с использованием общего для всех имени.

Коллекция – это группа объектов. C# определяет несколько типов коллекций, и одним из них является массив.

Кроме рассмотренных в [11], C# позволяет создавать двумерный массив специального типа, именуемый рваным, или с рваными краями, или ступенчатым. У такого массива строки могут иметь различную длину.

Рваные массивы объявляются с помощью наборов квадратных скобок, обозначающих размерности массива. Например:

```
<тип>[][] <имя> = new <тип>[размер][];
```

Здесь элемент *размер* означает количество строк в массиве. Для самих строк память выделяется индивидуально, что позволяет строкам иметь разную длину.

Пример 1.

```
int [][] gg = new int [3][] ;
gg[0] = new int [4];
gg[1] = new int [3];
gg[2] = new int [5];
```

Доступ к элементу осуществляется посредством задания индекса внутри собственного набора квадратных скобок. Например:

```
gg[2][1] = 10;
```

Поскольку рваные массивы – по сути массивы массивов, то "внутренние" массивы (строки) могут иметь разный тип.

Можно присваивать одной ссылочной переменной массива другую. При этом не делается копия массива и не копируется содержимое одного массива в другой, а просто изменяете объект, на который ссылается эта переменная.

2.2. Цикл foreach

Цикл **foreach** используется для опроса элементов коллекции. Формат записи цикла имеет вид:

```
foreach (<тип> <имя_переменной> in <коллекция>)
<тело цикла;>
```

Здесь элементы *тип* и *имя_переменной* задают тип и имя итерационной переменной, которая при выполнении цикла **foreach** будет последовательно получать значения элементов из коллекции.

Элемент *коллекция* служит для указания опрашиваемой коллекции. Таким образом, элемент *тип* должен совпадать (или быть совместимым) с базовым типом массива. С помощью **foreach**, невозможно изменить содержимое коллекции.

Пример 2. Создать массив для хранения целых чисел и присвоить его элементам начальные значения. Затем вывести элементы массива, и попутно вычислить их сумму.

```
using System;
```

```
class ForeachDemo {// Использование цикла foreach.
public static void Main() {
 int sum = 0;
 int[] n = new int[10];
 for(int i = 0; i < 10; i++)
 n[i] = i;    //Присваиваем элементам массива n значения
 foreach(int x in n) //Используем цикл foreach для вывода
                     //значений элементов массива и их суммирования,
 {Console.WriteLine("Значение элемента равно:
 " + x);
 sum += x; }
 Console.WriteLine("Сумма равна: " + sum);}}
```

Цикл **foreach** работает и с многомерными массивами. В этом случае он возвращает элементы в порядке следования строк: от первой до последней.

2.3. Индексаторы

Индексатор предназначен для обращения к скрытому полю класса, представляющему собой массив, используя имя объекта и номер элемента массива в квадратных скобках.

Синтаксис индексатора:

```
<атрибуты><спецификаторы> <тип> this [<список_параметров>]
{
  get код_доступа
  set код_доступа
}
```

Спецификаторы аналогичны спецификаторам свойств и методов.

Здесь *тип* – базовый тип индексатора. Он соответствует базовому типу массива. *Код доступа* представляет собой блоки операторов, которые выполняются при получении (**get**) или установке значения (**set**) элемента массива. Может отсутствовать либо часть **get**, либо **set**. Если отсутствует часть **set**, индексатор доступен только для чтения (read-only), если отсутствует часть **get**, индексатор доступен только для записи (write-only). *Список параметров* содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу.

Пример 3. Создайте класс с закрытым массивом, элементы которого должны находиться в диапазоне [0, 100]. Кроме того, при доступе к элементу проверяется, не вышел ли индекс за допустимые границы. Используйте индексатор.

```
using System;
namespace ConsoleApplicationl
{class SafeArray
```

```
 {public bool error =false;  //открытый признак ошибки
  int[] a;                   //закрытый массив
  int length;                //закрытая  размерность
  public SafeArray(int size)  // конструктор класса
   {a = new int[size];
   length = size;}
   public int Length    // свойство - размерность
    {get { return length; }}
   public int this[int i]  // индексатор
    {get {
     if (i >= 0 && i < length) return a[i];
     else { error = true; return 0; }}
    set{
     if ( i >= 0 && i < length && value >= 0 &&
     value <= 100 ) a[i] = value;
     else error = true; }}}
class Classl
 {static void Main()
  {int n = 100;
   SafeArray sa = new SafeArray(n);//создание объекта
   for ( int i = 0; i < n; ++i )
   {sa[i] = i * 2;     // использование индексатора
    Console.Write ( sa[i] ); }//использование индексатора
  if (sa.error) Console.Write ("Были ошибки!"); }}}
```

Вообще говоря, индексатор не обязательно должен быть связан с каким-либо внутренним полем данных.

Язык C# допускает использование многомерных индексаторов. Они описываются аналогично обычным и применяются в основном для контроля за занесением данных в многомерные массивы и выборке данных из многомерных массивов, оформленных в виде классов.

Например, если внутри класса объявлен двумерный массив `int[,] a;`, то заголовок индексатора должен иметь вид:

```
public int this[int i, int j]
```

Контрольные вопросы:

1. Что понимается под массивом?
2. В каких случаях целесообразно описывать двумерный массив с помощью одномерных?
3. Для чего предназначен цикл foreach? Можно ли использовать этот цикл для ввода элементов массива?
4. Как определяется базовый тип индексатора?
5. Что записывается в качестве имени индексатора?
6. Что содержит список параметров индексатора?

ЛАБОРАТОРНАЯ РАБОТА № 2. **Массивы. Индексаторы**

Задание 1. Создайте проект, в котором опишите класс для решения задачи Вашего варианта.

Класс должен содержать закрытое поле двумерного динамического массива, конструктор без параметров и три конструктора с параметрами, свойства, индексаторы, методы (ввода, вывода, обработки массива). Обработку массива в соответствии с заданием варианта осуществлять в одном методе, исходные данные и результаты работы метода передавать параметрами. В программе должны проверятся все элементы разработанного класса.

Варианты заданий:

№	Тип массива	Размерность	Метод
1.	вещественный	NxN	Проверить является ли матрица симметричной.
2.	символьный из цифр	NxM	Рассматривая символы строк как числа, определить сумму четных и нечетных цифр в каждой строке.
3.	целочисленный	NxM	Найти суммы элементов, стоящих до и после введенного с клавиатуры значения
4.	вещественный	NxM	Найти количество столбцов, начинающихся с отрицательного числа.
5.	целочисленный	NxN	Проверить, является ли матрицы верхней треугольной.
6.	вещественный	NxN	Преобразовать матрицу следующим образом: каждый элемент строки разделить на максимальный элемент этой строки, если он не равна 0. В противном случае элементы строки оставить без изменений.
7.	строковый	NxN	Подсчитать, сколько элементов массива, начинаются с прописной буквы.
8.	целочисленный	Nx2	Массив состоит из номеров зачетной книжки и годов рождений студентов. Вычислить сколько лет студенту и вывести на экран информацию вида «номер зачетной книжки – количество лет».
9.	целочисленный	NxN	Установить, является ли упорядоченной заданная своим номером строка.
10.	строковый	NxM	Проверить, является ли заданный индексами элемент массива палиндромом. Палиндром принимает одно и то же значение при чтении его как справа налево, так и слева направо.
11.	целочисленный	NxM	Найти элемент в массиве по заданному значению. (его индексы)
12.	вещественный	NxM	Найти в заданном своим номером столбце максимальный по модулю элемент.

3.ПЕРЕГРУЗКА МЕТОДОВ. ПЕРЕГРУЗКА ОПЕРАЦИЙ

Цель: освоить приемы создания операций класса и их использование.

Использование нескольких методов с одним и тем же именем, но различными типами параметров и их способами передачи называется *перегрузкой методов.*

Компилятор определяет, какой именно метод требуется вызвать, по типу фактических параметров. Этот процесс называется *разрешением* (resolution) перегрузки. Тип возвращаемого методом значения в разрешении не участвует. Допустим, имеется четыре варианта метода, определяющего наибольшее значение:

```
// Возвращает наибольшее из двух целых:
int max( int a, int b )
// Возвращает наибольшее из трех целых:
int max( int a, int b, int c )
// Возвращает наибольшее из первого параметра и длины второго:
int max ( int a, string b )
// Возвращает наибольшее из второго параметра и длины первого:
int max ( string b, int a )
Console.WriteLine( max(  1, 2 )  ):
Console.Writel_ine( max( 1, 2, 3 ) ):
Console.Writel_ine( max( 1, "2"  ) );
Console.WriteLineC max(  "1", 2 )  ):
```

При вызове метода `max` компилятор выбирает вариант метода, соответствующий типу передаваемых в метод аргументов (в приведенном примере будут последовательно вызваны все четыре варианта метода).

Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если соответствие на одном и том же этапе может быть получено более чем одним способом, выбирается «лучший» из вариантов, то есть вариант, содержащий меньшие количество и длину преобразований. Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

Например, методы, заголовки которых приведены ниже, имеют различные сигнатуры и считаются перегруженными:

```
int max( int a.  int b )
int max( int a,  ref int b )
```

Перегрузка методов является проявлением *полиморфизма,* одного из основных свойств ООП.

