Керов Леонид Александрович

ДОПОЛНИТЕЛЬНЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C#

Аннотация

Данная статья является шестой из серии статей, посвященных изложению «нулевого уровня» языка C#. Рассматриваются следующие приемы программирования на языке C#: использование механизма наследования, определение и использование виртуальных методов, интерфейсов, абстрактных классов и методов, назначение и объявление пространства имен, перегрузка операторов, определение и использование свойств и индексаторов.

Ключевые слова: C#, наследование, виртуальные методы, интерфейсы, абстрактные классы и методы, пространство имен, перегрузка операторов, свойства, индексаторы.

1. ИСПОЛЬЗОВАНИЕ МЕХАНИЗМА НАСЛЕДОВАНИЯ

1.1. ПОНЯТИЕ И ОПРЕДЕЛЕНИЕ ПРОИЗВОДНОГО КЛАССА

Одним из принципов объектно-ориентированного программирования является *инкапсуляция*, то есть ограничение доступа к членам класса для предотвращения несанкционированного их использования. Следующим важным принципом объектно-ориентированного программирования является *наследование* (inheritance): можно определить новый класс, основанный на существующем классе. При этом исходный класс называется базовым классом, а новый класс – *производным классом*. В качестве примера приведем программу, в которой определен класс `Point1D` для представления точки в одномерном пространстве и определен производный от него класс `Point2D` для представления точки на плоскости (см. листинг 1, рис. 1).

Производный класс может пользоваться членами базового класса, которые объявлены как `protected` или `public`. Члены базового класса, имеющие модификатор доступа `private`, по-прежнему доступны только внутри этого класса. Для классов, которые не являются производными от данного класса, члены базового класса, имеющие модификатор доступа `protected`, имеют такой же статус, как члены с модификатором доступа `private`.

1.2. ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫЗОВА КОНСТРУКТОРОВ

При создании объекта производного класса всегда действует следующее правило: сначала вызывается конструктор базового класса, затем вызывается конструктор

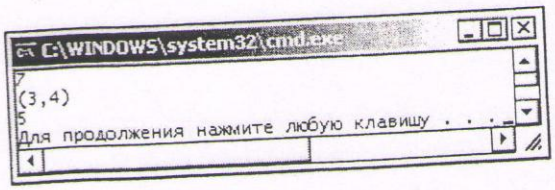

Рис. 1. Результаты работы программы с классами Point1D и Point2D

Листинг 1

```
using System;
class Point1D
{
    protected int x;
    public void setX(int x) { this.x = x; }
    public int Distance1D() { return x; }
}
class Point2D : Point1D
{
    protected int y;
    public void setY(int y) { this.y = y; }
    public int Distance2D()
    { return (int)Math.Sqrt(x * x + y * y); }
    public string getPoint2D()
    { return "(" + x + "," + y + ")"; }
}
class Program
{
    static void Main(string[] args)
    {
        Point1D p1 = new Point1D(); p1.setX(7);
        Console.WriteLine(p1.Distance1D());

        Point2D p2 = new Point2D(); p2.setX(3); p2.setY(4);
        Console.WriteLine(p2.getPoint2D());
        Console.WriteLine(p2.Distance2D());
    }
}
```

Листинг 2

```
using System;
class Point1D
{
    protected int x;
    public Point1D()
    {
        Console.WriteLine("Конструктор Point1D: x={0}", this.x);
    }
}
class Point2D : Point1D
{
    protected int y;
    public Point2D(int x, int y)
    {
        this.x = x; this.y = y;
        Console.WriteLine("Конструктор Point2D: x={0},y={1}",
                          this.x, this.y);
    }
}
class P02
{
    public static void Main()
    {
        Point2D p2 = new Point2D(3, 4);
    }
}
```

производного класса. Проиллюстрируем это правило посредством следующей программы (см. листинг 2, рис. 2).

