Лабораторная работа №06. Классы и операции в языке C#

1 Цель и порядок работы

Цель работы – ознакомиться с понятием перегрузки операторов, понятием свойств и индексаторов в языке C#.

Порядок выполнения работы:
- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Перезагрузка операторов

Введение в перегрузку операторов

Перегрузка операторов позволяет указать, как стандартные операторы будут использоваться с объектами класса. Перегрузка

Требования к перегрузке операторов:

- перегрузка операторов должна выполняться открытыми статическими методами класса (спецификаторы public static);
- у метода - оператора тип возвращаемого значения или одного из параметров должен совпадать с типом, в котором выполняется перегрузка оператора;
- параметры метода - оператора не должны включать модификатор out и ref.

Таким образом, невозможно изменить значение стандартных операций для стандартных типов данных.

Таблица 1. Операторы, допускающие перегрузку.

Операторы	Категория операторов
-	Изменение знака переменной
!	Операция логического отрицания
~	Операция побитового дополнения, которая приводит к инверсии каждого бита
++, --	Инкремент и декремент
true, false	Критерий истинности объекта, определяется разработчиком класса
+, –, *, /, %	Арифметические операторы
&, |, ^, <<, >>	Битовые операции

==, !=, <, >, <=, >=	Операторы сравнения
&&, ||	Логические операторы
[]	Операции доступа к элементам массивов моделируются за счет индексаторов
()	Операции преобразования

Таблица 2. Операторы, не допускающие перегрузку.

Операторы	Категория операторов
+=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Перегружаются автоматически при перегрузке соответствующих бинарных операций
=	Присвоение
.	Доступ к членам типа
?:	Оператора условия
new	Создание объекта
as, is, typeof	Используются для получения информации о типе
–>, sizeof,*,&	Доступны только в небезопасном коде

Перегрузка операторов имеет некоторые ограничения:

- перегрузка не может изменить приоритет операторов;
- при перегрузке невозможно изменить число операндов, с которыми работает оператор;
- не все операторы можно перегружать (таблицы 1 и 2).

Перегрузку операторов можно использовать как в классах, так и в структурах.

Поскольку перегруженные операторы являются статическими методами, они не получают указателя this, поэтому унарные операторы должны получать 1 параметр, бинарные 2.

Синтаксис перегрузки:

public static <тип результата>

operator <символ операции> (параметры)

Перегрузка унарных операторов

Можно определять в классе следующие унарные операции:

+ - ! ~ ++ -- true false

Синтаксис объявителя унарной операции:

тип operator унарнаяоперация (параметр)

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.

Пример операторов инкремента, декремента и изменения знака -.

Класс Point описывает точку на плоскости, точка имеет координаты x и y. Оператор ++ увеличивает обе координаты на 1, оператор -- уменьшает, оператор – изменяет знак координат на противоположный.

```
namespace UnaryOperator
  {
  //класс точки на плоскости – пример для перегрузки операторорв
  class CPoint
   {
    int x, y;
    public CPoint(int x, int y)
      { this.x = x; this.y = y; }
    //перегрузка инкремента
    public static CPoint operator ++(CPoint s)
      { s.x++; s.y++; return s;}
    //перегрузка декремента
    public static CPoint operator --(CPoint s)
      { s.x--; s.y--; return s; }
    //перегрузка оператора -
    public static CPoint operator -(CPoint s)
      {
```

```
    CPoint p = new CPoint(s.x, s.y);
    p.x = -p.x; p.y = -p.y; return p;
   }
  public override string ToString()
   {
    return string.Format("X = {0} Y = {1}", x, y);
   }
 }
class Test
 {
  static void Main()
   {
    CPoint p = new CPoint(10, 10);
    //префиксная и постфиксная формы выполняются одинаково
    Console.WriteLine(++p); //x=11, y=11
    CPoint p1 = new CPoint(10, 10);
    Console.WriteLine(p1++); //x=11, y=11
    Console.WriteLine(--p); //x=10, y=10
    Console.WriteLine(-p); //x=-10, y=-10
    //после выполнения оператора –
    //состояние исходного объекта не изменилось
    Console.WriteLine(p); //x=10, y=10
   }
 }
```

Результат выполнения программы изображен на рисунке 2.1.

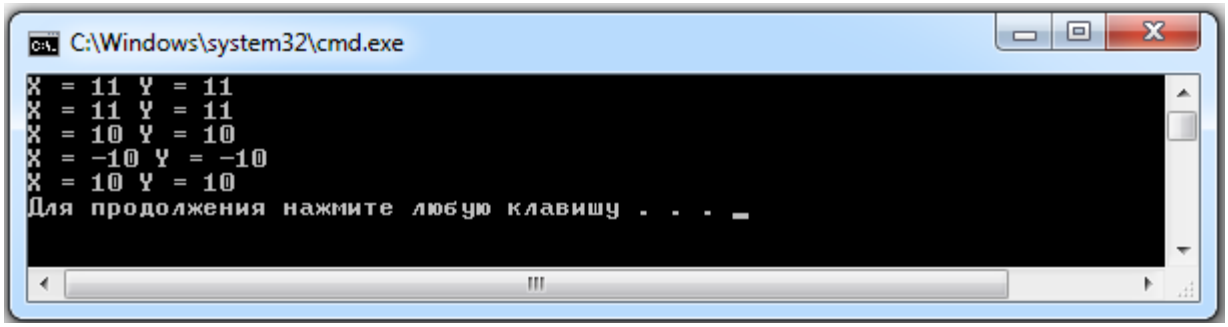

Рис. 2.1 Результат выполнения программы.

В данном примере CPoint является ссылочным типом, поэтому изменения значений x и y, которые выполняются в перегруженных операторах инкремента и декремента, изменяют переданный в них объект. Оператор – (изменение знака) не должен изменять состояние

переданного объекта, а должен возвращать новый объект с измененным знаком. Для этого в реализации этого метода создается новый объект CPoint, изменяется знак его координат и этот объект возвращается из метода.

В C# нет возможности выполнить отдельно перегрузку постфиксной и префиксной форм операторов инкремента и декремента. Поэтому при вызове постфиксная и префиксная форма работают одинаково.

При перегрузке операторов true и false разработчик задает критерий истинности для своего типа данных. После этого объекты типа напрямую можно использовать в структуре операторов if, do, while, for в качестве условных выражений.

Перегрузка выполняется по следующим правилам:

- оператор true должен возвращать значение true, если состояние объекта истинно и false в противном случае;
- оператор false должен возвращать значение true, если состояние объекта ложно и false в противном случае;
- операторы true и false надо перегружать в паре.

При этом возможна ситуация, когда состояние не является не истинным ни ложным, т.е. оба оператора могут вернуть результат false.

При перегрузке операторов true и false используется следующая таблица истинности (табл. 3).

Таблица 3. Таблица истинности при перегрузке операторов true и false.

Значение	Оператор True	Оператор False
1	true	False
-1	false	True
0	false	false

```
public struct DBBool
   {
      //три возможных значения
      // Значение параметра может быть: - 1(false), 1 - true и 0 - null.
      public static readonly DBBool Null = new DBBool(0);
      public static readonly DBBool False = new DBBool(-1);
      public static readonly DBBool True = new DBBool(1);
      sbyte value;
      DBBool(int value)
```

```
    {  this.value = (sbyte)value; }
    public DBBool(DBBool b)
    { this.value = (sbyte)b.value; }
    // Возвращает true, если в операнде содержится True,
    // иначе возвращает false
    public static bool operator true(DBBool x)
    { return x.value > 0; }
    // Возвращает true, если в операнде содержится False,
    // иначе возвращает false
    public static bool operator false(DBBool x)
    { return x.value < 0; }
}
class Test
{
    static void Main()
    {
        DBBool b1 = new DBBool(DBBool.True);
        if (b1) Console.WriteLine("b1 is true");
        else Console.WriteLine("b1 is not true");
    }
}
```

Как видно из реализации, если в объекте класса содержится значение null, то оба оператора возвращают значение false.

Перегрузка бинарных операторов

Можно определять в классе следующие бинарные операции:

+ - * / % & ≪ ≫ == != > < >= <=

Синтаксис объявителя бинарной операции:

тип operator бинарная_операция (параметр1, параметр2)

Примеры заголовков бинарных операций:

Cjblic static MyObject operator + (MyObject ml, MyObject m2)

Cublic 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. Чаще всего в классе определяют операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение объектов, а не их ссылок, как определено по умолчанию для ссылочных типов.