3.1. Методы с переменным количеством аргументов

Язык C# предоставляет возможность создать метод, в который можно передавать разное количество аргументов. Для этого используется ключевое слово `params`. Параметр, помеченный этим ключевым словом, может быть только один и размещается в списке параметров последним и обозначает массив заданного типа неопределенной длины. Соответствующие ему аргументы должны иметь типы, для которых возможно неявное преобразование к базовому типу массива.

Например:

```
public int Calculate( int a, params int[] d ) ...
```

В этот метод можно передать три и более параметров. Внутри метода к параметрам, начиная с третьего, обращаются как к обычным элементам массива. Количество элементов массива получают с помощью его свойства `Length`.

```
int[] a = { 10, 20,  30 };
Console.WriteLine(Calculate (5, a ) );
int[] b = { -11, -4, 12, 14, 32, -1, 28 }
Console.WriteLine(Calculate (3, b ) );
short z = 1, e = 12;
byte v = 100;
Console.WriteLine(Calculate ( z, e, v ) );
Console.WriteLine(Calculate () );      //ошибка
```

Когда компилятор пытается разрешить вызов метода, он выполняет поиск метода, список аргументов которого соответствует вызываемому методу. Если не удается найти перегрузку метода с соответствующим списком аргументов, но имеется подходящая версия с параметром *params* нужного типа, выполняется вызов этого метода с добавлением в массив дополнительных аргументов.

3.2. Перегрузка операций

C# позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Определение собственных операций класса называют *перегрузкой операций.*

При перегрузке операции ни одно из его исходных значений не теряется. Перегрузку операции можно расценивать как введение новой операции для класса.

Например, можно применять операции таким же образом, как стандартные:

```
MyObject a, b, c;
c = a + b;        // используется операция сложения для класса MyObject
```

Операции класса описываются с помощью методов специального вида *(функций-операций).*

Синтаксис операции:

```
<спецификаторы> <тип_возврата> operator
op(тип_параметра операнд1[,тип_параметра операнд2])
{<тело операции>}
```

В качестве *спецификаторов* одновременно используются ключевые слова `public` и `static`. Элемент `op` – это оператор (например " + " или " / "), который перегружается. *Тело* операции определяет действия, которые выполняются при использовании операции в выражении.

При описании операций необходимо соблюдать следующие правила:

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

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

– сигнатуры всех операций класса должны различаться;

– типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

В C# существуют три вида операций класса: унарные, бинарные и операции преобразования типа.

Унарные операции

Можно определять в классе следующие *унарные операции:*

+ , **-** , **!** , **~** , **++** , **--** , `true` , `false`

Примеры заголовков унарных операций:

```
public static int operator +(
MyObject m )
public static MyObject operator --( MyObject m )
public static bool operator
true( MyObject m )
```

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция должна возвращать:

– для операций +, -, ! и ~ величину любого типа;

– для операций ++ и -- величину типа класса, для которого она определяется;

– для операций `true` и `false` величину типа `bool`.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

Бинарные операции

Можно определять в классе следующие бинарные операции:

+ , - , * , / , | , & , || , && , == , =! , > , < , >= , <=

Примеры заголовков бинарных операций:

```
public static MyObject operator + (MyObject ml, MyObject m2 )
public static bool operator == (MyObject ml, MyObject m2 )
```

Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа.

Операции == и !=, > и <, >= и <= определяются только парами и обычно возвращают логическое значение.

Пример 1. Определение операции отношения (сравнение длин векторов) для класса TD.

```
class TD // Класс трехмерных координат
{double x, y, z; // 3-х-мерные координаты
 public TD() { x = y = z = 0; }
 public TD(double i, double j, double k)
   { x = i; y = j; z = k; }
 static double r(TD op) //Вычисление длины вектора
   {return
    Math.Sqrt(op.x*op.x+op.y*op.y+op.z*op.z);}
   // Перегрузка оператора ">"
 public static bool operator >(TD op1, TD op2)
   { return r(op1) > r(op2); }
   // Перегрузка оператора "<"
 public static bool operator <(TD op1, TD op2)
   { return r(op1) < r(op2); }
 public void show()    // Отображаем координаты X, Y, Z
   { Console.WriteLine(x + ", " + y + ", " + z);}
   // Перегрузка унарного оператора "++"
 public static TD operator ++(TD op)
   {op.x++;   // Оператор "++" модифицирует аргумент
   op.y++;
   op.z++;
    return op;} }
class TDDemo
{public static void Main()
{TD a = new TD(1, 5, 3);
TD c = new TD(3, 2, 1);
Console.Write("Координаты точки a: ");
a.show();
Console.Write("Координаты точки c: ");
c.show();
```

```
Console.WriteLine();
if (a > c)
Console.WriteLine("a>c ");
else Console.WriteLine("a<c ");
a++;            // Инкрементирование a.
Console.Write("Результат a++:");
a.show();}}}
```

ЛАБОРАТОРНАЯ РАБОТА № 3. **Перегрузка методов и операций**

Задание 1. Перегрузка операций

Создайте проект, в котором опишите класс для решения задачи Вашего варианта. Разрабатываемый класс должен содержать следующие элементы: скрытые и открытые поля, конструкторы, перегруженные операции. В программе должна выполняться проверка всех разработанных элементов класса.

Варианты заданий:

1. Описать класс для работы с двумерными массивами чисел. Реализовать возможность выполнения для согласованных массивов комбинированных операций присваивания (+=, -=).
2. Описать класс для работы с двумерным массивом строк фиксированной длины. Обеспечить сравнение массивов на равенство (перегрузку операции == для поэлементного сравнения).
3. Описать класс для работы с двумерным массивом целых чисел. Реализовать возможность выполнение операции нахождения остатков от деления всех элементов массива на заданное число.
4. Описать класс, реализующий тип данных «вещественная матрица». Реализовать вычитание строки заданного номера из всех остальных строк, кроме данной строки.
5. Описать класс для работы с двумерным массивом строк. Обеспечить перегрузку операции + для построчного соединения элементов.
6. Описать класс для работы с двумерным массивом. Реализовать перегруженные операции отношений (>,<), выполняющие сравнение сумм элементов главной диагонали.
7. Описать класс для работы с двумерным массивом чисел. Реализовать возможность выполнения операции умножения согласованных массивов.
8. Описать класс для работы с двумерным массивом целых чисел. Реализовать возможность уменьшения количества столбцов массива на заданное число (перегрузка операции --) с удалением их с начала массива.

9. Описать класс для работы с двумерным массивом вещественных чисел. Обеспечить добавление к первому столбцу столбца заданного номера (перегрузка операции +).
10. Описать класс, реализующий тип данных «вещественная матрица». Реализовать операцию (++) для выполнения циклического сдвига столбцов матрицы.
11. Описать класс, реализующий тип данных «вещественная матрица». Реализовать изменение значений ненулевых элементов матрицы на противоположные, нули заменить 1.
12. Описать класс для работы с двумерным массивом целых чисел. Реализовать возможность нахождения числа, полученного перемножением положительных элементов массива, меньших 10.

Задание 2. Переопределение методов класса Object

Создать класс с закрытыми полями a и b, строковой переменной, означающей операцию и свойством C. Свойство – значение выражения над полями a и b (выражение и типы полей – см. в таблице). Поля инициализировать при создании объекта. Выполнить переопределение метода `Equals`(с одним и двумя параметрами) для сравнения объектов и метода `ToString()` для вывода состояния объекта.

Варианты заданий:

№	Тип полей a, b	Выражения	Равенство объектов
1.	float	/,-,+=	По a и свойству C для /, -
2.	double	/=,+,*	По b и свойству C для /=, *
3.	int	-,++,/	По свойству C для всех операций
4.	short	*=,+,/	По a, b и свойству C для *=
5.	long	++,*=,-	По b и свойству C для ++, -
6.	ushort	%=,+,--	По любой операции свойства и a
7.	float	/=,-,*	По a и свойству C для --
8.	double	*=,%,/	По свойству C для *= и ++
9.	decimal	*=,/,-	По свойству C для *= , – и b
10.	int	+=,--,-	По a, b и свойству C для --
11.	long	-=,++,-	По a, свойству C для -= и ++
12.	float	/=,*,%	По свойству C для всех операций

4. ИЕРАРХИИ КЛАССОВ. НАСЛЕДОВАНИЕ

Цель: данный раздел посвящен изучению приемов создания иерархии классов, выделению общих признаков объектов в базовый класс, организации доступа к элементам базового и производных классов.

4.1. Наследование

Класс в C# может иметь произвольное количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса после двоеточия:

```
[атрибуты][спецификаторы] class <имя_класса>
[:предки] <тело класса>
```

Если имя предка не указано, предком считается базовый класс всей иерархии `System.Object`.

Элементы базового класса, определенные как **`private`**, в производном классе недоступны. Поля, определенные со спецификатором **`protected`**, будут доступны методам всех классов, производных от базового. Этот модификатор остается со своим членом независимо от реализуемого количества уровней наследования.