Если базовый класс не имеет конструктора, тогда сначала вызывается конструктор «по умолчанию» базового класса, затем вызывается конструктор производного класса. Если в базовом классе определен конструктор с параметрами и не определен конструктор без параметров, то в конструкторе производного класса необходимо явно вызывать конструктор базового класса с помощью ключевого слова **base** (см. листинг 3, рис. 3).

Ключевое слово **base** можно использовать для явного вызова любого из конструкторов базового класса. Для конструкторов действуют те же правила доступа, что и для методов, то есть если конструктор базового класса объявлен как **private**, то он будет недоступен из производного класса.

1.3. ОПРЕДЕЛЕНИЕ И ИСПОЛЬЗОВАНИЕ ВИРТУАЛЬНЫХ МЕТОДОВ

Как отмечалось выше, фундаментальными принципами объектно-ориентированного программирования являются инкапсуляция

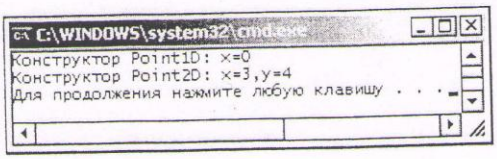

Рис. 2. Последовательность вызовов конструкторов базового и производного классов

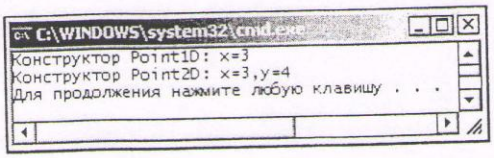

Рис. 3. Результат работы программы с явным вызовом конструктора базового класса

Листинг 3

```
using System;
class Point1D
{
    protected int x;
    public Point1D(int x)
    {
        this.x = x;
        Console.WriteLine("Конструктор Point1D: x={0}", this.x);
    }
}
class Point2D : Point1D
{
    protected int y;
    public Point2D(int x, int y)
        : base(x)
    {
        this.y = y;
        Console.WriteLine("Конструктор Point2D: x={0},y={1}",
                          this.x, this.y);
    }
}
class P03
{
    public static void Main()
    {
        Point2D p2 = new Point2D(3, 4);
    }
}
```

и наследование (возможность определить новый класс, основанный на существующем классе и имеющий доступ к членам базового класса). Третьим фундаментальным принципом объектно-ориентированного программирования является *полиморфизм*, суть которого заключается в следующем. Можно определить несколько функций с одинаковыми именами, но различным поведением. Поведение каждой из указанных функций определяется данными, к которым она применяется.

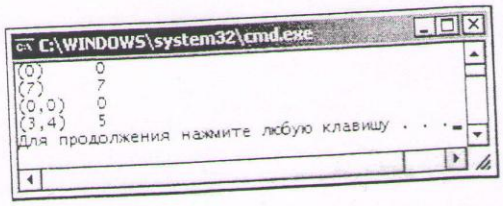

Рис. 4. Пример работы программы с виртуальными методами

Одним из способов реализации полиморфизма является использование виртуальных методов: в базовом классе определяется метод с ключевым словом **virtual**, а в производном классе этот метод переопределяется (перегружается) с ключевым словом **override**. При этом количество и типы параметров перегруженной и исходной функций должны совпадать (см. листинг 4, рис. 4).

Можно перегружать перегруженные методы, причем слово **virtua** указывается только в самом первом базовом классе. Перегруженные методы не могут иметь модификаторов **static** и **private**.

Листинг 4

```
using System;
class Point1D
{
    protected int x;
    public Point1D() { this.x = 0; }
    public Point1D(int x) { this.x = x; }
    public virtual int Distance() { return x; }
    public virtual string getPoint() { return "(" + x + ")"; }
}
class Point2D : Point1D
{
    protected int y;
    public Point2D() { this.y = 0; }
    public Point2D(int x, int y) : base(x) { this.y = y; }
    public override int Distance() { return (int)Math.Sqrt(x * x + y * y);
}
    public override string getPoint() { return "(" + x + "," + y + ")"; }
}
class P04
{
    public static void Main()
    {
        Point1D p11 = new Point1D(); Point1D p12 = new Point1D(7);
        Point2D p21 = new Point2D(); Point2D p22 = new Point2D(3, 4);
        Console.WriteLine("{0}\t{1}", p11.getPoint(), p11.Distance());
        Console.WriteLine("{0}\t{1}", p12.getPoint(), p12.Distance());
        Console.WriteLine("{0}\t{1}", p21.getPoint(), p21.Distance());
        Console.WriteLine("{0}\t{1}", p22.getPoint(), p22.Distance());
    }
}
```