Пример перегрузки бинарных операций:

```
namespace BinaryOperator
 {
  class CVector
  {
    public int x;
    public int y;
    public CVector(int x, int y)
     { this.x = x; this.y = y; }
    public override string ToString()
     { return string.Format("Vector: X = {0} Y = {1}", x, y); }
  }
 class CPoint
  {
    private int x;
    private int y;
    public CPoint(int x, int y)
     { this.x = x; this.y = y; }
    //перегрузка бинарного оператора +
    public static CPoint operator +(CPoint p, CVector v)
     { return new CPoint(p.x + v.x, p.y + v.y); }
    //перегрузка бинарного оператора *
    public static CPoint operator *(CPoint p, int a)
     { return new CPoint(p.x * a, p.y * a); }
    //перегрузка бинарного оператора -
    public static CVector operator -(CPoint p1, CPoint p2)
     { return new CVector(p1.x - p2.x, p1.y - p2.y); }
    public override string ToString()
     { return string.Format("Point: X = {0} Y = {1}", x, y); }
  }
 class Program
```

```
{
 static void Main(string[] args)
 {
    CPoint p1 = new CPoint(10, 10);
    CPoint p2 = new CPoint(12, 20);
    CVector v = new CVector(10, 20);
    Console.WriteLine("Точка p1: {0}", p1);
    Console.WriteLine("Сдвиг: {0}", p1 + v);
    Console.WriteLine("Масштабирование: {0}", p1 * 10);
    Console.WriteLine("Точка p2: {0}", p2);
    Console.WriteLine("Расстояние: {0}", p2 - p1);
 }
}
}
```

Результат выполнения программы изображен на рисунке 2.2.

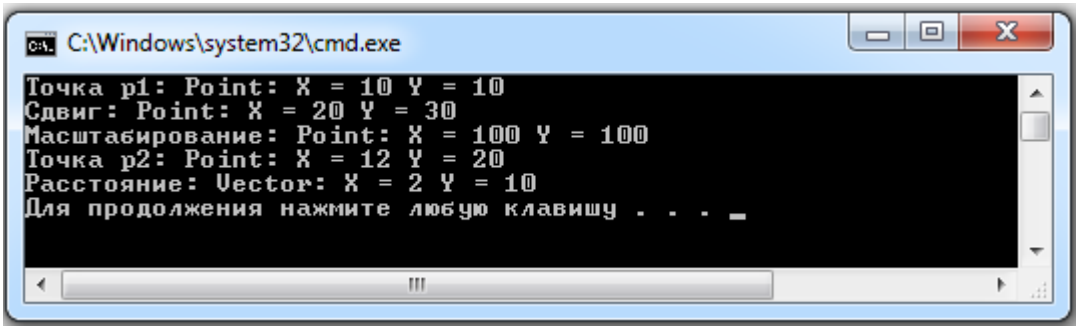

Рис. 2.2 Результат выполнения программы.

Выполненная перегрузка автоматически перегружает операторы +=, *=, -=. Например, можно записать: p1+= v;

Для реализации этого действия будет использован перегруженный оператор +.

Однако перегруженные в примере операторы будут использоваться компилятором только если переменная типа CPoint находится слева от знака операнда. Т.е. выражение p * 10 откомпилируется нормально, а при перестановке сомножителей, т.е. в выражении 10 * p произойдет ошибка компиляции. Для исправления этой ошибки следует перегрузить оператор * с другим порядком операндов:

```
public static CPoint operator *(int a, CPoint p)
{ return p * a; }
```

При перегрузке операторов отношения надо учитывать, что есть два способа проверки равенства:

- равенство ссылок (тождество);

- равенство значений.

В классе Object определены следующие методы сравнения объектов:

public static bool ReferenceEquals(Object obj1, Object obj2)

public bool virtual Equals(Object obj)

Есть отличия в работе этих методов со значимыми и ссылочными типами.

Метод ReferenceEquals() проверяет, указывают ли две ссылки на один и тот же экземпляр класса; точнее - содержат ли две ссылки один и тот адрес памяти. Этот метод невозможно переопределить. Со значимыми типами ReferenceEquals() всегда возвращает false, т.к. при сравнении выполняется приведение к Object и упаковка, упакованные объекты располагаются по разным адресам.

Метод Equals() является виртуальным. Его реализация в Object выполняется проверку равенства ссылок, т.е. работает так же как и ReferenceEquals. Для значимых типов в базовом типе System.ValueType выполнена перегрузка метода Equals(), которая выполняет сравнение объектов путем сравнения всех полей (побитовое сравнение).

Пример использования операторов ReferenceEquals() и Equals() со ссылочными и значимыми типами:

```
namespace Equals_and_ReferenceEquals
  {
    class CPoint
    {   private int x, y;
       public CPoint(int x, int y)
       { this.x = x; this.y = y; }
    }
    struct SPoint
    {
       private int x, y;
       public SPoint(int x, int y)
       { this.x = x; this.y = y; }
    }
    class Program
    {
       static void Main()
       {
 //Работа метода ReferenceEquals с ссылочным и значимым типами
          //ссылочный тип
```

```
            CPoint p = new CPoint(0, 0);
            CPoint p1 = new CPoint(0, 0);
            CPoint p2 = p1;
            Console.WriteLine("ReferenceEquals(p, p1)= {0}",
            ReferenceEquals(p, p1)); //false
            //хотя p,p1содержат одинаковые значения,
            //они указывают на разные адреса памяти
            Console.WriteLine("ReferenceEquals(p1, p2)= {0}",
            ReferenceEquals(p1, p2));//true
            //p1 и p2 указывают на один и тот же адрес памяти
            //значимый тип
            SPoint p3 = new SPoint(0, 0);
    //при передаче в метод ReferenceEquals выполняется упаковка,
    //упакованные объекты располагаются по разным адресам
            Console.WriteLine("ReferenceEquals(p3, p3) = {0}",
            ReferenceEquals(p3, p3)); //false
            //Работа метода Equals с ссылочным и значимым типами
            //ссылочный тип
            CPoint cp = new CPoint(0, 0);
            CPoint cp1 = new CPoint(0, 0);
      Console.WriteLine("Equals(cp, cp1) = {0}", Equals(cp, cp1)); //false
            //выполняется сравнение адресов значимый тип
            SPoint sp = new SPoint(0, 0);
            SPoint sp1 = new SPoint(0, 0);
      Console.WriteLine("Equals(sp, sp1) = {0}", Equals(sp, sp1)); //true
            //выполняется сравнение значений полей
          }
        }
    }
```

Результат выполнения программы изображен на рисунке 2.3.

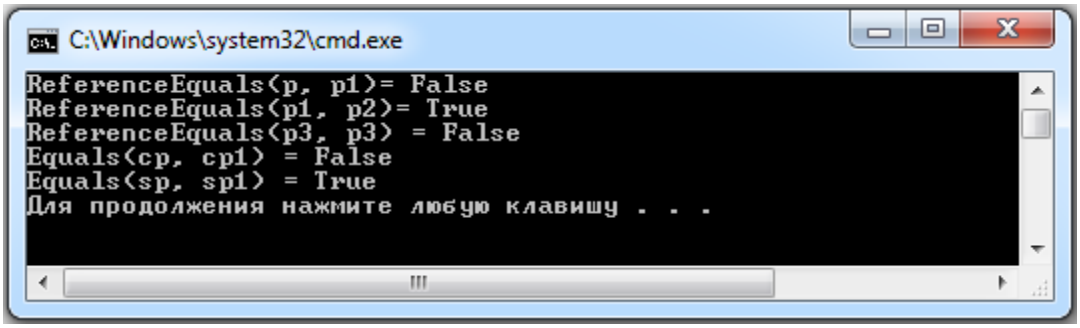

Рис. 2.3 Результат выполнения программы.

При создании собственного типа оператор Equals() можно перегрузить. Для ссылочных типов перегрузку следует выполнять, только если тип представляет собой неизменяемый объект. Например, для типа String, который содержит в себе неизменную строку, имеется перегруженный метод Equals() и оператор ==.

Поскольку в System.ValueType перегруженный метод Equals() выполняет побитовое сравнение, то в собственных значимых типах его можно не перегружать. Однако, в System.ValueType получение значений полей для сравнения в методе Equals() выполняется с помощью рефлексии, что приводит к снижению производительности. Поэтому при разработке значимого типа для увеличения быстродействия рекомендуется выполнить перегрузку метода Equals().

При перегрузке метода Equals() следует также перегружать метод GetHashCode(). Этот метод предназначен для получения целочисленного значения хеш - кода объекта. Причем, различным (т.е. не равным между собой) объектам должны соответствовать различные хеш – коды. Если перегрузку метода GetHashCode() не выполнить возникнет предупреждение компилятора.

Перегрузка оператора == обычно выполняется путем вызова метода Equals().

Если предполагается сравнивать экземпляры собственного типа для целей сортировки, рекомендуется унаследовать этот тип от интерфейсов System.IComparable и System.IComparable<T> и реализовать метод CompareTo(). В дальнейшем этот метод можно вызывать из реализации Equals() и возвращать true, если CompareTo() возвращает 0.