Пример 1. Рассмотрим в качестве базового класс, в котором описывается печатная продукция (журналы, газеты и др.)

```
class Press
  {  //Закрытые поля
  string name;  //Название
  int copies;   //Тираж
  double price;//Цена
  //Конструктор с параметрами
  public Press(string name, int copies, double price)
  {  this.name = name;
     this.copies = copies;
     this.price = price;}
  //Свойства
  public string Name {get {return name;}}
  public int Copies {get {return copies;}
                     set {copies = value;}}
  public double Price {get {return price;}}
  //Метод вычисления стоимости тиража
  public double Cost(){return copies*price;}
  //Метод вывода состояния объекта
  public void Output(){
    Console.WriteLine("Название: {0} \t Тираж: {1}
    \t Цена: {2}",name,copies,price);}}
```

4.2. Вызов конструкторов базового класса

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными далее правилами:

– Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров.

– Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким образом, каждый конструктор инициализирует свою часть объекта.

– Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации. Вызов выполняется с помощью ключевого слова **base.**

Формат расширенного объявления таков:

```
имя_производного_класса(список_параметров):base
(список_аргументов)
{// тело конструктора}
```

Здесь с помощью элемента `список_аргументов` задаются аргументы, необходимые конструктору в базовом классе. При отсутствии ключевого слова `base` автоматически вызывается конструктор базового класса, действующий по умолчанию.

Пример 2. Дополним базовый класс из примера 1 производным `Magazine`, в котором есть дополнительное поле – качество бумаги.

```
class Magazine: Press {
  string quality;   //Качество бумаги
  public string Quality //Свойство
     { get { return quality; } }
  public Magazine(string name, int copies, double
  price, string quality)//Конструктор
    : base(name,copies,price)   //Вызов базового конструктора
     { this.quality = quality;} }
```

Ключевое слово `base` всегда отсылает к базовому классу, стоящему в иерархии классов непосредственно над вызывающим классом. В примере 2 это класс Press и его конструктор.

4.3. Наследование и сокрытие имен

Новым членам (полям, методам и свойствам) производного класса можно давать имена, совпадающие с именами членов базового класса. В этом случае перед членом производного класса необходимо поставить ключевое слово **new**. При этом, хотя соответствующие члены базового класса наследуются, они становятся скрытыми в производном классе.

Для обращения к скрытому члену применяется ссылка **base**, которая указывает на базовый класс производного класса. Формат ее записи такой:

base.<член базового класса>

Здесь в качестве элемента член базового класса можно указывать либо метод, либо переменную экземпляра.

Объекту базового класса можно присваивать объект производного класса, но вызываются для него только методы и свойства, определенные в базовом классе. Иными словами, возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает. Это и понятно: ведь компилятор еще до выполнения программы определяет, какой метод вызывать, и вставляет в код фрагмент, передающий управление на этот метод. Этот процесс называется ***ранним связыванием***.

Пример 3. Изменим метод Cost базового класса Press из предыдущего примера в производном классе Magazine следующим образом: в зависимости от его значения (высокое, среднее, низкое) стоимость тиража увеличивается на 10%, не изменяется или уменьшается на 10% соответственно. В метод Output добавим вывод нового поля.

```
class Magazine: Press {
   ...
public new double Cost(){
  if (quality == "HIGH")
  return base.Cost()*1.1; //Обращение к методу базового класса
     else
      if (quality == "LOW")
        return base.Cost()*0.9;
          else
            return base.Cost(); }
public new void Output(){
  base.Output(); //Вызов метода класса Press
  Console.WriteLine("Качество бумаги:"+ quality);}
   ...}
```

Использование созданной в примерах 1-3 иерархии классов:

```
class Program
```

```
{ static void Main()
{   //Объект базового класса
Press edu = new Press("Информатика", 250, 1.5);
edu.Output();   //обращение к методу базового класса
Console.WriteLine("Стоимость тиража:"+edu.Cost());
   //Объект производного класса
Magazine IiO = new Magazine("Информатика и
образование", 1500, 3.9, "LOW");
IiO.Output();   //обращение к методу производного класса
Console.WriteLine("Стоимость тиража:"+IiO.Cost());
   //Ссылка типа базового класса на объект производного класса
Press informatica = new Magazine("Мир ПК",1000, 2.1,
"HIGH");
informatica.Output();   //обращение к методу базового класса
Console.WriteLine("Стоимость
тиража:"+informatica.Cost());} }
```

Поскольку в производном классе определяется собственный метод с именем `Cost` , он скрывает одноименный метод, определенный в базовом классе. Следовательно, при вызове этого метода для объекта типа `Magazine`, вычисления выполняются так, как предусмотрено в этом классе, а не в классе `Press`. Ссылка `base` позволяет получить доступ к методу в базовом классе. Объект `informatica` типа `Press` создается с помощью конструктора класса `Magazine`. Но так как разрешение, какой метод вызвать, в данном случае осуществляется в момент компиляции (раннее связывание), то обращение происходит к методу базового класса.

Контрольные вопросы:

1. В чем состоит принцип наследования?
2. Какие члены класса наследуются?
3. Что представляет собой защищенный доступ?
4. Как происходит вызов конструкторов базового класса?
5. Что такое сокрытие имен при наследовании?
6. Как получить доступ к сокрытому члену базового класса?
7. Когда проявляется механизм раннего связывания?

ЛАБОРАТОРНАЯ РАБОТА № 4. **Наследование**

Задание 1. Создайте проект, в котором опишите иерархию классов для решения задачи Вашего варианта. Дополните базовый класс, который был создан в лабораторной работе № 1, классами-потомками. Производные классы должны содержать необходимые дополнения и изменения, указанные в варианте задания.

Составить тестирующую программу с выдачей результатов. В программе должна выполняться проверка разработанных элементов всей иерархии классов, вывод состояния различных объектов.

Варианты заданий:

№	Потомки	Дополнения и изменения в потомках
1.	Менеджер (поле объем продаж в тоннах)	Доход менеджера изменить в зависимости от объема продаж (если > H, то увеличить на 1% от H)
	Инженер (поле количество разработанных проектов – n)	Доход инженера увеличить на 4.8*n.
2.	Квартира в центре (поля номер этажа, этажность дома)	Увеличить стоимость с учетом надбавки за расположение в центре на 1%, и уменьшить на 1000$ для квартир на 1 и последнем этажах.
	Квартира в пригороде (поля расстояние от центра r)	Изменить стоимость в зависимости от расстояния: если < 10км увеличить на 3%, в противном случае уменьшить на 0,01*r.
3.	Фермерское (поле: количество внесенных удобрений m)	Найти урожай с учетом увеличения на 0,001*m на единицу площади.
	Приусадебное (поле: сроки посева – ранний, средний, поздний)	Изменить урожай с учетом сроков (+10% для раннего, –5% для позднего) и площади посевов (<0,01 увеличить на 50% от S)
4.	Письменный (поле – используемый материал, стоимость отделки)	Увеличить стоимость на стоимость отделки.
	Обеденный (поле – форма)	Изменить стоимость в зависимости от формы: увеличить для прямоугольной формы на 10%, овальной – на 20% при S<0,5 м^2 и S>2 м^2.
5.	Экспресс (поле – средняя скорость v, марка автобуса)	Изменить выручку с учетом увеличения на 0.05*v к цене билета.
	Пригородный (поле – расстояние r)	Изменить выручку с учетом уменьшения на 0.01*r к цене билета.
6.	Самолет (высота – h км, скорость – v км/ч)	Увеличить стоимость проезда на 100*h*v
	Корабль (количество палуб k, номер палубы n)	Увеличить стоимость проезда на палубах №3-4 на k^2%
7.	Инвалид (поле – номер группы)	Увеличить пособие для инвалидов 1-й группы на 30%, 2-й – на 20%.
	Многодетные семьи (поле – количество детей)	Увеличить пособие от 3 до 5 детей на 10%, > 5 – на 20%.
8.	Сотовый (поля – модель, год производства)	Изменить стоимость для телефонов, выпущенных: менее 1 года назад на +20%, более 3 лет – на –60%.

	Стационарный (поле – мобильность: переносной, непереносной)	Увеличить стоимость переносного телефона на 5.7, уменьшить стоимость непереносного телефона на 3.2.
9.	Грузовой (поле – грузоподъемность p в т)	Увеличить объем горючего на величину $M = \sqrt{p} * R$
	Легковой (поле – объем двигателя V в л)	Увеличить объем горючего для объема > 3 на величину M= 0,005*V*R
10.	Пальто (поле размер V)	Увеличить расход ткани: для пальто на (V/6.5+0.5)/10
	Костюм (поле рост H)	для костюма на (2*H+0.3)/8
11.	Офис (поле – количество этажей N)	Изменить высоту фундамента: для офиса с N>10 на+ 0,005*N
	Завод (поле -– вес G)	для завода на +0,000002*G
12.	Лекционная (поле количество ярусов K)	Увеличить количество мест в лекционной аудитории на 2*K
	Компьютерная (поле количество компьютеров P)	Заменить количество мест в компьютерной аудитории на P-1

5. ВИРТУАЛЬНЫЕ МЕТОДЫ И ПОЛИМОРФИЗМ

Цель: формирование представления о реализации принципа полиморфизма с помощью механизма позднего связывания.