1.4. ИСПОЛЬЗОВАНИЕ «NEW» ДЛЯ СОКРЫТИЯ МЕТОДОВ

В производном классе можно скрыть реализацию метода базового класса и заменить ее своей реализацией, указав ключевое слово **new**. При этом можно скрыть как виртуальный, так и обычный метод (см., например, листинг 5, рис. 5).

Различие между виртуальными и скрываемыми методами заключается в следующем. Если используются виртуальные методы, то их семантика определяется данными, к которым они применяются. Если используются скрываемые методы, то принцип полиморфизма не работает.

2. ОПРЕДЕЛЕНИЕ И ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСОВ

2.1. ПОНЯТИЕ И ОПРЕДЕЛЕНИЕ ИНТЕРФЕЙСА

В некоторых случаях полезно использовать механизм наследования, но не требуется использовать базовый класс для создания объектов. Нужно лишь указать набор методов, которые должен реализовать производный класс. В таком слу-

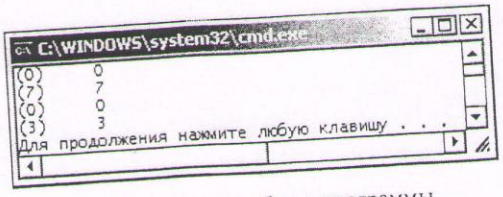

Рис. 5. Пример работы программы, скрывающей реализацию методов базового класса

Листинг 5

```
using System;
class Point1D
{
    protected int x;
    public Point1D() { this.x = 0; }
    public Point1D(int x) { this.x = x; }
    public virtual int Distance() { return x; }
    public string getPoint() { return "(" + x + ")"; }
}
class Point2D : Point1D
{
    protected int y;
    public Point2D() { this.y = 0; }
    public Point2D(int x, int y) : base(x) { this.y = y; }
    public new int Distance() { return (int)Math.Sqrt(x * x + y * y);
}
    public new string getPoint() { return "(" + x + "," + y + ")"; }
}
class P07_05
{
    public static void Main()
    {
        Point1D[] p = new Point1D[4];
        p[0] = new Point1D();
        p[1] = new Point1D(7);
        p[2] = new Point2D();
        p[3] = new Point2D(3, 4);
        for (int i = 0; i < p.Length; i++)
            Console.WriteLine("{0}\t{1}",
            p[i].getPoint(), p[i].Distance());
    }
}
```

чае вместо базового класса объявляется интерфейс:

1) вместо ключевого слова **class** используется слово **interface**;

2) при объявлении интерфейса все его методы неявно имеют модификатор **public**, причем явно указывать его нельзя;

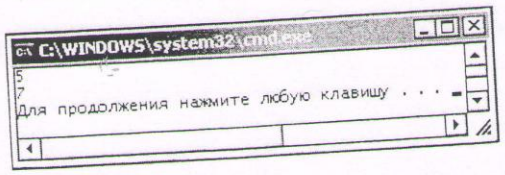

Рис. 6. Результат работы программы, в которой определен и используется интерфейс

3) методы интерфейса не содержат тела с реализацией;

4) интерфейс не может содержать данных, в нем могут быть объявлены (без реализации!) только методы, события, свойства и индексаторы.

Приведем пример программы, в которой определен и используется интерфейс (см. листинг 6, рис. 6).

Можно создавать ссылки на интерфейс и работать через эти ссылки с объектами классов, реализующий данный интерфейс. При этом автоматически поддерживается полиморфизм.