Пример перегрузки операторов == и != для класса CPoint:

```
namespace ComparisonOperator
  {
    using System;
    class CPoint
    {
      int x, y;
      public CPoint(int x, int y)
```

```
    { this.x = x; this.y = y; }
    //Перегрузка метода Equals
    public override bool Equals(object obj)
    {
      //если obj == null,
//значит он != объекту, от имени которого вызывается этот метод
      if (obj == null) return false;
      CPoint p = obj as CPoint;
      //переданный объект не является ссылкой на CPoint
      if (p == null) return false;
      //проверяется равенство содержимого
      return ((x == p.x) && (y == p.y));
    }
 //При перегрузке Equals надо также перегрузить GetHashCode()
    public override int GetHashCode()
    {
      return x ^ y;//использование XOR для получения хеш кода
    }
    public static bool operator ==(CPoint p1, CPoint p2)
    {
  //проверка, что переменные ссылаются на один и тот же адрес
      //сравнение p1 == p2 приведет к бесконечной рекурсии
      if (ReferenceEquals(p1, p2)) return true;
      //приведение к object необходимо,
//т.к. сравнение p1 == null приведет к бесконечной рекурсии
      if ((object)p1 == null) return false;
      return p1.Equals(p2);
    }
    public static bool operator !=(CPoint p1, CPoint p2)
    { return (!(p1 == p2)); }
  }
  class Program
  {
    static void Main(string[] args)
    {
```

```
            CPoint cp = new CPoint(0, 0);
            CPoint cp1 = new CPoint(0, 0);
            CPoint cp2 = new CPoint(1, 1);
            Console.WriteLine("cp == cp1: {0}", cp == cp1); //true
            Console.WriteLine("cp == cp1: {0}", cp == cp2); //false
        }
    }
}
```

Условные логические операторы && и || нельзя перегрузить, но они вычисляются с помощью & и |, допускающих перегрузку.

Пример класса DBBool использующего перегрузку логических операторов:

```
public struct DBBool
  {
    //три возможных значения
    // Значение параметра может быть: - 1(false), 1 - true и 0 - null.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);
    sbyte value;
    DBBool(int value)
    { this.value = (sbyte)value; }
    public DBBool(DBBool b)
    { this.value = (sbyte)b.value; }
    // Возвращает true, если в операнде содержится True,
    // иначе возвращает false
    public static bool operator true(DBBool x)
    { return x.value > 0; }
    // Возвращает true, если в операнде содержится False,
    // иначе возвращает false
    public static bool operator false(DBBool x)
    { return x.value < 0; }
    // Оператор Логическое И. Возвращает:
    // False, если один из операндов False независимо
    // от 2-ого операнда
    // Null, если один из операндов Null, а другой Null или True
```

```
    // True, если оба операнда True
    public static DBBool operator &(DBBool x, DBBool y)
    { return new DBBool(x.value < y.value ? x.value : y.value); }
    // Оператор Логическое ИЛИ. Возвращает:
    // True, если один из операндов True независимо
   // от 2-ого операнда
    // Null, если один из операндов Null, а другой False или Null
    // False, если оба операнда False
    public static DBBool operator |(DBBool x, DBBool y)
    { return new DBBool(x.value > y.value ? x.value : y.value); }
    // Оператор Логической инверсии. Возвращает:
    // False, если операнд содержит True
    // True, если операнд содержит False
    // Null, если операнд содержит Null
    public static DBBool operator !(DBBool x)
    { return new DBBool(-x.value); }
    public override string ToString()
    {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
class Test
{
    static void Main()
    {
        DBBool bTrue = new DBBool(DBBool.True);
        DBBool bNull = new DBBool(DBBool.Null);
        DBBool bFalse = new DBBool(DBBool.False);
        Console.WriteLine("bTrue && bNull is {0}", bTrue && bNull);
        Console.WriteLine("bTrue && bFalse is {0}", bTrue && bFalse);
        Console.WriteLine("bTrue && bTrue is {0}", bTrue && bTrue);
        Console.WriteLine();
        Console.WriteLine("bTrue || bNull is {0}", bTrue || bNull);
```

```
            Console.WriteLine("bFalse || bFalse is {0}", bFalse || bFalse);
            Console.WriteLine("bTrue || bFalse is {0}", bTrue || bFalse);
            Console.WriteLine();
            Console.WriteLine("!bTrue is {0}", !bTrue);
            Console.WriteLine("!bFalse is {0}", !bFalse);
            Console.WriteLine("!bNull is {0}", !bNull);
        }
    }
```

Результат выполнения программы изображен на рисунке 2.4.

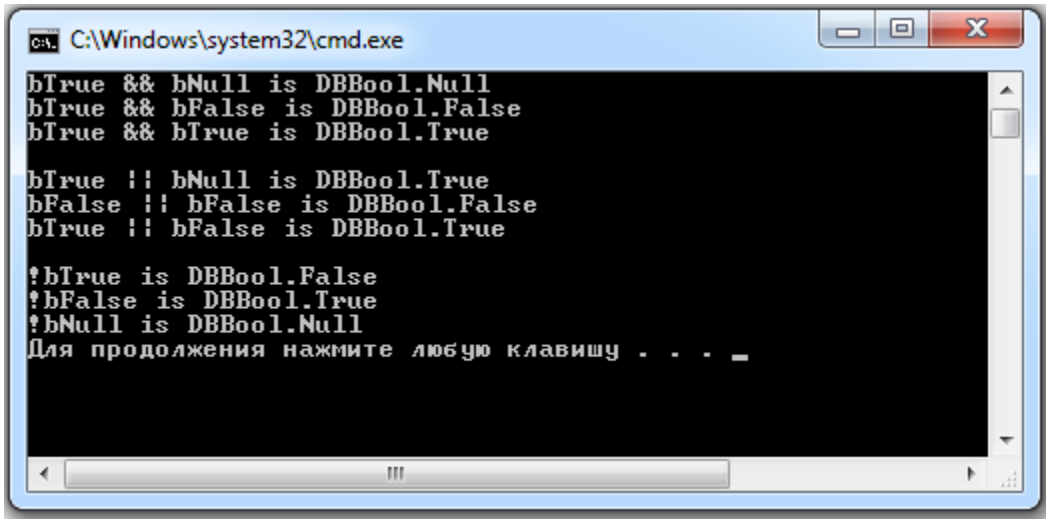

Рис. 2.4 Результат выполнения программы.

Перегрузка операторов преобразования типа

Операции преобразования типа обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных. Синтаксис объявителя операции преобразования типа:

implicit operator тип (параметр) // неявное преобразование

explicit operator тип (параметр) // явное преобразование

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого определяется операция. Таким образом, операции выполняют преобразования либо типа класса к другому типу, либо наоборот. Преобразуемые типы не должны быть связаны отношениями наследования.

Конструктор с 1-им параметром не используется для преобразования произвольного типа в собственный тип. Для ссылочных и значимых типов приведение выполняется одинаково.

Приведение может выполняться явным и неявным образом. Явное приведение типов требуется, если возможна потеря данных в результате приведения. Например:

- при преобразовании int в short, потому что размер short недостаточен для сохранения значения int;
- при преобразовании типов данных со знаком в беззнаковые может быть получен неверный результат, если переменная со знаком содержит отрицательное значение;
- при конвертировании типов с плавающей точкой в целые дробная часть теряется;
- при конвертировании типа, допускающего null-значения, в тип, не допускающий null, если исходная переменная содержит null, генерируется исключение.

Если потери данных в результате приведения не происходит приведение можно выполнять как неявное.

Пример:

Приведение CPoint к типу int с потерей точности, к типу double без потери точности. В качестве результата возвращается расстояние от точки до начала координат.

Приведение типа int к CPoint без потери точности, типа double к CPoint с потерей точности. В качестве результата возвращается точка, содержащее заданное значение в качестве значений координат.

```
namespace CastOperator
  {
    class CPoint
    {
      int x, y;
      public CPoint(int x, int y)
      { this.x = x; this.y = y; }
//может быть потеря точности, преобразование должно быть явным
      public static explicit operator int(CPoint p)
      {
        return (int)Math.Sqrt(p.x * p.x + p.y * p.y);
        //можно и так: return (int)(double)p;
      }
      //преобразование без потери точности, может быть неявным
      public static implicit operator double(CPoint p)
      { return Math.Sqrt(p.x * p.x + p.y * p.y); }
```

```
        //переданное значение сохраняется в x и y координате,
        //преобразование без потери точности, может быть неявным
        public static implicit operator CPoint(int a)
        { return new CPoint(a, a); }
        //преобразование с потерей точности, должно быть явным
        public static explicit operator CPoint(double a)
        {
          return new CPoint((int)a, (int)a);
         }
        public override string ToString()
        {
         return string.Format("X = {0} Y = {1}", x, y);
         }
    }
    class Test
    {
        static void Main()
        {
          CPoint p = new CPoint(2, 2);
           //выполнение явного преобразования CPoint в int
           int a = (int)p;
           //выполнение неявного преобразования CPoint в double
           double d = p;
           Console.WriteLine("p as int: {0}", a); //2
           Console.WriteLine("p as double: {0:0.0000}", d);          //2.8284
           p = 5; //выполнение неявного преобразования int в CPoint
           Console.WriteLine("p: {0}", p); //x = 5 y = 5
          //выполнение явного преобразования double в CPoint
           p = (CPoint)2.5;
           Console.WriteLine("p: {0}", p); //x = 2 y = 2
        }
    }
}
```

Результат выполнения программы изображен на рисунке 2.5.