5.1. Виртуальные методы

В том случае, если ссылке базового класса присваивается объект производного класса, то вызываются для него только методы и свойства, определенные в базовом классе. Чтобы вызываемые методы соответствовали типу объекта, необходимо отложить процесс связывания до этапа выполнения программы, а точнее – до момента вызова метода, когда уже точно известно, на объект какого типа указывает ссылка. Такой механизм называется ***поздним связыванием*** и реализуется с помощью так называемых виртуальных методов.

В C# виртуальные методы описываются с использованием ключевого слова **`virtual`**. Оно записывается в заголовке метода базового класса, например:

```
virtual public void Output()…
```

Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово **`override`**, например:

```
override public void Output()…
```

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

При описании 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.

Все сказанное о виртуальных методах относится также к свойствам и индексаторам.

Пример 1. Опишем методы `Cost` и `Output` базового класса `Press` из примеров 1-3 раздела 4 как виртуальные. В производном классе `Magazine` переопределим их. Протестируем процесс вызова метода с помощью объектов как базового, так и производного классов.

```
class Press {
   …
//Виртуальный метод вычисления стоимости тиража
  virtual public double Cost()
    {return copies*price;}
//Виртуальный метод вывода состояния объекта
  virtual public void Output(){
    Console.WriteLine("Название: {0} \t Тираж: {1}
    \t Цена: {2}",name,copies,price);}
   … }
class Magazine: Press {
   …
//Переопределенные методы
override public double Cost(){
  if (quality == "HIGH")
  return base.Cost()*1.1;
     else
       if (quality == "LOW")
         return base.Cost()*0.9;
           else
             return base.Cost(); }
override public void Output(){
  base.Output();
  Console.WriteLine("Качество бумаги:"+ quality);}
   …}
  class Program
  {static void Main()
   {const int n = 3;
```

```
Base[] st = new Base[n]; //создаем контейнер ссылок
                         //на объекты базового класса
//создаем объекты базового класса
st[0] = new Press("Информатика", 250, 1.5);
st[1] = new Press("Компьютер", 100, 3.5);
//создаем объект класса-наследника
st[2] = new Magazine("Информатика и образова-
ние", 1500, 3.9, "LOW");
//выводим поля объектов массива
foreach (Press elem in st )
{elem.Output(); Console.WriteLine("Стоимость
тиража:"+ st[i].Cost());}
//обнуляем поля объектов
for ( int i = 0; i < n; ++i ) st[i].Copies = 0;
//выводим поля объектов массива
foreach (Press elem in st ) elem.Output();}}
```

В циклах вызываются методы и свойства, соответствующие типу объекта, помещенного в массив. Это иллюстрирует механизм позднего связывания.

5.2. Абстрактные классы

Абстрактный класс служит только для порождения потомков. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по-своему.

Абстрактный класс задает интерфейс для всей иерархии, при этом метод класса может не содержать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором **abstract**. Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть описан как абстрактный. Абстрактный класс может содержать и полностью определенные методы.

Пример 2. Абстрактный класс и его потомки.

```
using System;
namespace ConsoleApplication2
{abstract class Spirit
  {public abstract void Print();}
class Base : Spirit
  {override public void Print()
  {Console.WriteLine( "Base ");}}
class Derivative : Base
  {override public void Print()
  {Console.WriteLine("Derivative");}}}
```

Если класс, производный от абстрактного, не переопределяет все абстрактные методы, он также должен описываться как абстрактный.

Можно создать метод, параметром которого является абстрактный класс. На место этого параметра при выполнении программы может передаваться объект любого производного класса. Это позволяет создавать полиморфные методы, работающие с объектом любого типа в пределах одной иерархии.

Контрольные вопросы:

1. Что означает принцип полиморфизма?
2. Для чего используется позднее связывание?
3. В каких случаях используются виртуальные методы?
4. Какие условия необходимо соблюдать при переопределении виртуального метода?
5. Что представляют собой абстрактные классы? Для чего они предназначены?
6. Могут ли в абстрактном классе быть неабстрактные методы?

ЛАБОРАТОРНАЯ РАБОТА № 5. **Полиморфизм. Виртуальные методы**

Задание 1. Модифицируйте проект из лабораторной работы № 4 следующим образом.

Создайте библиотеку классов и добавьте в нее базовый класс. Опишите методы в базовом классе как виртуальные.

Составьте новое приложение с двумя потомками базового класса из библиотеки. Тестирующая программа должна содержать полиморфный контейнер – массив ссылок базового класса на объекты базового и производных классов (количество объектов >=5). В программе должна выполняться проверка всех разработанных элементов.

Задание 2. Модифицируйте проект задания 1 следующим образом.

Базовый класс опишите как абстрактный и добавьте его в созданную библиотеку. Указанный в таблице (лаб. раб. №1) метод базового класса не должен содержать никаких конкретных действий.

В тестирующей программе используйте полиморфный контейнер – массив ссылок абстрактного класса на объекты производных классов (количество объектов >=5 и размещаются в массиве в произвольном порядке) и реализуйте указанное в варианте задания дополнение. В программе должна выполняться проверка всех разработанных элементов.

Варианты заданий:

1. Организовать вычисление суммарной величины дохода сотрудников и максимальное для менеджеров.
2. Рассчитать среднюю стоимость квартир в центре и найти наибольшее расстояние от города до квартир в пригороде.
3. Найти суммарный урожай фермеров и суммарный урожай с приусадебных участков.
4. Найти суммарную стоимость всех столов и наименьшую площадь обеденных столов.
5. Рассчитать общую выручку и найти наибольшее расстояние для пригородных автобусов.
6. Организовать вычисление средней стоимости проезда на самолетах и средней стоимости проезда кораблем.
7. Определить, на какой вид пособий потребуется больше средств.
8. Найти максимальные стоимости сотовых и стационарных телефонов.
9. Организовать вычисление суммарного расхода горючего и средний расход для легковых автомобилей.
10. Организовать вычисление суммарного расхода ткани для каждого вида одежды.
11. Найти максимальную высоту фундаментов для офиса и минимальную для заводов.
12. Найти общее количество мест в лекционных аудиториях и общее число компьютеров.

6. ИНТЕРФЕЙСЫ И СТРУКТУРНЫЕ ТИПЫ

Цель: сформировать понятие о реализации принципа полиморфизма с помощью интерфейсов, умения использовать пользовательские интерфейсы для задания поведения различных классов.

6.1. Понятие интерфейса

Интерфейс является специальным видом классов.

Синтаксис интерфейса аналогичен синтаксису класса:

```
[атрибуты][спецификаторы] interface
<имя_интерфейса> [:предки] <тело_интерфейса> [;]
```

Для интерфейса могут быть указаны спецификаторы, **new**, **public**, **protected**, **internal** и **private**. По умолчанию интерфейс доступен только из сборки, в которой он описан (**internal**). Интерфейс может наследовать свойства нескольких интерфейсов. Тело интерфейса составляют абстрактные методы, шаблоны свойств и индексаторов, а также события.

Интерфейс не может содержать константы, поля, операции, конструкторы, деструкторы, типы и любые статические элементы. В интерфейсе методы неявно являются открытыми (**public**-методами), при этом не разрешается явным образом указывать спецификатор доступа.

Чтобы реализовать интерфейс, нужно указать его имя после имени класса. В списке предков класса сначала указывается его базовый класс, если он есть, а затем через запятую – интерфейсы, которые реализует этот класс.

Методы, которые реализуют интерфейс, должны быть объявлены открытыми. Сигнатура типа в реализации метода должна в точности совпадать с сигнатурой типа, заданной в определении интерфейса. В классах, которые реализуют интерфейсы, можно определять дополнительные члены.

Пример 1.

```
interface IMy_A
{void meth(int x);
// В классе MyClass реализован интерфейс  IMy_A
class MyClass : IMy_A
{
 int h;
 public MyClass()
 { h = 5; }
 public void meth(int x)
 {h= x + x;
  Console.WriteLine("MyClass.meth..."+h);}
}
class Class1: IMy_A
 { public static void meth(int a)
 {Console.WriteLine("Class1.meth..."+a*a);}
 public static void Main()
 {
  MyClass ob = new MyClass();
  Console.Write ("Вызов метода ob.meth(): ");
  ob.meth(3);
 meth(3); } //реализация интерфейса IMy_A в классе Class1
}
```

Можно объявить ссылочную переменную интерфейсного типа, которая может ссылаться на любой объект, реализующий ее интерфейс. При вызове метода для объекта посредством интерфейсной ссылки будет выполнена та версия указанного метода, которая реализована этим объектом.

Элементы с одинаковыми именами или сигнатурой могут встречаться более чем в одном интерфейсе. В этом случае при множественном наследовании может возникнуть конфликт из-за неоднозначности ситуации, так как компилятор не может определить из контекста обращения к элементу, элемент какого именно из реализуемых интерфейсов требуется вызвать. Избежать неоднозначности можно с помощью бинарных операций **is** или **as**, которые позволяют убедиться, что объект поддерживает данный интерфейс или выполнить приведение к данному интерфейсу.

Для этой же цели можно использовать так называемую ***явную реализацию***, когда имя интерфейса явно указывается перед реализуемым элементом через точку. Такая реализация интерфейса является закрытой. Спецификаторы доступа при этом не указываются. К таким элементам можно обращаться в программе только через объект типа интерфейса.