Листинг 6

```
using System;
interface IPoint
{
    int Distance();
}
class Point2D : IPoint
{
    protected int x, y;
    public Point2D() { this.x = 0; this.y = 0; }
    public Point2D(int x, int y) { this.x = x; this.y = y; }
    public int Distance() { return (int)Math.Sqrt(x * x + y * y); }
}
class Point3D : IPoint
{
    protected int x, y, z;
    public Point3D() { this.x = 0; this.y = 0; this.z = 0; }
    public Point3D(int x, int y, int z)
    { this.x = x; this.y = y; this.z = z; }
    public int Distance()
    { return (int)Math.Sqrt(x * x + y * y + z * z); }
}
class P06
{
    public static void Main()
    {
        IPoint[] ip = new IPoint[2];
        ip[0] = new Point2D(3, 4);
        ip[1] = new Point3D(3, 4, 5);
        for (int i = 0; i < ip.Length; i++)
        {
            Console.WriteLine(ip[i].Distance());
        }
    }
}
```

2.2. РЕАЛИЗАЦИЯ НЕСКОЛЬКИХ ИНТЕРФЕЙСОВ В КЛАССЕ

Язык C# не позволяет использовать множественное наследование: производный класс может иметь только один базовый класс. Это ограничение компенсируется тем, что класс может реализовать несколько интерфейсов. В этом случае они перечисляются через запятую в объявлении класса (см. листинг 7, рис. 7).

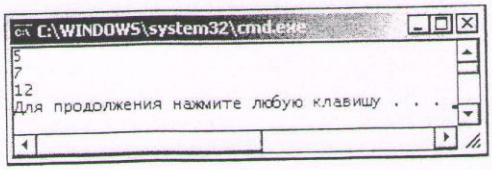

Рис. 7. Пример работы программы, в которой класс реализует несколько интерфейсов

Листинг 7

```
using System;
interface IPoint
{
    int Distance();
}
interface IShape
{
    int Area();
}
class Point2D : IPoint, IShape
{
    protected int x, y;
    public Point2D()
    { this.x = 0; this.y = 0; }
    public Point2D(int x, int y)
    { this.x = x; this.y = y; }
    public int Distance()
    { return (int)Math.Sqrt(x * x + y * y); }
    public int Area()
    { return x * y; }
}
class Point3D : IPoint
{
    protected int x, y, z;
    public Point3D()
    { this.x = 0; this.y = 0; this.z = 0; }
    public Point3D(int x, int y, int z)
    { this.x = x; this.y = y; this.z = z; }
    public int Distance()
    { return (int)Math.Sqrt(x * x + y * y + z * z); }
}
class P07
{
    public static void Main()
    {
        IPoint[] ip = new IPoint[2];
        ip[0] = new Point2D(3, 4); ip[1] = new Point3D(3, 4, 5);
        for (int i = 0; i < ip.Length; i++)
            Console.WriteLine(ip[i].Distance());
        IShape ish = new Point2D(3, 4);
        Console.WriteLine(ish.Area());
    }
}
```

2.3. ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСОВ В ПРОИЗВОДНОМ КЛАССЕ

Если определяется новый класс, то он может быть производным от другого класса и, вместе с тем, может реализовать некоторые интерфейсы. При этом в заголовке определения такого класса сначала указывается имя базового класса, затем перечисляются имена реализуемых интерфейсов (см., например, листинг 8, рис. 8).