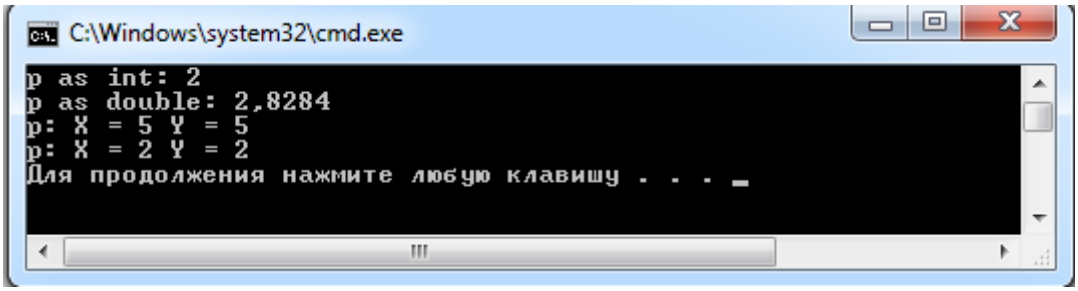

Рис. 2.5 Результат выполнения программы.

Имеется возможность выполнять приведение между экземплярами разных собственных структур или классов. Однако при этом существуют следующие ограничения:

- нельзя определить приведение между классами, если один из них является наследником другого;
- приведение может быть определено только в одном из типов: либо в исходном типе, либо в типе назначения.

Например, имеется следующая иерархия классов (рис. 2.6).

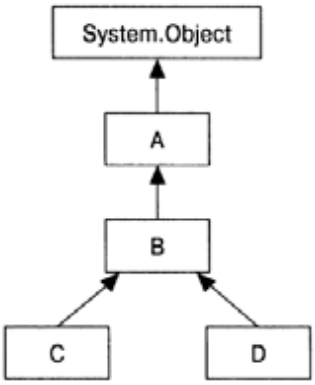

Рис. 2.6 Пример иерархии классов.

Единственное допустимое приведение типов – это приведения между классами C и D, потому что эти классы не наследуют друг друга. Код таких приведений может выглядеть следующим образом:

```
public static explicit operator D(C value) {…}
public static explicit operator C(D value) {…}
```

Эти операторы могут быть внутри определения класса C или же внутрь определения класса D. Если приведение определено внутри одного класса, то нельзя определить такое же приведение внутри другого.

2.2 Свойства

Свойства служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки.

Синтаксис свойства:

[атрибуты] [спецификаторы] тип имя_свойства

{

[get код_доступа]

[set код_доступа]

}

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором public), поскольку они входят в интерфейс объекта.

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).

В C# введена удобная возможность задавать разные уровни доступа для частей **get** и **set**. Например, во многих классах возникает потребность обеспечить неограниченный доступ для чтения и ограниченный — для записи.

Спецификаторы доступа для отдельной части должны задавать либо такой же, либо более ограниченный доступ, чем спецификатор доступа для свойства в целом. Например, если свойство описано как public, его части могут иметь любой спецификатор доступа, а если свойство имеет доступ protected internal, его части могут объявляться как internal, protected или private. Синтаксис свойства имеет вид

[атрибуты] [спецификаторы] тип имя_свойства

{

[[атрибуты] [спецификаторы] get код_доступа]

[[атрибуты] [спецификаторы] set код_доступа]

}

Пример описания свойств:

```
public class Button : Control
   {
      // закрытое поле, с которым связано свойство
      private string caption;
      public string Caption
      { // свойство
         get
         { // способ получения свойства
```

```
            return caption;
        }
        set
        { // способ установки свойства
            if (caption != value)
            {
                caption = value;
            }
        }
    }
…..
  }
```

При обращении к свойству автоматически вызываются указанные в нем методы чтения и установки.

Синтаксически чтение и запись свойства выглядят почти как методы. Метод **get** должен содержать оператор **return**, возвращающий выражение, для типа которого должно существовать неявное преобразование к типу свойства. В методе **set** используется параметр со стандартным именем **value**, который содержит устанавливаемое значение.

Вообще говоря, свойство может и не связываться с полем. Фактически, оно описывает один или два метода, которые осуществляют некоторые действия над данными того же типа, что и свойство. В отличие от открытых полей, свойства обеспечивают разделение между внутренним состоянием объекта и его интерфейсом и, таким образом, упрощают внесение изменений в класс.

С помощью свойств можно отложить инициализацию поля до того момента, когда оно фактически потребуется, например:

```
class A
  {
    private static ComplexObject x;// закрытое поле
    public static ComplexObject X  // свойство
    {
      get
      {
        if (x == null)
        {
          // создание объекта при 1-м обращении
```

```
            x = new ComplexObject();
        }
        return x;
    }
  }
…….
  }
```

Пример: Класс Employee реализован с четырьмя приватными полями, обозначающими имя, фамилию, возраст и зарплату сотрудника. Класс также включает два перегруженных конструктора (с параметрам и без параметров). Для каждого из полей класса предусмотрено открытое свойство с двумя методами get и set. В свойствах осуществляется дополнительная проверка задаваемых значений. Имя и фамилия приводятся к верхнему регистру, возраст проверяется на принадлежность интервалу допустимых значений, зарплат не может быть отрицательной величиной. Перегруженный метод ToString() позволяет распечатать состояние объекта.

```
class Emoloyee
  {
    private string firstName;
    private string lastName;
    private int age;
    private float wage;
    public Emoloyee()
    {
    }
    public Emoloyee(string first, string last, int age, float wage)
    {
      this.FirstName = first;
      this.LastName = last;
      this.Age = age;
      this.Wage = wage;
    }
    public string FirstName
    {
      get { return firstName != null ? firstName : "Not set"; }
```

```
        set { firstName = value.ToUpper(); }
    }
    public string LastName
    {
        get { return lastName != null ? lastName : "Not set"; ; }
        set { lastName = value.ToUpper(); }
    }
    public int Age
    {
        get { return age; }
        set { age = (value > 100 || value < 1) ? 0 : value; }
    }
    public float Wage
    {
        get { return wage; }
        set { wage = value < 0 ? 0 : value; }
    }
    public override string ToString()
    {
        return string.Format("First name: {0}\nLast name:
                                        {1}\nAge:{2}\nWage: {3}\n",
        this.FirstName, this.LastName, this.Age, this.Wage);
    }
}
class Tester
{
    static void Main(string[] args)
    {
        Emoloyee emp1 = new Emoloyee("Oleg", "Sikorsky", 29,
                                                    4800F);
        Emoloyee emp2 = new Emoloyee();
        emp2.FirstName = "Daniel";
        //Last name не установлено
        //попытка присвоить невозможный возраст
        emp2.Age = 120;
```

```
            //попытка задать зарплату со знаком минус
            emp2.Wage = -1000;
            Emoloyee emp3 = new Emoloyee("Natali", "Borisova", 29,
                                                        2500F);
            Console.WriteLine(emp1.ToString());
            Console.WriteLine(emp2.ToString());
            Console.WriteLine(emp3.ToString());
        }
    }
```

Результат выполнения программы изображен на рисунке 2.7.

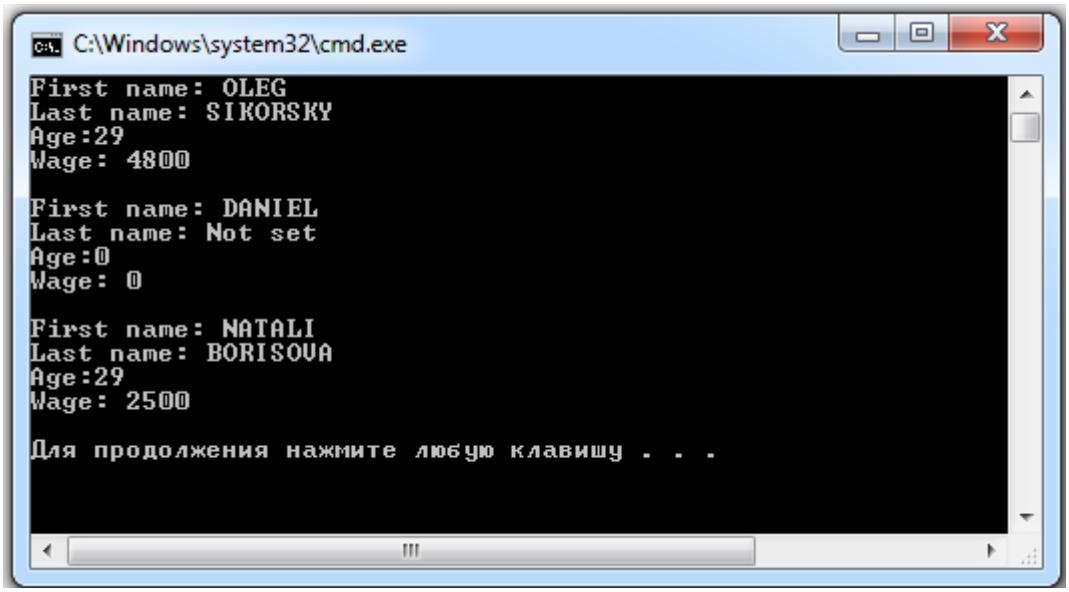

Рис. 2.7 Результат выполнения программы.