Кроме того, явная реализация позволяет избежать конфликтов при множественном наследовании, если элементы с одинаковыми именами или сигнатурой встречаются более чем в одном интерфейсе.
Пример 2. Реализовано два интерфейса, причем оба объявляют метод с одним именем.

```
//Использование явной реализации, чтобы избежать неоднозначности.
interface IMy_A
{void meth(int x);}
interface IMy_B
{void meth(int x);}
// В классе MyClass реализованы оба интерфейса
class MyClass : IMy_A, IMy_B
{int h;
public MyClass()
{ h = 5; }
// Явным образом реализуем метод meth() интерфейса IMyIF_A
void IMy_A.meth(int x)
{h = x + x;
Console.WriteLine("IMy_A.meth..." + h);}
public void meth(int x)
{h = x * x;
Console.WriteLine("meth..." + h);}
}
class Demo
{public static void Main()
{MyClass ob = new MyClass();
//Реализация интерфейса IMy_B
Console.Write("Вызов метода ob.meth(): ");
```

```
ob.meth(3);
Console.WriteLine("Реализация интерфейса IMy_A");
(ob as IMy_A).meth(4);
Console.WriteLine("Вызов метода meth
инт_ссылкой");
IMy_A a_ob= ob;   //интерфейсной ссылке присваиваем объект класса
Console.WriteLine("Вызов метода IMy_A.meth()");
a_ob.meth(3); //реализация интерфейса IMy_A
}
```

Метод `meth()` имеет одинаковую сигнатуру в интерфейсах `IMy_A` и `IMy_B`. Поскольку единственный способ вызова явно заданного метода состоит в использовании интерфейсной ссылки, метод `meth()`, объявленный в интерфейсе `IMy_A`, создает ссылку на интерфейс `IMy_A`, а метод ob.`meth()`, вызывает метод, объявленный в интерфейсе `IMy_B`.

6.2. Наследование интерфейсов

Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в последнем случае он наследует все элементы всех своих базовых интерфейсов, начиная с самого верхнего уровня. Базовые интерфейсы должны быть доступны так же, как их потомки.

В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово **new**.

Любой класс, который реализует интерфейс, должен реализовать все методы, определенные этим интерфейсом, включая методы, которые унаследованы от других интерфейсов.

Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора **new**, но обращаться к ним можно будет только через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия.

Однако если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату

Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается.

Если класс наследует от класса и интерфейса, которые содержат методы с одинаковыми сигнатурами, унаследованный метод класса воспринимается как реализация интерфейса.

6.3. Стандартные интерфейсы среды .NET Framework

В библиотеке классов **.NET** определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

Во многих классах необходимо реализовать интерфейс ***IComparable*** или ***IComparer***, поскольку они позволяют сравнить два объекта.

Интерфейс ***IComparable*** состоит только из одного метода:

```
int CompareTo(object v)
```

Этот метод сравнивает вызывающий объект со значением параметра `v`. Метод возвращает положительное число, если вызывающий объект больше объекта `v`, нуль, если два сравниваемых объекта равны, и отрицательное число, если вызывающий объект меньше объекта `v`. Метод ***CompareTo*** может сгенерировать исключение типа ***ArgumentException***, если тип объекта v несовместим с вызывающим объектом.

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

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

Метод ***Compare***() возвращает положительное число, если значение `vl` больше значения `v2`, отрицательное, если `vl` меньше `v2`, и нуль, если сравниваемые значения равны. Этот интерфейс можно использовать для задания способа сортировки элементов коллекции.

Контрольные вопросы:

1. Как описывается интерфейс? Его назначение.
2. Какие члены может содержать интерфейс?
3. Какие спецификаторы допустимы у методов, реализующих интерфейс?
4. В каких случаях используется явная реализация интерфейса?
5. Как осуществляется наследование интерфейсов?
6. Можно ли явно реализованные методы объявлять виртуальными?
7. Можно ли повторно реализовать интерфейс, указав его имя в списке предков класса наряду с классом-предком?
8. Какие стандартные интерфейсы используются для работы с коллекциями?

ЛАБОРАТОРНАЯ РАБОТА № 6. **Интерфейсы**

Задание 1.

1.1. Реализовать неявно интерфейсы в классах 1 и 2, содержащих поле указанного типа.

Переменная `w` *обозначает* `параметр` *метода или* `поле`*. Результат метода со спецификатором* `void` *присвоить полю класса.*

В тестирующей программе должны выполняться:

- неявная неоднозначная реализация методов интерфейсов;
- вызов методов с явным приведением к типу интерфейса;
- вызов методов для объекта посредством интерфейсной ссылки.

1.2. Модифицировать проект, используя явную реализацию интерфейса для методов с одинаковой сигнатурой в классе 1.

Варианты заданий:

№	интерфейсы	поле	классы	F0 F1 возвращают:	
				неявная реализация	явная реализация
1.	`interface Iy {void F0(double параметр); void F1();} interface Iz {void F0(double параметр); float F1(int параметр);}`	`double`	1	$\sqrt{w}$	w^2+5
			2	e^w	-
2.	`interface IA {void F0(out double параметр); int F1(double параметр);} interface IB {void F0( out double параметр); void F1();}`	`double`	1	w^2	`15/w`
			2	w^3	-
3.	`interface Ix {char F0(); void F1(ref char параметр);} interface Iy {void F0(char параметр); void F1(ref char параметр);}`	`char`	1	Символ, преобразованный в нижний регистр	цифру ‘5’, если символ буква
			2	`*`, если символ – буква	-

4.	interface Ix {void F0(); void F1(string параметр);} interface Iy {void F0(string параметр); void F1(string параметр);}	string	1	Строку, удалив два первых символа	Строку, заменив первый символ символом '_'
			2	Строку, удалив два последних символа	_
5.	interface Ix {bool F0(char параметр); void F1(int параметр);} interface Iy {bool F0(char параметр); void F1(char параметр);}	char	1	Определяет, является ли символ (код символа) цифрой	Определяет, содержится ли значение кода символа в диапазоне кодов ASCII
			2	Определяет, является ли символ знаком препинания	_
6.	interface Ix {string F0(int параметр); void F1(int параметр);} interface Iy {void F0(int параметр,out string параметр); void F1(int параметр);}	string	1	Удвоенную строку с удаленным k-м символом	Строку, заменив k-ый символ знаком +
			2	подстроку, начиная с k-ей позиции	_
7.	interface Iy {void F0(double параметр); void F1(double параметр);} interface Iz {void F0(double параметр); double F1();}	double	1	sin(w)	w+2
			2	2/w*ln(w)	_
8.	interface Ix {void F0(int k); void F1(int k, out int параметр);} interface Iy {void F0(int k); void F1(int k);}	int	1	$k*w^2$	2w-k
			2	|w|-k	-

9.	interface IA {double F0(double параметр); int F1(double параметр);} interface IB {double F0(double параметр); void F1();}	double	1	$1/e^w$	w/10
			2	w-1010*w	–
10.	interface Ix {int F0(); void F1();} interface Iy {int F0(); void F1(char параметр);}	char	1	Если символ является цифрой найти его числовое значение, иначе его код	вернуть код символа
			2	Если символ является буквой x, вывести число 0	–
11.	interface Iy {void F0(double параметр); float F1(int параметр);} interface Iz {void F0(out double параметр); float F1(int параметр);}	float	1	7w/ln(2)	7w*2.6
			2	|w-10|	–
12.	interface Is {string F0(); void F1();} interface Ir {void F0(string параметр); void F1();}	string	1	Заменить все буквы в строке на прописные символы	изменить регистр всех букв в строке
			2	Заменить на пробелы знаки препинания	–

Задание 2.

Выполнить задания, используя для хранения экземпляров разработанных классов стандартную коллекцию ***ArrayList***. Во всех классах реализовать интерфейсы ***IComparable и IComparer***. Перегрузить операции отношения для реализации сравнения объектов по указанному полю. Результат вывести на экран. Организовать вывод коллекции.

Варианты заданий:

1. Составить багажную ведомость камеры хранения, включив следующие данные: ФИО пассажира, количество вещей, общий вес вещей. Вывести в новый список информацию о тех пассажирах, средний вес багажа которых превышает заданный, отсортировав их по количеству вещей, сданных в камеру хранения.
2. Составить список студентов, включающий ФИО, курс, группу, дату и результат забега. Вывести в новый список информацию о студентах, показавших три лучших результата в забеге. Если окажется, что некоторые студенты получили такие же высокие результаты, то добавить их к списку победителей, отсортировать по результатам.
3. Составить инвентарную ведомость склада, включив следующие данные: вид продукции, стоимость, сорт, количество. Вывести в новый список информацию о той продукции, количество которой меньше заданной величины, отсортировав ее по количеству продукции на складе.
4. Составить список студентов группы, включив следующие данные: ФИО, номер группы, результаты сдачи трех экзаменов. Вывести в новый список информацию о студентах, успешно сдавших сессию, отсортировав по номеру группы.
5. Составить список вкладчиков банка, включив следующие данные: ФИО, № счета, сумма, дату открытия счета. Вывести в новый список информацию о тех вкладчиках, которые открыли вклад в текущем году, отсортировав их по сумме вклада.
6. Составить автомобильную ведомость, включив следующие данные: марка автомобиля, фамилия его владельца, год приобретения, пробег. Вывести в новый список информацию об автомобилях, выпущенных ранее определенного года, отсортировав их по пробегу.
7. Составить инвентарную ведомость игрушек, включив следующие данные: название игрушки, ее стоимость (в руб.), возрастные границы детей, для которых предназначена игрушка. Вывести в новый список информацию о тех игрушках, которые предназначены для детей от N до M лет, отсортировав их по стоимости.
8. Составить список студентов группы, включив следующие данные: ФИО, дату рождения, какую школу окончил. Вывести в новый список информацию о студентах, окончивших заданную школу, отсортировав их по году рождения.
9. В библиотеке имеется список книг. Каждая запись этого списка содержит фамилии авторов, название книги, год издания, цена экземпляра. Определить, имеются ли в данном списке книги, в названии которых встречается некоторое ключевое слово (например, "программирование"). Если имеются, то вывести фамилии авторов, название и год издания всех таких книг.