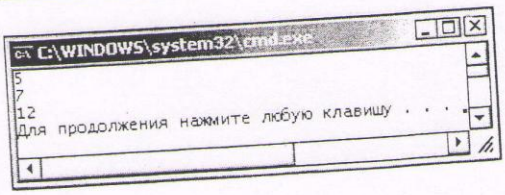

Рис. 8. Пример работы программы с производным классом, реализующим интерфейс

Листинг 8

```
using System;
interface IPoint
{
    int Distance();
}
interface IShape
{
    int Area();
}
class Point2D : IPoint, IShape
{
    protected int x, y;
    public Point2D()
    { this.x = 0; this.y = 0; }
    public Point2D(int x, int y)
    { this.x = x; this.y = y; }
    public int Distance()
    { return (int)Math.Sqrt(x * x + y * y); }
    public int Area()
    { return x * y; }
}
class Point3D : Point2D, IPoint
{
    protected int z;
    public Point3D() : base()
    { this.z = 0; }
    public Point3D(int x, int y, int z) : base(x, y)
    { this.z = z; }
    public new int Distance()
    { return (int)Math.Sqrt(x * x + y * y + z * z); }
}
class P08
{
    public static void Main()
    {
        IPoint[] ip = new IPoint[2];
        ip[0] = new Point2D(3, 4); ip[1] = new Point3D(3, 4, 5);
        for (int i = 0; i < ip.Length; i++)
            Console.WriteLine(ip[i].Distance());
        IShape ish = new Point2D(3, 4);
        Console.WriteLine(ish.Area());
    }
}
```

2.4. ИСПОЛЬЗОВАНИЕ ОДНОИМЕННЫХ МЕТОДОВ В РАЗНЫХ ИНТЕРФЕЙСАХ

В разных интерфейсах разрешается указывать методы с одинаковыми именами. При реализации такого метода требуется соблюдать следующие правила. При определении метода перед его именем указывают имя интерфейса. Нельзя указывать для такого метода модификатор доступа (как и для интерфейса, метод неявно использует модификатор `public`). Нельзя объявлять такой метод как виртуальный. При доступе к методу необходимо вызывать его через ссылку на интерфейс (см. листинг 9, рис. 9).

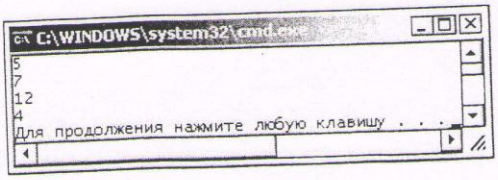

Рис. 9. Пример работы программы, реализующей одноименный метод из двух интерфейсов

Листинг 9

```
using System;
interface IPoint { int Distance();}
interface IShape { int Area(); int Distance();}
class Point2D : IPoint, IShape
{
    protected int x, y;
    public Point2D()
    { this.x = 0; this.y = 0; }
    public Point2D(int x, int y)
    { this.x = x; this.y = y; }
    int IPoint.Distance()
    { return (int)Math.Sqrt(x * x + y * y); }
    int IShape.Distance()
    { return y; }
    public int Area()
    { return x * y; }
}
class Point3D : Point2D, IPoint
{
    protected int z;
    public Point3D() : base()
    { this.z = 0; }
    public Point3D(int x, int y, int z) : base(x, y)
    { this.z = z; }
    public int Distance()
    { return (int)Math.Sqrt(x * x + y * y + z * z); }
}
class P09
{
    public static void Main()
    {
        IPoint[] ip = new IPoint[2];
        ip[0] = new Point2D(3, 4);
        ip[1] = new Point3D(3, 4, 5);
        for (int i = 0; i < ip.Length; i++)
            Console.WriteLine(ip[i].Distance());
        IShape ish = new Point2D(3, 4);
        Console.WriteLine(ish.Area());
        Console.WriteLine(ish.Distance());
    }
}
```

3. ИСПОЛЬЗОВАНИЕ АБСТРАКТНЫХ КЛАССОВ И МЕТОДОВ

По определению, интерфейсы не содержат реализации своих методов. Однако иногда необходимо реализовать часть методов, так как они являются общими для производных классов. Для этой цели ис-