2.3 Индексаторы

Понятие индексатора

Индексатор представляет собой разновидность свойства. Если у класса есть скрытое поле, представляющее собой массив, то с помощью индексатора можно обратиться к элементу этого массива, используя имя объекта и номер элемента массива в квадратных скобках.

Объявление индексатора подобно свойству, но с той разницей, что индексаторы безымянные (вместо имени используется ссылка this) и что индексаторы включают параметры индексирования.

Синтаксис объявления индексатора следующий:

тип this [тип аргумента] {get; set;}

Тип – это тип объектов коллекции. This - это ссылка на объект, в котором появляется индексатор. Тип аргумента представляет индекс объекта в коллекции, причём этот индекс необязательно целочисленный, как мы привыкли, он может быть любого типа.

У каждого индексатора должен быть минимум один параметр, но их может быть и больше (напоминает многомерные массивы).

То, что для индексаторов используется синтаксис со ссылкой this, подчёркивает, что нельзя использовать их иначе, как на экземплярном уровне.

Создание одномерного индексатора

Пример создания и применения индексатора. Предположим, есть некий магазин, занимающийся реализацией ноутбуков. Эта ситуация отображается при помощи двух классов: класса Shop, изображающего магазин, и класса Laptop, изображающего его продукцию. Дабы не перегружать пример лишней информацией, снабдим класс Laptop только двумя полями: vendor – для отображения имени фирмы-производителя, а также price – для отображения цены ноутбука. Класс будет включать соответствующие открытые свойства Vendor и Price, конструктор с двумя параметрами, а также переопределённый метод ToString() для отображения информации по конкретной единице товара. В качестве единственного поля класса Shop выступает ссылка на массив объектов Laptop. В конструкторе с одним параметром задаётся количество элементов массива и выделяется память для их хранения. Далее нам нужно сделать возможным обращение к элементам этого массива через экземпляр класса Shop, пользуясь синтаксисом массива так, словно класс Shop и есть массив элементов типа Laptop. Для этого добавлен в класс Shop индексатор:

```
public Laptop this[int pos]
        {
           get
           {
              if (pos >= LaptopArr.Length || pos < 0)
                 throw new IndexOutOfRangeException();
              else
                 return (Laptop)LaptopArr[pos];
           }
           set
           { LaptopArr[pos] = (Laptop)value; }
        }
```

Здесь в аксессоре get мы предусматриваем выход за пределы массива, и при этом генерируется исключение IndexOutOfRangeException.

Для проверки работы классов создаётся отдельный класс Tester, содержащий точку входа. В нём создаётся экземпляр класса Shop, причём в конструкторе задается количество элементов, которые в нём можно разместить.

```
Shop sh = new Shop(3);
```

Далее мы заполняем этот массив объектами Laptop.

```
sh[0] = new Laptop("Samsung", 5200);
sh[1] = new Laptop("Asus", 4700);
sh[2] = new Laptop("LG", 4300);
```

И, наконец, вывод на экран данных по каждому объекту Laptop, пользуясь синтаксисом массива.

```
for (int i = 0; i < 3; i++)
Console.WriteLine(sh[i].ToString());
```

В цикле ограничивающим значением в условии является явно заданное число 3. Дело в том, что индексатор позволяет нам пользоваться лишь синтаксисом индексирования массива, но других функциональных возможностей массива не предоставляет. Если это был бы стандартный массив, то это условие описывается иначе:

```
for (int i = 0; i < sh.Length; i++)
. . .
```

Для того, чтобы подобная функциональная возможность появилась и в примере, необходимо добавить в класс Shop дополнительное свойство Length.

```
public int Length
{
get { return LaptopArr.Length; }
}
```

Общий вид программы:

```
namespace NS
   {
      public class Laptop
      {
         private string vendor;
         private double price;
         public string Vendor
         {
```

```
        get { return vendor; }
        set { vendor = value; }
    }
    public double Price
    {
        get { return price; }
        set { price = value; }
    }
    public Laptop(string v, double p)
    {
        vendor = v;
        price = p;
    }
    public override string ToString()
    {
        return vendor + " " + price.ToString();
    }
}
public class Shop
{
    private Laptop[] LaptopArr;
    public Shop(int size)
    {
        LaptopArr = new Laptop[size];
    }
    public int Length
    {
        get { return LaptopArr.Length; }
    }
    public Laptop this[int pos]
    {
        get
        {
            if (pos >= LaptopArr.Length || pos < 0)
                throw new IndexOutOfRangeException();
```

```
            else
                return (Laptop)LaptopArr[pos];
        }
        set
        {
            LaptopArr[pos] = (Laptop)value;
        }
    }
}
public class Tester
{
    public static void Main()
    {
        Shop sh = new Shop(3);
        sh[0] = new Laptop("Samsung", 5200);
        sh[1] = new Laptop("Asus", 4700);
        sh[2] = new Laptop("LG", 4300);
        try
        {
            for (int i = 0; i < sh.Length; i++)
                Console.WriteLine(sh[i].ToString());
            Console.WriteLine();
        }
        catch (System.NullReferenceException)
        { }
    }
}
```

Результат выполнения программы изображен на рисунке 2.8.

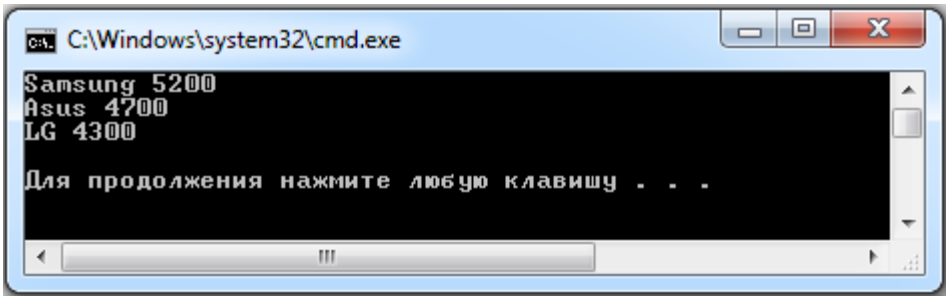

Рис. 2.8 Результат выполнения программы.

Создание многомерных индексаторов

В C# есть возможность создавать не только одномерные, но и многомерные индексаторы. Это возможно, если класс-контейнер содержит в качестве поля массив с более чем одним измерением.

Пример использования многомерного индексатора:

```
namespace NS
{
   public class A
   {
      private int[,] arr;
      private int rows, cols;
      public int Rows
      {
         get { return rows; }
      }
      public int Cols
      {
         get { return cols; }
      }
      public A(int rows, int cols)
      {
         this.rows = rows;
         this.cols = cols;
         arr = new int[rows, cols];
      }
      public int this[int r, int c]
      {
         get { return arr[r, c]; }
         set { arr[r, c] = value; }
      }
   }
   public class Tester
   {
```

```
        static void Main()
        {
            A obj = new A(2, 3);
            for (int i = 0; i < obj.Rows; i++)
            {
                for (int j = 0; j < obj.Cols; j++)
                {
                    obj[i, j] = i + j;
                    Console.Write(obj[i, j].ToString());
                }
                Console.WriteLine();
            }
        }
    }
```

2.4 Деструкторы

В C# существует специальный вид метода, называемый деструктором. Он вызывается сборщиком мусора непосредственно перед удалением объекта из памяти. В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта, например, проверяется, все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т.п.).

Синтаксис деструктора:

[атрибуты] [extern] ~имя_класса()

тело

Как видно из определения, деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется тильдой (~), символизирующей обратные по отношению к конструктору действия. Тело деструктора представляет собой блок или просто точку с запятой, если деструктор определен как внешний (extern).

Сборщик мусора удаляет объекты, на которые нет ссылок. Он работает в соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени. Поскольку деструктор вызывается сборщиком мусора, невозможно гарантировать, что деструктор будет обязательно вызван в процессе работы программы. Следовательно, его

лучше использовать только для гарантии освобождения ресурсов, а «штатное» освобождение выполнять в другом месте программы.

Применение деструкторов замедляет процесс сборки мусора.

2.5 Вложенные типы

В классе можно определять типы данных, внутренние по отношению к классу. Так определяются вспомогательные типы, которые используются только содержащим их классом. Механизм вложенных типов позволяет скрыть ненужные детали и более полно реализовать принцип инкапсуляции. Непосредственный доступ извне к такому классу невозможен (имеется в виду доступ по имени без уточнения). Для вложенных типов можно использовать те же спецификаторы, что и для полей класса.

Например, класс Monster содержит вспомогательный класс Gun. Объекты этого класса без «хозяина» бесполезны, поэтому его можно определить как внутренний:

```
using System;
namespace ConsoleApplicationi
{ class Monster
   {
     class Gun
       { ….. }
…..
   }
}
```

Помимо классов вложенными могут быть и другие типы данных: интерфейсы, структуры и перечисления.

2.6 Исключения

Иерархия исключений

Возникновение ошибок при выполнении приложения - не всегда следствие ошибок в коде приложения. Ошибки могут быть вызваны неправильными действиями пользователя или внешними причинами такими как аппаратные сбои, недоступность некоторых ресурсов (например, сетевого диска или сервера базы данных).