10. Составить список студентов, включающий фамилию, факультет, курс, группу, 5 оценок. Вывести в новый список информацию о тех студентах, которые имеют хотя бы одну двойку, отсортировав их по курсу.
11. Составить список больных отделения, включив следующие данные: ФИО, болезнь, дата поступления, рабочий стаж. Вывести в новый список информацию о больных, находящихся на лечении больше недели, отсортировав их по ФИО.
12. В пресс-центре выставки программных средств хранятся данные о каждом экспонате: название, автор, количество заявок на него. Вывести в новый список экспонаты, получившие больше трех заявок, отсортировав по числу заявок.

7. ДЕЛЕГАТЫ. СОБЫТИЯ

Цель: овладение основными приемами событийного программирования и их программной реализацией.

7.1. Делегаты

Делегат – это вид класса, предназначенный для хранения ссылок на методы. Делегат представляет собой тип данных. Его базовым классом является класс System.Delegate. Наследовать от делегата нельзя. Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса.

Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

```
[атрибуты] [спецификаторы] delegate <тип>
<имя_делегата>([параметры])
```

Спецификаторы делегата имеют тот же смысл, что и для класса, причем допускаются только спецификаторы new, public, protected, internal и private.

Тип описывает возвращаемое значение методов, вызываемых с помощью делегата, а необязательными *параметрами* делегата являются параметры этих методов.

Делегат может хранить ссылки на несколько методов и вызывать их по очереди.

Пример описания делегата:

```
public delegate void D(int i);
```

Делегат может вызывать либо метод экземпляра класса, связанный с объектом, или статический метод, связанный с классом.

При реализации делегата создается ссылка типа делегата и ей присваивается ссылка на метод, который передается делегату в качестве параметра, причем используется только имя метода (параметры

не указываются). Затем этот метод вызывается посредством экземпляра делегата.

Таким образом, вызов экземпляра делегата трансформируется в обращение к методу, на который он ссылается при вызове. И, следовательно, решение о вызываемом методе принимается во время выполнения программы, а не в период компиляции.

Многоадресатная передача

Многоадресатная передача (multicasting) – это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата. Для этого достаточно создать экземпляр делегата, а затем для добавления методов в эту цепочку использовать оператор "+=". Для удаления метода из цепочки используется оператор " - = ".

Делегат с многоадресатной передачей имеет одно ограничение: он должен возвращать тип void.

Пример 1.

```
// Демонстрация использования многоадресатной передачи
using System;
// Объявляем делегат
delegate void strMod(ref string str);
class SOps
{// Метод заменяет пробелы дефисами
    static void rs(ref string a)
    {   a = a.Replace(' ', '-');
        Console.WriteLine("Замена пробелов дефисами." + a);
    }
// Метод удаляет символ т.
    static void ms(ref string a)
    {   string temp = "";
        temp = a.Replace("т", "");
        a = temp;
        Console.WriteLine("Удаление  т ." + a);
    }
    public static void Main()
    {// Создаем экземпляры делегатов
        strMod strOp;
        strMod rSp = new strMod(rs);
        strMod mSp = new strMod(ms);
        string str = "Это простой тест.";
// Организация многоадресатной передачи
        strOp = rSp;
        strOp += mSp;
// Вызов делегата с многоадресатной передачей
        strOp(ref str);
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();
```

```
        str = "Это простой тест."; // Восстановление исходной строки
// Вызов делегата с многоадресатной передачей
        strOp(ref str);
        Console.WriteLine("Результирующая строка: " + str);
        Console.WriteLine();    }}
```

7.2. События

На основе делегатов построено важное средство C#: *событие* (event). Синтаксис события:

```
[атрибуты][спецификаторы]event <тип> <имя>
```

Для событий применяются спецификаторы `new`, `public`, `protected`, `internal`, `private`, `static`, `virtual`, `sealed`, `override`, `abstract` и `extern`. Событие может быть статическим (`static`), тогда оно связано с классом в целом, или обычным – в этом случае оно связано с экземпляром класса.

Тип события – это тип делегата, на котором основано событие.

Пример описания делегата и соответствующего ему события:

```
  public delegate void Del(object o); //объявление делегата
class A
{public event Del Oops; //объявление события
}
```

События работают следующим образом. Объект, которому необходима информация о некотором событии, регистрирует обработчик для этого события, сигнатура которых соответствует типу делегата. Когда ожидаемое событие происходит, вызываются все зарегистрированные обработчики.

Если в качестве обработчика используется статический метод, уведомление о событии применяется к классу (и неявно ко всем объектам этого класса). Если же в качестве обработчика событий используется метод экземпляра класса, события посылаются к конкретным экземплярам этого класса.

Добавлять обработчики в список или удалять их можно с помощью операторов "+=" и " - = ".

Внутри класса, в котором описано событие, с ним можно обращаться, как с обычным полем, имеющим тип делегата: использовать операции отношения, присваивания и т. д. Значение события по умолчанию – null.

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

Пример 2.

```
// Демонстрация использования события, предназначенного
// для многоадресатной передачи.
using System;
```

```
// Объявляем делегат для события,
delegate void Del();
// Объявляем класс события,
class Et
{
  public event Del SE;
// Этот метод вызывается для генерирования события,
  public void OnSE()
  {if(SE != null) SE();}
}
class X
{public void Xh()
{Console.WriteLine("Событие, полученное объектом X.");}
}
class Y
{public void Yh()
  {Console.WriteLine("Событие, полученное объектом Y.);}}
class EventDemo
{static void h()
  {Console.WriteLine("Событие, полученное классом
  EventDemo");}
  public static void Main()
  {Et evt = new Et();
    X xOb = new X();
    Y yOb = new Y();
// Добавляем обработчики в список события,
    evt.SE += new Del(h);
    evt.SE += new Del(xOb.Xh);
    evt.SE += new Del(yOb.Yh);
// Генерируем событие,
    evt.OnSE();
    Console.WriteLine();
// Удаляем один обработчик.
    evt.SE -= new Del(xOb.Xh);
    evt.OnSE();}}
```

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 заканчивается суффиксом `EventHandler`;

– делегат получает два параметра: первый параметр задает источник события и имеет тип `object`; второй параметр задает аргументы события и имеет тип `EventArgs` или производный от него.

Если обработчикам события требуется специфическая информация о событии, то для этого создают класс, производный от стандартного класса `EventArgs`, и добавляют в него необходимую ин-

формацию. Если делегат не использует такую информацию, можно обойтись стандартным классом делегата `System.EventHandler`.

Имя обработчика события принято составлять из префикса `On` и имени события.

ЛАБОРАТОРНАЯ РАБОТА № 7. **Делегаты. События**

Задание 1.

Разработать проект использования делегата для:

- вызова разных методов одним экземпляром делегата;
- многоадресной передачи.

В методах предусмотреть вывод результата.

Варианты заданий:

№	Параметры делегата	Методы возвращают:		
		класс1		класс2
		метод_1 (статич.)	метод_2 (экз.)	
1.	double	$\sqrt{w}$	h^2+5	e^p
2.	double	x^2	15/y	n^3
3.	char	символ, преобразованный в нижний регистр	цифру ‘5’, если символ – буква	*, если символ – буква
4.	string	строку, удалив два первых символа	строку, заменив первый символ символом ‘-‘	строку, удалив два последних символа
5.	char,bool	определяет, является ли символ (код символа) цифрой	определяет, содержится ли значение кода символа в диапазоне кодов ASCII	определяет, является ли символ знаком препинания
6.	int, string	удвоенную строку с удаленным k-м символом	строку, заменив s-ый символ знаком ‘+’	подстроку, начиная с h-ой позиции
7.	double	sin(r)	n+2	2/y*ln(y)
8.	int, int	$n*u^2$	2*w-t	|x|-k
9.	double	$1/e^p$	x/10	w-1010*w
10.	char,int	если символ является цифрой найти его числовое значение, иначе его код	вернуть код символа	если символ является буквой ‘x’, вывести число 0
11.	int,float	7*m/ln(2)	7*i*2.6	|k-10|
12.	string	заменить все буквы в строке на прописные символы	изменить регистр всех букв в строке	заменить на пробелы знаки препинания

Задание 2.

Создать событие на основе делегата (задание 1). В тестирующем классе организовать цепочку вызовов.

8. ФАЙЛЫ

Цель: освоить способы работы с файловыми потоками через основные классы для работы с файлами.