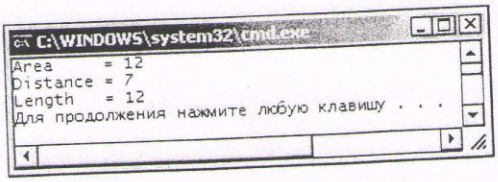

Рис. 10. Пример работы программы с абстрактным классом

Листинг 10

```
using System;
interface IPoint
{
    int Distance();
}
interface IShape
{
    int Area();
}
abstract class Point2D : IPoint, IShape
{
    protected int x, y;
    public Point2D()
    { this.x = 0; this.y = 0; }
    public Point2D(int x, int y)
    { this.x = x; this.y = y; }
    public virtual int Distance()
    { return (int)Math.Sqrt(x * x + y * y); }
    public int Area()
    { return x * y; }
    abstract public int Length();
}
class Point3D : Point2D, IPoint
{
    protected int z;
    public Point3D() : base()
    { this.z = 0; }
    public Point3D(int x, int y, int z) : base(x, y)
    { this.z = z; }
    public override int Distance()
    { return (int)Math.Sqrt(x * x + y * y + z * z); }
    public override int Length()
    { return x + y + z; }
}
class P10
{
    public static void Main()
    {
        Point3D p = new Point3D(3, 4, 5);
        Console.WriteLine("Area     = {0}", p.Area());
        Console.WriteLine("Distance = {0}", p.Distance());
        Console.WriteLine("Length   = {0}", p.Length());
    }
}
```

пользуется абстрактный класс, определяя который следует придерживаться следующих правил. Перед словом **class** указывается ключевое слово **abstract**. Нельзя создавать объекты абстрактного класса (так как не все его методы реализованы). Можно объявлять абстрактные методы, то есть методы без реализации. Абстрактный метод автоматически становится виртуальным, причем указывать ключевое слово **virtual** нельзя. Перегрузка абстрактного метода в классе, производном от абстрактного, ничем не отличается от перегрузки виртуального метода. Абстрактный класс может реализовать один или несколько интерфейсов. Наследуемые от интерфейса методы должны быть реализованы. Можно сказать, что абстрактный класс предоставляет интерфейс для других классов, а также реализацию их общих методов (см. листинг 10, рис. 10).

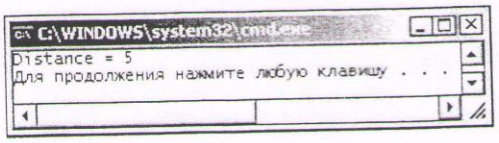

Рис. 11. Пример работы программы, в которой определено пространство имен

4. НАЗНАЧЕНИЕ И ОБЪЯВЛЕНИЕ ПРОСТРАНСТВА ИМЕН

Пространство имен (**namespace**) позволяет разбить классы и интерфейсы на группы, что позволяет в разных пространствах имен иметь одинаковые имена классов и интерфейсов. Для определения пространства имен используется ключевое слово namespace (см. листинг 11, рис. 11).

Можно добавлять новые классы и интерфейсы в любое пространство имен, в том числе и в системное пространство имен, например листинг 12.

Можно определять вложенные пространства имен. Для этого достаточно указать полное имя вложенного пространства имен. В начале файла, перед любыми объявлени-

Листинг 11

```
using System;
namespace Geometry
{
    interface IPoint
    {
        int Distance();
    }
    class Point : IPoint
    {
        protected int x, y;
        public Point()
        { this.x = 0; this.y = 0; }
        public Point(int x, int y)
        { this.x = x; this.y = y; }
        public int Distance()
        { return (int)Math.Sqrt(x * x + y * y); }
    }
}
class P11
{
    public static void Main()
    {
        Geometry.Point p = new Geometry.Point(3, 4);
        Console.WriteLine("Distance = {0}", p.Distance());
    }
}
```

Листинг 12

```
namespace System
{
    interface IPoint
    {
        int Distance();
    }
    class Point : IPoint
    {
        protected int x, y;
        public Point()
        { this.x = 0; this.y = 0; }
        public Point(int x, int y)
        { this.x = x; this.y = y; }
        public int Distance()
        { return (int)Math.Sqrt(x * x + y * y); }
    }
}
class P12
{
    public static void Main()
    {
        System.Point p = new System.Point(3, 4);
        System.Console.WriteLine("Distance = {0}", p.Distance());
    }
}
```