Таким образом, профессионально разработанное приложение должно быть готово к возникновению ошибок и должно обеспечивать их обработку.

Для обработки ошибок можно использовать различные подходы. В WinAPI многие функции при неуспешном выполнении возвращают признак ошибки (например, FALSE, INVALID_HANDLE_VALUE, NULL). Далее с помощью функции GetLastError() можно получить код ошибки и найти по этому коду ее описание. Такой способ обработки ошибок является трудоемким, т.к. после вызова функции надо проверять результат возврата, и не вполне надежным, т.к. можно продолжить работу без проверки результата, так как будто функция завершилась успешно.

В .Net Framework способ обработки ошибок значительно улучшен. Все методы при неуспешном выполнении генерируют исключения. Этот подход имеет следующие преимущества:

- Обработку ошибок можно отделить от основной логики работы программы: всю обработку ошибок можно сосредоточить в одном блоке, а не выполнять проверку после каждого вызова метода.

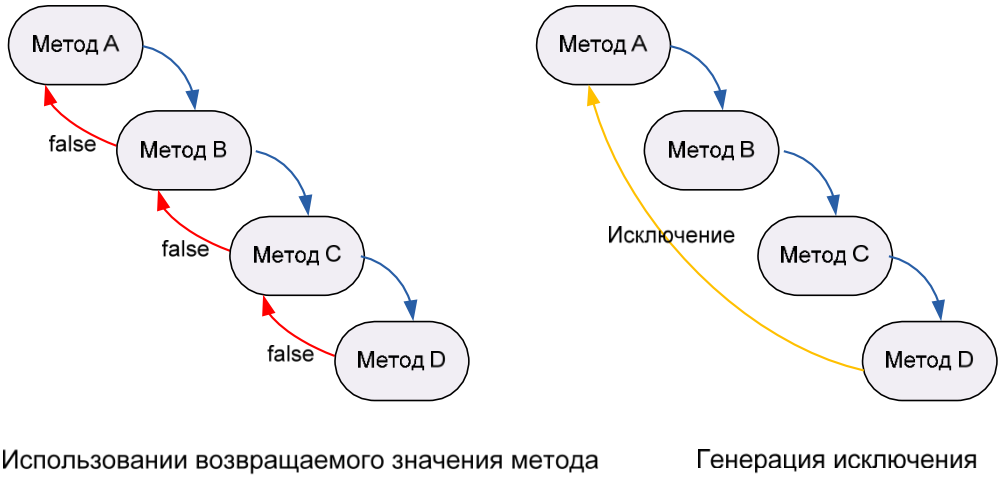

Рис. 2.9. Передача информации о возникновении ошибки по цепочке вызовов

- Возможна ситуация, когда имеется цепочка вызовов (рис. 2.9.), и в последнем методе цепочки возникает ошибка. Без обработки исключений каждый из методов должен вернуть вызывающему методу код ошибки, в случае использования исключений при возникновении ошибки управление будет сразу передано методу, содержащему обработчик исключения.
- Исключение нельзя проигнорировать, т.к. необработанное исключение приведет к аварийному завершению программы.

Базовый класс System.Exception.

Таблица 4. Свойства класса System.Exception

Название свойства	Описание
string Message	Содержит текст сообщения с указанием причины возникновения исключения.
IDictionary Data	Ссылка на набор пар «параметр-значение». Обычно код, генерирующий исключение, добавляет записи в этот набор. Код, перехвативший исключение, может использовать эти данные для получения дополнительной информации о причине возникновения исключения.
string Source	Содержит имя сборки, сгенерировавшей исключение.
string StackTrace	Содержит имена и сигнатуры методов, вызов которых привел к возникновению исключения.
MethodBase TargetSite	Содержит метод, сгенерировавший исключение.
string HelpLink	Содержит URL документа с описанием исключения.
Exception InnerException	Указывает предыдущее исключение, если текущее было сгенерировано при обработке предыдущего исключения.

В C# исключение является объектом, который создается и «выбрасывается» (throw) в случае возникновения ошибки. CLR позволяет генерировать исключения любого типа, например Int32, String и др. CLS-совместимый язык должен быть способен генерировать и перехватывать типы исключений, производные от базового класса SystemException (таб. 4). Эти исключения называются CLS-совместимыми. Такое исключение несет дополнительную информацию об ошибке, которая облегчает отладку программы.

Существует множество классов исключений (наследников System.Exception), разработчик может использовать эти классы и также создавать собственные классы исключений. Все исключения делятся на 2 группы: SystemException и ApplicationException.

SystemException – это класс исключений, которые обычно генерируются CLR или являются исключениями общей природы и могут быть сгенерированы любым приложением. Например, исключение StackOverflowException генерируется CLR при переполнении стека, исключение ArgumentException (и производные от него) могут быть сгенерированы любым приложением, если метод получает недопустимые значения аргументов.

ApplicationExcption – от этого класса должны наследоваться пользовательские исключения, специфичные для приложения.

Иерархия классов исключений является в некоторой степени необычной, т.к. производные классы в основном не добавляют новую функциональность к возможностям базового класса. Эти классы используются для указания более специфических причин возникновения ошибки. Например, от класса ArgumentException наследуются классы

ArgumentNullException (генерируется при передаче null в качестве параметра метода) и ArgumentOutOfRange (генерируется при выходе переменной за допустимый диапазон значений).

Обработка исключений

Синтаксис блока обработки исключений:

```
try
  {
    // код, в котором может возникнуть исключение
  }
catch(тип исключения)
  {
    // обработка исключения
  }
finally
  {
     // освобождение ресурсов
  }
```

Блок try содержит код, требующий общей очистки ресурсов или восстановления после исключения.

Блок catch содержит код, который должен выполняться при возникновении исключения. При объявлении блока catch указывается тип исключения, для обработки которого он предназначен. Если блок try завершился без генерации исключения, блоки catch не выполняются. Если в блоке try не предполагается возникновение исключения, блок catch может отсутствовать, но тогда обязательно должен быть блок finally.

Блок finally обычно содержит очистку ресурсов, а также другие действия, которые необходимо гарантировано выполнить после завершения блока try и catch. Например, в этом блоке можно выполнить закрытие файла или закрытие соединения с БД. Блок finally выполняется всегда, независимо от возникновения исключения в блоке try. Если нет необходимости выполнять очистку ресурсов, блок finally может отсутствовать, но тогда обязательно должен быть блок catch. Блок try сам по себе не имеет смысла.

Для генерации исключения используется ключевое слово throw. Синтаксис генерации исключения:

```
throw new Конструктор класса исключения()
```

В качестве типа исключения надо использовать производный класс иерархии исключений, который наиболее полно описывает возникшую проблему. Не рекомендуется использовать базовые классы, т.к. в этом случае при обработке исключения трудно будет точно определить причину его возникновения.

При возникновении исключения в блоке try выполнение этого блока прекращается и CLR выполняет поиск блока catch, предназначенного для обработки исключений такого типа. Если этот блок найден, он выполняется, затем выполняется блок finally (если он есть). Если подходящий блок catch не найден, исключение считается необработанным и приводит к аварийному завершению программы.

Пример. Последовательность выполнения кода при возникновении исключения.

```
namespace ExceptionExample1
  {
    class Program
    {
      static void Main(string[] args)
      {
        try
        {
          Console.WriteLine("Before Exception"); //1
          throw new Exception("Test Exception"); //2
     Console.WriteLine("After Exception.This line will never appear");
        }
        catch (Exception e)
        { Console.WriteLine("Exception: {0}", e.Message); //3 }
        finally
        { Console.WriteLine("In finally block"); //4 }
      }
    }
  }
```

Результат выполнения программы изображен на рисунке 2.10.

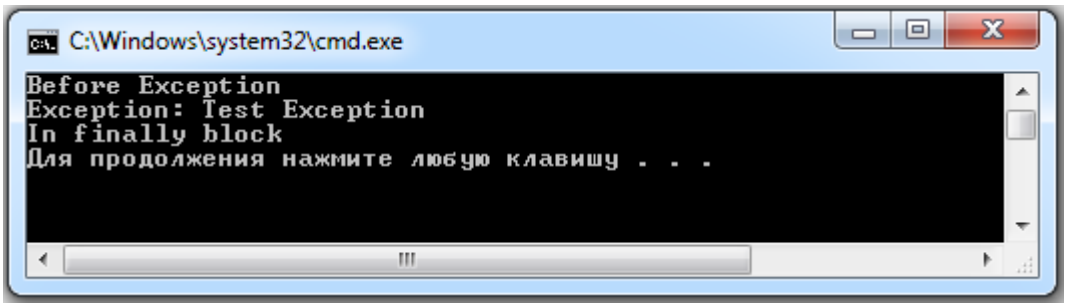

Рис. 2.10 Результат выполнения программы.

Текст «After Exception» не будет выведен, т.к. после генерации исключения выполнение блока try прекращается.