8.1. Файловый ввод-вывод

Основными классами для работы с файлами и потоками в C# являются **File**, **FileStream** и **StreamReader StreamWriter**. Класс **File** предназначен для создания, открытия, удаления, изменения атрибутов файла.

Выполнять обмен с внешними устройствами можно на уровне:

- *двоичного представления данных* (BinaryReader, BinaryWriter);
- *байтов* (FileStream);
- *текста,* то есть символов (StreamWriter, StreamReader).

Рассмотрим простейшие работы с файловыми потоками.

FileStream - представляет поток, который позволяет выполнять операции чтения/записи в файл.

Использование классов файловых потоков в программе предполагает следующие операции:

1. Создание потока и связывание его с физическим файлом.
2. Обмен (ввод-вывод).
3. Закрытие файла.

Каждый класс файловых потоков содержит несколько вариантов конструкторов, с помощью которых можно создавать объекты этих классов различными способами и в различных режимах. Например, файлы можно открывать только для чтения, только для записи или для чтения и записи. Эти *режимы доступа* к файлу содержатся в перечислениях FileMode и FileAccess, определенном в пространстве имен System.I0.

Пример инициализации объекта FileStream:

```
FileStream myStream. = File.Open("C:\MyFile.rxt", FileMode.Open, FileAccess.Read);
```

В данном случае *режим открытия* установлен как FileMode.Open, что означает открыть файл, если он существует; права доступа установлены FileAccess.Read, что означает возможность только читать файл. Функция Open возвращает объект типа FileStream, посредством которого в дальнейшем происходят чтение или запись в файл.

Иногда удобнее работать с файлом с помощью класса и StreamReader. StreamReader и StreamWriter связываются с потоком при помощи конструктора инициализации.

Для StreamWriter используйте один из следующих конструкторов:

```
StreamWriter(string filename)
StreamWriter(string filename, bool appendFlag)
```

Здесь элемент *filename* означает имя открываемого файла, причем имя может включать полный путь к файлу. Во второй форме используется параметр *appendFlag* типа bool: если *appendFlag* равен значению true, выводимые данные добавляются в конец существующего файла. В противном случае заданный файл перезаписывается.

В обоих случаях, если файл не существует, он создается, а при возникновении ошибки ввода-вывода генерируется исключение типа IOException (также возможны и другие исключения).

Для StreamReader чаще всего используется следующий конструктор:

```
StreamReader(string filename);
```

Здесь элемент stream означает имя открытого потока. Этот конструктор генерирует исключение типа ArgumentNullException, если поток stream имеет null-значение, и исключение типа ArgumentException, если поток stream не открыт для ввода. После создания объект класса StreamReader автоматически преобразует байты в символы.

Можно указывать кодировку, в которой будут считываться/записываться данные при создании StreamReader/StreamWriter:

```
static void Main(string[] args)
 {FileStream file1=new File-
Stream("d:\\test.txt",FileMode.Open);
  StreamReader reader = new StreamReader(file1,
Encoding.Unicode);
  StreamWriter writer = new StreamWriter(file1,
Encoding.UTF8); }
```

Контрольные вопросы:

1. Что понимается под потоком данных?
2. На каких уровнях может осуществляться обмен данными с внешними устройствами?
3. Какой класс обеспечивает работу с файловыми потоками?
4. Перечислите основные операции при использовании классов файловых потоков?
5. Какие классы обеспечивают ввод/вывод данных в/из файлов?

9. РАЗРАБОТКА WINDOWS-ПРИЛОЖЕНИЙ

Цель: разработка графического интерфейса для Windows-приложений с помощью технологии Windows Forms. Работа с диалогами открытия и сохранения файлов.

9.1. Разработка графического интерфейса для Windows-приложений с помощью технологии WindowsForms

Процесс создания Windows-приложения состоит из двух основных этапов:

- визуальное проектирование, то есть задание внешнего облика приложения;
- определение поведения приложения путем написания процедур обработки событий.

Визуальное проектирование заключается в помещении на форму компонентов (элементов управления) и задании их свойств и свойств самой формы.

Windows-приложение состоит из главной программы, обеспечивающей инициализацию и завершение приложения, цикла обработки сообщений и набора обработчиков событий.

Принципиально важным признаком Windows-приложений является то, что все они – программы, управляемые событиями (event-driven applications).

Среда Visual Studio.NET содержит удобные средства разработки Windows-приложений, выполняющие вместо программиста рутинную работу – создание шаблонов приложения и форм, заготовок обработчиков событий, организацию цикла обработки сообщений и т. д.

Шаблон Windows-приложения

Среда создает заготовку формы и шаблон текста приложения.

Процесс создания Windows-приложения состоит из двух основных этапов:

1. Визуальное проектирование, то есть задание внешнего облика приложения.
2. Определение поведения приложения путем написания процедур обработки событий.

Визуальное проектирование заключается в помещении на форму компонентов (элементов управления) и задании их свойств и свойств самой формы с помощью окна свойств.

Таким образом, для размещения компонента на форме необходимо выполнить три действия:

1. Создать экземпляр соответствующего класса.
2. Настроить свойства экземпляра, в том числе зарегистрировать обработчик событий.

3. Поместить экземпляр в коллекцию Controls компонентов формы

В теле обработчиков событий программист может самостоятельно написать код, который будет выполняться при наступлении события.

ЛАБОРАТОРНАЯ РАБОТА № 8. **Разработка простейшего Windows-приложения**

Задание 1. Работа с формой

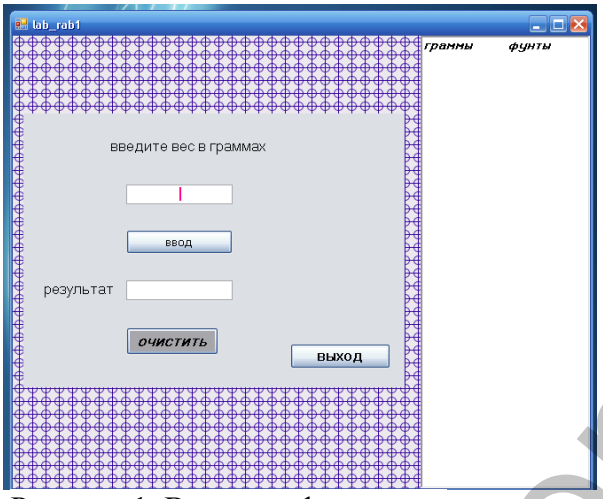

Рисунок 1. Вид окна формы.

Для создания приложения, изображенного на рис. 1, выполните следующие действия:

1. Создайте новый проект**: File / New / Project**, либо **New** в окне **Start Page.**
2. В окне **New project** в левой части выбрать *Visual C# Projects*, в правой – пункт *Windows Forms Application*
3. В поле ***Name*** введите имя проекта, в поле ***Location*** – место его сохранения на диске.
4. Просмотрите свойства формы, представленные на странице *Properties.*
5. Измените свойство *Text* с **Forml** на **Лабораторная работа 8.**
6. Обратите внимание на свойство *Name.* Это свойство определяет имя компонента, под которым компонент будет известен программе.
7. Измените свойство *BackGroundImage* **Forml** на какой-либо рисунок. Для этого в окне **Select Resource** установите переключатель *Local Resource*, нажмите *Import* и выберите файл изображения.
8. Поместите на форму компонент **Panel.** Осуществите прогон пустой программы. В рабочем приложении максимизируйте окно, а затем закройте его.
9. *Задайте* свойству компонента BackColor **Panel1** значение *Control-Light.*
10. *Разместите* **Panel1** в центре формы.
11. *Отбуксируйте* сторону компонента **Panel,** ухватившись за верхний обрамляющий черный квадратик. Обратите внимание на то, что это значение установилось в свойстве *Height* инспектора объектов. Задайте размер *Size* 430; 320.

12. Установите на панель **Panel1** метку **Labei1.** Свойству *Text* придайте значение «Введите значение веса в граммах». В свойстве *Font* раскройте диалоговое окно настройки шрифта и измените высоту шрифта – 12пт.
13. Установите на панель **Panel1** редактор **TextBox1.** Очистите свойство *Text.*
14. Установите на панель кнопку **Buttonl.** Задайте свойству *Text* значение «Ввод».
15. Поместите на панель **Panel1** метку **Label2.** Свойству *Text* придайте значение «Результат». Установите высоту шрифта 12пт.
16. *Установите* на панель **Panel1** редактор **TextBox2.** Очистите свойство *Text.* Установите начертание шрифта – курсив, размер 10пт.
17. Поместите на форму компонент **richTextBox1** (многострочный редактор) и задайте свойству *Dock* значение *Richt.* Компонент займет правую часть формы.
18. Свойству многострочного редактора *Lines* введите текст «Граммы Фунты» через 2 табуляции.
19. Установите на панель кнопку **Button2.** Задайте свойству *Text* значение «Выход». Задайте высоту шрифта 12пт. Измените свойство *UseVisualStyleBackColor* на false.
20. Выделите кнопку **Buttonl.** В инспекторе щелкните по закладке **events**. Дважды щелкните по правой части строки события *OnClick.* В ответ C# активизирует окно программы. Событие *OnClick* возникает в работающей программе при щелчке мышью по кнопке.
21. Используя линейки прокрутки, просмотрите содержимое окна программы.
22. Занесите в программу (обработчик события) следующий код.