У одного блока try может быть несколько блоков catch для обработки исключений различных типов. При возникновении исключения поиск обработчика начинается с 1-ого блока catch, поэтому следует, сначала размещать обработчики для исключений производных классов, затем базовых.

Для обработки всех типов CLS совместимых исключений можно использовать блок catch, который перехватывает System.Exception, т.к. этот класс является базовым для всех классов исключений:

catch(Exception e)

{ … }

Пример. Использование нескольких блоков catch: вычисляется выражение d = 100/ln(n), n вводится пользователем. Перехватываются следующие типы исключений:

- FormatException – возникает, если введенную пользователем строку невозможно преобразовать в число.
- DivisionByZeroException – возникает, если Ln(n) = 0, т.е. n = 1
- Exception – все остальные исключения, наследуемые от Exception, например Overfloaw.

```
namespace MultipleCatchBlocks
  {
    class Program
    {
      static void Main(string[] args)
      {
        do
        {
          try
          {
```

```
    Console.Write("{0}Input int number: ", Environment.NewLine);
            //чтение ввода пользователя
            string s = Console.ReadLine();
            //условие выхода из цикла
            if (s == string.Empty) return;
            //преобразование строки в число
            int n = Convert.ToInt32(s);
            //проверка, что полученное число принадлежит
            //области определения функции ln()
  if (n <= 0) throw new ArgumentOutOfRangeException("n <= 0");
            double f = Math.Log(n);
            int d = 100 / (int)f;
            Console.WriteLine("d = {0} f = {1}", d, f);
        }
        catch (FormatException)
        {
         //происходит, если введенное пользователем значение
         //невозможно преобразовать в целое число
            Console.WriteLine("FormatException");
        }
        catch (DivideByZeroException)
        {
            //происходит, елси Log(n) = 0 (т.е. n = 1)
            Console.WriteLine("DivideByZeroException");
        }
        catch (Exception e)
        {
 //прехват всех остальных исключений
//например, исключения ArgumentOutOfRangeException,
//которое генерируется, если Log(n) не определен (т.е. n <= 0)
            Console.WriteLine("Exception: {0}", e.Message);
        }
    }
    while (true);
  }
```

```
    }
  }
```

Результат выполнения программы изображен на рисунке 2.11.

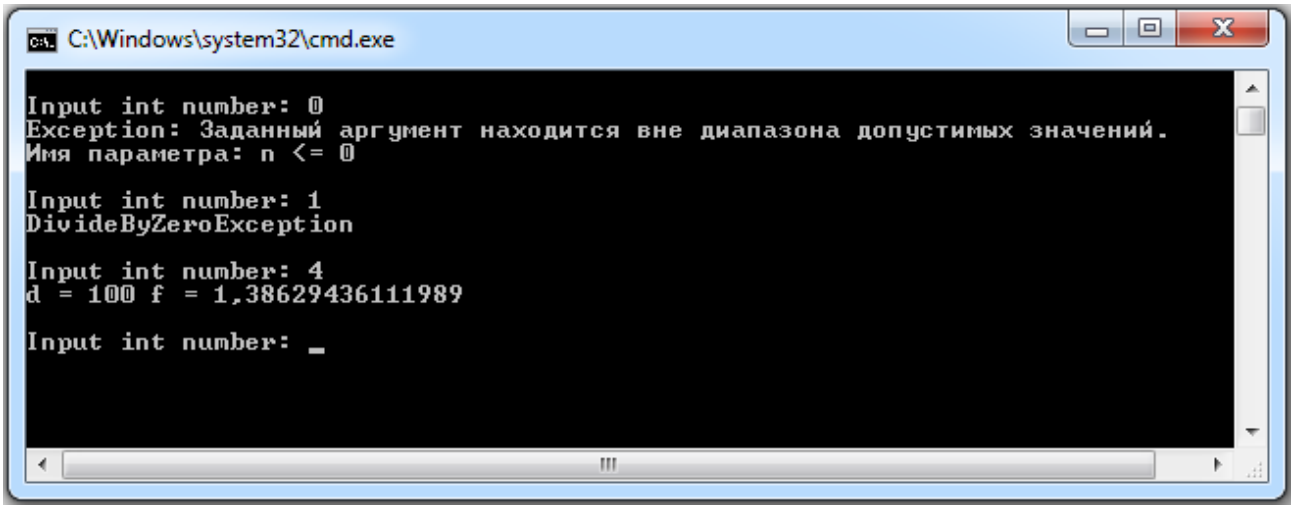

Рис. 2.11 Результат выполнения программы.

Блоки try могут быть вложенными.

Если исключение возникает во вложенном блоке, выполняется блок finally вложенного блока try, затем выполняется поиск подходящего обработчика исключения. Если исключение может быть обработано внутренним блоков catch, оно перехватывается и обрабатывается, после чего продолжается выполнение кода внешнего блока. Если не возникало нового исключения, блок catch внешнего блока игнорируется, блок finally внешнего блока выполняется в любом случае. Если исключение не может быть перехвачено внутренним блоком catch, выполняется проверка внешнего блока catch. Если этот блок не в состоянии обработать исключение, поиск подходящего обработчика выполняется выше по стеку вызовов.

Пример. Вложенные блоки try. Во внутреннем блоке происходит 2 вида исключений:

- деление на 0;
- обращение к массиву по недопустимому индексу.

1-ое исключение перехватывается внутренним блоком catch, 2-ое – внешним.

```
namespace NestedTryBlocks
  {
    class Program
    {
      static void Main()
      {
        int[] a = new int[5];
        int cnt = 0;
```

```
            try //внешний блок try
            {
               for (int i = -3; i <= 3; i++)
               {
                  //при делении на 0 не происходит выход из цикла:
                  //это исключение перехватывается
                  //и обрабатывается вложенным блоком try
                  try //вложенный блок try
                  {
                     a[cnt] = 100 / i;
                     Console.WriteLine(a[cnt]);
                     cnt++;
                  }
                  catch (DivideByZeroException e)
                  {
                     Console.WriteLine("In inner catch");
                     Console.WriteLine(e.Message);
                  }
               }
            }
            catch (IndexOutOfRangeException e)
            {
               Console.WriteLine("In outer catch");
               Console.WriteLine(e.Message);
            }
         }
      }
   }
```

Результат выполнения программы изображен на рисунке 2.12.

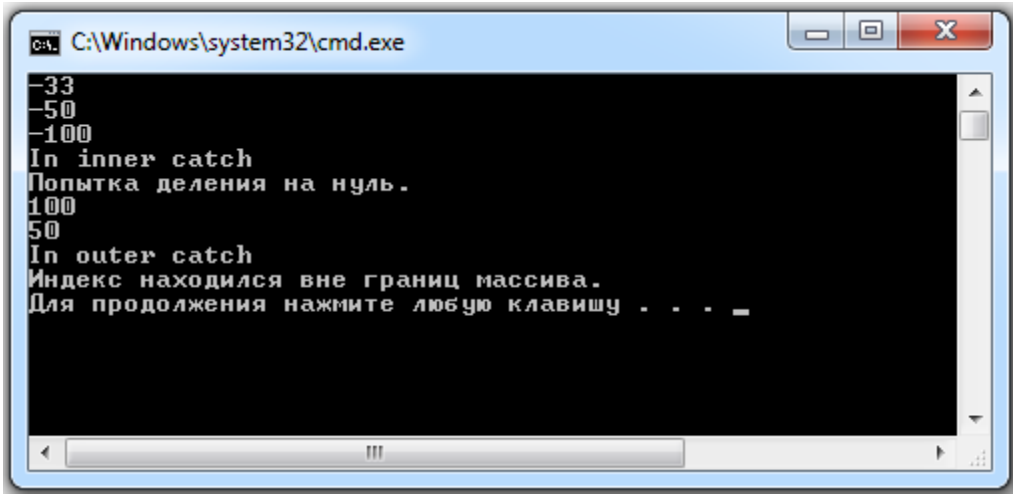

Рис. 2.12 Результат выполнения программы.

Применение конструкций checked и unchecked

При выполнении арифметических операций с целочисленными типами данных может возникать переполнение, т.е. ситуация при которой количество двоичных разрядов полученного результата превышает разрядность переменной, в которую этот результат записывается.

Переполнение может возникнуть в следующих ситуациях:

- в выражениях, которые используют арифметические операторы ++ -- - (унарный) + - * /;
- при выполнении явного преобразования целочисленных типов.

При возникновении переполнения CLR может использовать один из двух вариантов:

- проигнорировать переполнение и отбросить старшие разряды;
- сгенерировать исключение OverflowException.

По умолчанию при возникновении переполнения старшие разряды отбрасываются.

Можно явно задать режим контроля переполнения с помощью ключевых слов checked и unchecked. Ключевое слово checked задает режим контроля переполнения с генерацией исключения. Ключевое слово unchecked задает игнорирование возникновения переполнения.

Пример: Оператор ++ выполняется для переменной b с начальным значением 255:

- в блоке checked – возникает исключение, которое перехватывается в блоке catch;
- в блоке unchecked – исключение не возникает, в переменную b записывается значение 0.

```
namespace CheckOverflowException
  { class Program
    {  static void Main(string[] args)
```

```
{  byte b = 255;
   try
   {  checked
      { b++; //генерация OverflowException thrown }
        Console.WriteLine(b.ToString()); }
   catch (OverflowException e)
   { Console.WriteLine(e.Message);}
   try
   { unchecked
      { b++; //переполнение игнорируется }
      Console.WriteLine(b.ToString()); //0
   }
   catch (OverflowException e)
   { Console.WriteLine(e.Message);}
}
  }
}
```

Результат выполнения программы изображен на рисунке 2.13.

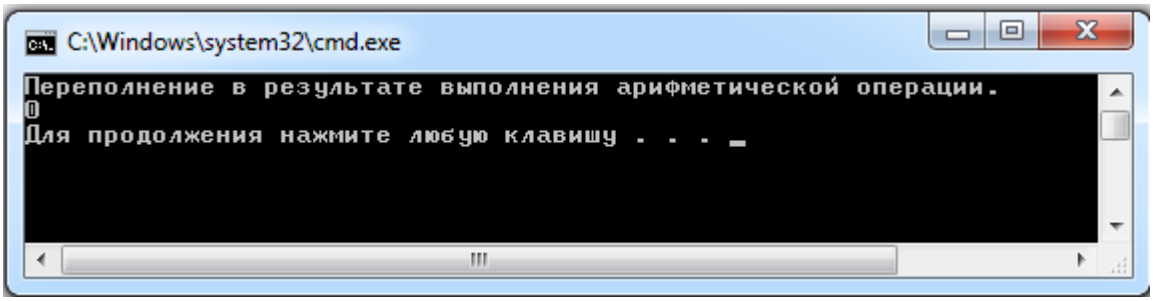

Рис. 2.13 Результат выполнения программы.

3 Контрольные вопросы

1 Что понимается под термином «деструктор»?
2 В чем состоит назначение деструктора?
3 Приведите синтаксис деструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
4 Что такое рекурсия?
5 Что такое статистический метод?
6 Что такое перегрузка операторов?
7 Какие операторы можно перегружать?
8 Что такое индексаторы?
9 Для чего нужны одномерные и многомерные индексаторы?

10 Для чего нужны вложенные типы?
11 Для чего нужна обработка исключений?
12 Какие виды исключений бывают? Приведите примеры использования исключений в программах.
13 Для чего используются конструкции checked и unchecked для обработки исключений?

4 Задание

1. Каждый разрабатываемый класс должен, как правило, содержать следующие элементы: скрытые поля, конструкторы с параметрами и без параметров, методы; свойства, индексаторы; перегруженные операции. Функциональные элементы класса должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. При возникновении ошибок должны выбрасываться исключения.
2. Отладить и протестировать программу.
3. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

Вариант 1.

Описать класс для работы с одномерным массивом целых чисел (вектором). Обеспечить следующие возможности:

- задание произвольных целых границ индексов при создании объекта;
- обращение к отдельному элементу массива с контролем выхода за пределы массива;
- выполнение операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов;
- выполнение операций умножения и деления всех элементов массива на скаляр;
- вывод на экран элемента массива по заданному индексу и всего массива.

Вариант 2.

Описать класс для работы с одномерным массивом строк фиксированной длины. Обеспечить следующие возможности:

- задание произвольных целых границ индексов при создании объекта;
- обращение к отдельной строке массива по индексу с контролем выхода за пределы массива;
- выполнение операций поэлементного сцепления двух массивов с образованием нового массива;
- выполнение операций слияния двух массивов с исключением повторяющихся элементов;
- вывод на экран элемента массива по заданному индексу и всего массива.

Вариант 3.

Описать класс многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Обеспечить следующие возможности:

- вычисление значения многочлена для заданного аргумента;
- операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена;
- получение коэффициента, заданного по индексу;
- вывод на экран описания многочлена.

Вариант 4.

Описать класс, обеспечивающий представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывода на экран подматрицы любого размера и всей матрицы, доступа по индексам к элементу матрицы, сложение и умножение элементов матрицы с заданным числом.

Вариант 5.

Описать класс для работы с восьмеричным числом, хранящимся в виде строки символов. Реализовать конструкторы, свойства, методы и следующие операции:

- операции присваивания, реализующие значимую семантику;
- операции сравнения;
- преобразование в десятичное число;
- форматный вывод;
- доступ к заданной цифре числа по индексу.

Вариант 6.

Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (по автору, по году издания или категории), добавления книг в библиотеку, удаления книг из нее, доступа к книге по номеру, сортировки данных (по автору, по году издания или категории).

Вариант 7.

Описать класс «записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по фамилии и доступа к записи по номеру.

Вариант 8.

Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, имени, дате рождения), добавления и удаления записей, сортировки по разным полям, доступа к записи по номеру.

Вариант 9.

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- сложение, вычитание (как с другой матрицей, так и с числом);
- комбинированные операции присваивания (+=, -=);
- операции сравнения на равенство/неравенство;
- операции вычисления обратной и транспонированной матрицы;
- доступ к элементу по индексам.

Вариант 10.

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- умножение, деление (как на другую матрицу, так и на число);
- комбинированные операции присваивания (*=, /=);
- операцию возведения в степень;
- методы вычисления определителя и нормы;
- доступ к элементу по индексам.

Вариант 11.

Описать класс для работы с двоичным числом, хранящимся в виде строки символов. Реализовать конструкторы, свойства, методы и следующие операции:

- операции присваивания, реализующие значимую семантику;
- операции сравнения;
- преобразование в десятичное число;
- форматный вывод;
- доступ к заданной цифре числа по индексу.

Вариант 12.

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- методы, реализующие проверку типа матрицы (квадратная, диагональная, нулевая, единичная, симметричная, верхняя треугольная, нижняя треугольная);
- операции сравнения на равенство/неравенство;
- доступ к элементу по индексам.

Вариант 13.

Описать класс «множество», позволяющий выполнять основные операции: добавление и удаление элемента, пересечение, объединение и разность множеств.

Вариант 14.

Описать класс «предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых, это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Предусмотреть возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя, сортировки по словам или по номеру страниц.

Вариант 15.

Описать класс «автостоянка» для хранения сведений об автомобилях. Для каждого автомобиля записываются госномер, цвет, фамилия владельца и признак присутствия на стоянке. Обеспечить возможность поиска автомобиля по разным критериям, вывода списка присутствующих и отсутствующих на стоянке автомобилей, доступа к имеющимся сведениям по номеру места, сортировки данных по различным полям.

Вариант 16.

Описать класс «колода карт», включающий закрытый массив элементов класса «карта». В карте хранятся масть и номер. Обеспечить возможность вывода карты по номеру, вывода всех карт, перемешивания колоды и выдачи всех карт из колоды поодиночке и по 6 штук в случайном порядке.

Вариант 17.

Описать класс для работы с четырехричным числом, хранящимся в виде строки символов. Реализовать конструкторы, свойства, методы и следующие операции:

- операции присваивания, реализующие значимую семантику;
- операции сравнения;
- преобразование в десятичное число;
- форматный вывод;
- доступ к заданной цифре числа по индексу.

Вариант 18.

Описать класс «поезд», содержащий следующие закрытые поля:

- название пункта назначения;
- номер поезда (может содержать буквы и цифры);
- время отправления.

Предусмотреть свойства для получения состояния объекта.

Описать класс «вокзал», содержащий закрытый массив поездов. Обеспечить следующие возможности:

вывод информации о поезде по номеру с помощью индекса;

- вывод информации о поездах, отправляющихся после введенного с клавиатуры времени;
- перегруженную операцию сравнения, выполняющую сравнение времени отправления двух поездов;
- вывод информации о поездах, отправляющихся в заданный пункт назначения.

Информация должна быть отсортирована по времени отправления.

Вариант 19.

Описать класс «товар», содержащий следующие закрытые поля:

- название товара;
- название магазина, в котором продается товар;
- стоимость товара в рублях.

Предусмотреть свойства для получения состояния объекта.

Описать класс «склад», содержащий закрытый массив товаров. Обеспечить следующие возможности:

- • вывод информации о товаре по номеру с помощью индекса;
- • вывод на экран информации о товаре, название которого введено с клавиатуры;
- если таких товаров нет, выдать соответствующее сообщение;
- • сортировку товаров по названию магазина, по наименованию и по цене;
- • перегруженную операцию сложения товаров, выполняющую сложение их цен.

Вариант 20.

Описать класс «самолет», содержащий следующие закрытые поля:

- название пункта назначения;
- шестизначный номер рейса;
- время отправления.

Предусмотреть свойства для получения состояния объекта.

Описать класс «аэропорт», содержащий закрытый массив самолетов. Обеспечить следующие возможности:

- вывод информации о самолете по номеру рейса с помощью индекса;
- вывод информации о самолетах, отправляющихся в течение часа после введенного с клавиатуры времени;
- вывод информации о самолетах, отправляющихся в заданный пункт назначения;
- перегруженную операцию сравнения, выполняющую сравнение времени отправления двух самолетов.

Информация должна быть отсортирована по времени отправления.