После `public partial class Form1 : Form` введите описания переменных:

```
double x; //значение веса в граммах, вводимое в textBox1
double y; // значение веса в фунтах
int i=1; //количество символов, в строке richTextBox1
```

В методе

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

между операторными скобками { } введите следующий код:

```
x = Convert.ToDouble(textBox1.Text);
y = x / 400;
textBox2.Text= Convert.ToString(y);
MessageBox.Show("Нажата кнопка buttonl");
richTextBox1.Multiline=true;
richTextBox1.Text += (x+
"\t\t"+Convert.ToString(y) + "\n");
```

```
MessageBox.Show("Нажата клавиша " +
e.ToString());
richTextBox1.TabIndex = i;
i += (Convert.ToString(y) + (char)13).Length;
```

Пояснения. В первом операторе присваивания содержимое окна редактора преобразуется в вещественное число. Для отражения результата расчета в окне `textBox2` *используется свойство* `Text`*. этого компонента. Свойство* `Text` *класса* `richTextBox` *добавляет новую строку к имеющемуся в* `Text` *набору строк. Добавленная строка отображается на экране. Свойство* `TabIndex` *устанавливает номер позиции в тексте* `richTextBox1.Text`*. Последний оператор изменяет эту позицию на длину добавленной строки.*

23. Создайте обработчик события для кнопки **Button2,** введя текст `Close();`
24. Сохраните программу на диске.
25. Выполните программу. Введите в окно редактора любое число, нажмите на кнопку **Ввод**, и Вы получите результат в окнах редакторов.
26. Закомментируйте вывод в окна `MessageBox`.
27. Добавьте на панель кнопку **Button3.** Задайте свойству *Text* значение «Очистить». Напишите обработчик события для очистки окон `textBox1 и textBox2`, и установки курсора в окно `textBox1` (метод `Focus()`).

Задание 2.

Создайте приложение в соответствии с рис. 2 для вычисления корней квадратного уравнения.

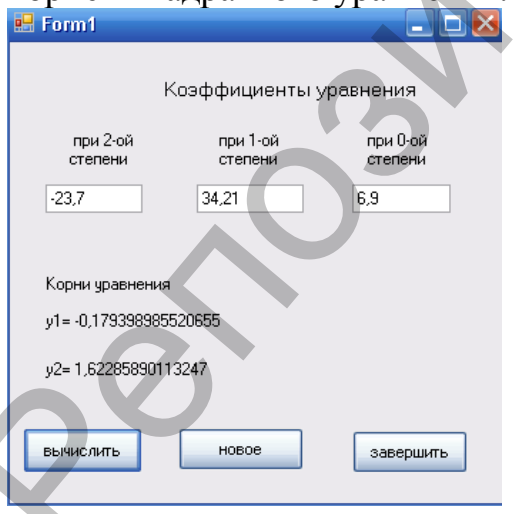

Рисунок 2. Вид окна приложения.

Создайте обработчик событий, который:

- предоставит защиту от ввода недопустимых символов в редакторы Editl, Edit2, Edit3
- будет вычислять корни квадратного уравнения
- будет подготавливать окно к новому расчету
- завершает работу приложения

ЛАБОРАТОРНАЯ РАБОТА № 9. **Использование элементов управления Common Controls и создание диалоговых окон**

Задание 1

Создайте приложение (рис. 3) для пересчета из одной системы измерения в другую, используя следующие соотношения:

1фунт=400г
1пуд=16380г
1унция=28,35г
1драхм=28,35*16г
1гран=28,35.437,5г

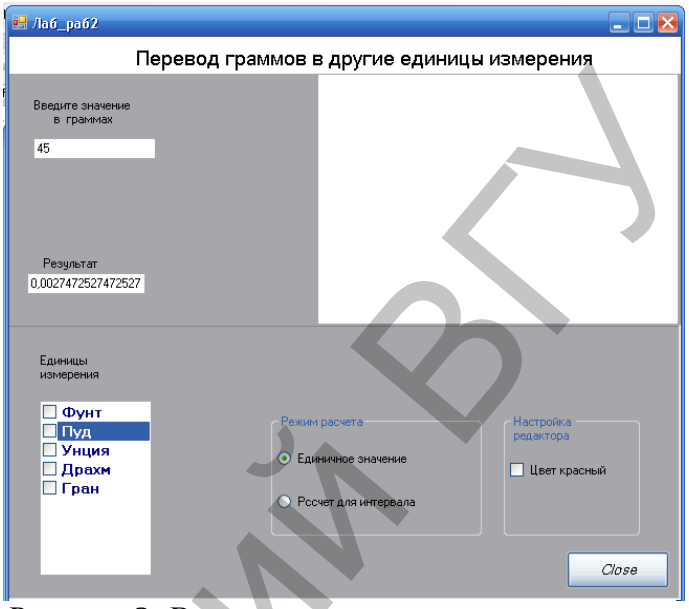

Рисунок 3. Вид окна приложения с элементами управления.

Задание 2. Создание Windows-приложений с меню

Разработать приложение, изображенное на рис. 4.

Меню приложения содержит команды Файл и Редактировать. Команда Файл содержит пункты: Сохранить, Открыть, Выход. Команда Редактировать содержит пункты Вставить, Удалить.

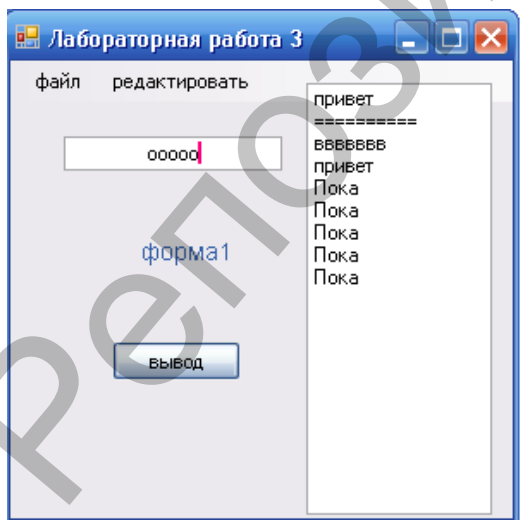

Рисунок 4. Приложение с меню.

Кнопка Вывод выводит на форму сообщение, и содержимое редактора TextBox в очередную строку списка ListBox.

При выборе *команды* Файл /Сохранить открывается диалоговое окно для сохранения в файле содержимого списка ListBox.

При выборе *команды* Файл /Открыть открывается диалоговое окно для чтения файла в список ListBox.

Команда Редактировать/ Вставить вставляет текст из TextBox в ListBox.

Команда Редактировать/ Удалить удаляет выделенный диапазон записей из ListBox.

Команда Файл /Выход закрывает окно.

ЛИТЕРАТУРА

1. Орлов, С.А. Теория и практика языков программирования: учебник для вузов. Стандарт 3-го поколения / С.А. Орлов. – СПб.: Питер, 2013. – 688 с.
2. Павловская, Т.А. C#. Программирование на языке высокого уровня / Т.А. Павловская. – СПб. [и др.]: Питер: Мир книг, 2013. – 432 с.
3. Троелсен, Э. Язык программирования C# и платформа .NET 4 / Э. Троелсен. – М.: Вильямс, 2007. – 1392 с.
4. Васильев, А. C#. Объектно-ориентированное программирование. Учебный курс. – СПб.: «Питер», 2012. – 316 с.
5. Нортроп, Т. Основы разработки приложений на платформе Microsoft .NET Framework / Т. Нортроп, Ш. Уилдермьюс, Б. Райан. – СПб.: Питер, 2007. – 864 с.
6. Пенкрат, В.В. Методы алгоритмизации: пособие для студентов высших учебных заведений, обучающихся по специальности 1-02 05 03 «Математика. Дополнительная специальность» (1-02 05 03-02 «Математика. Информатика») / В.В. Пенкрат. – Минск: БГПУ, 2012. – 107 с.
7. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C# / Дж. Рихтер. – СПб.: Питер, 2013. – 896 с.
8. Севернева, Е.В. Основы алгоритмизации и программирования: учебно-методический комплекс для студентов высших учебных заведений / Е.В. Севернева, Н.М. Жалобкевич. – Минск: БГАТУ, 2009. – 112 с.
9. Шилдт, Г. C# 4.0: полное руководство / Г. Шилдт. – М.: Вильяме, 2012. – 1056 с.
10. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C# / Дж. Рихтер. – СПб.: Питер, 2007. – 636 с.
11. Потапова, Л.Е. Алгоритмизация и программирование на языке C#: методические рекомендации к выполнению лабораторных работ / Л.Е. Потапова, Т.Г. Алейникова. – Витебск: ВГУ имени П.М. Машерова, 2014. – 50 с.
12. Справочная онлайн-библиотека для разработчиков на платформе Microsoft.NET. – Режим доступа: http://msdn.microsoft.com

Учебное издание

ПОТАПОВА Людмила Евгеньевна
АЛЕЙНИКОВА Татьяна Григорьевна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C#

Методические рекомендации
для лабораторных работ

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *Т.Е. Сафранкова*

Подписано в печать .2016. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 2,96. Уч.-изд. л. 2,11. Тираж экз. Заказ .

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014 г.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.