Листинг 13

```
using System;
namespace System.Geometry
{
    interface IPoint
    {
        int Distance();
    }
    class Point : IPoint
    {
        protected int x, y;
        public Point()
        { this.x = 0; this.y = 0; }
        public Point(int x, int y)
        { this.x = x; this.y = y; }
        public int Distance()
        { return (int)Math.Sqrt(x * x + y * y); }
    }
}
class P13
{
    public static void Main()
    {
        System.Geometry.Point p =
            new System.Geometry.Point(3, 4);
        Console.WriteLine("Distance = {0}", p.Distance());
    }
}
```

ями можно указать директиву **using**. Ее использование позволяет не писать имя пространства имен перед именем класса. Директива **using** подключает только указанное в ней пространство имен; вложенные пространства имен нужно подключать с помощью отдельных директив **using**, например листинг 13.

5. ПЕРЕГРУЗКА ОПЕРАТОРОВ

5.1. ПЕРЕГРУЗКА АРИФМЕТИЧЕСКИХ ОПЕРАТОРОВ

Язык C# позволяет перегрузить арифметические операторы для пользовательских типов. При этом оператор должен быть объявлен как **static** и **public**. В качестве параметров оператора принимаются объекты класса, для которого переопределяется данный оператор. Возвращаемым значением бывает, как правило, тип самого класса (см., например, листинг 14, рис. 12).

5.2. ПЕРЕГРУЗКА ОПЕРАТОРОВ СРАВНЕНИЯ

Операции сравнения необходимо перегружать попарно:

- операцию **<** вместе с операцией **>**;
- операцию **<=** вместе с операцией **>=**;
- операцию **==** вместе с операцией **!=**.

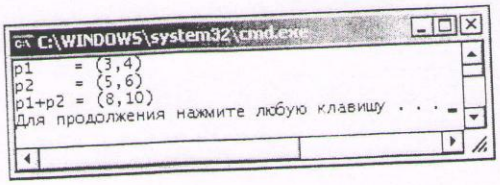

Рис. 12. Пример работы программы с перегруженным оператором сложения

Листинг 14

```
using System;
class Point
{
    private int x, y;
    public Point()
    {
        this.x = 0; this.y = 0;
    }
    public Point(int x, int y)
    {
        this.x = x; this.y = y;
    }
    public override string ToString()
    {
        return "(" + x + "," + y + ")";
    }
    public static Point operator +(Point p1, Point p2)
    {
        return new Point(p1.x + p2.x, p1.y + p2.y);
    }
}
class P15
{
    static void Main(string[] args)
    {
        Point p1 = new Point(3, 4);
        Point p2 = new Point(5, 6);
        Console.WriteLine("p1    = {0}", p1);
        Console.WriteLine("p2    = {0}", p2);
        Console.WriteLine("p1+p2 = {0}", p1 + p2);
    }
}
```

Если перегружается операция ==, то рекомендуется перегрузить и метод **Equals()**, чтобы можно было использовать любой из способов сравнения объектов. Если перегружается метод **Equals()**, то рекомендуется перегрузить и метод **GetHashCode** (см. листинг 15, рис. 13).

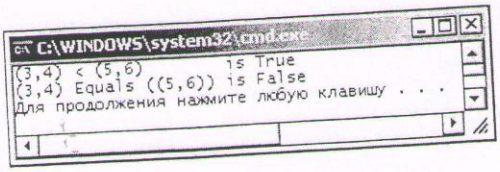

Рис. 13. Пример работы программы с перегруженными операторами сравнения

5.3. ПЕРЕГРУЗКА ОПЕРАТОРОВ ПРЕОБРАЗОВАНИЯ ТИПА

Операции преобразования типа бывают явные и неявные. Операция явного преобразования типа определяется с помощью ключевого слова **explicit**. Операция неявного преобразования типа определяется

Листинг 15

```
using System;
class Point
{
    private int x, y;
    public Point() { this.x = 0; this.y = 0; }
    public Point(int x, int y) { this.x = x; this.y = y; }
    public override string ToString()
    { return "(" + x + "," + y + ")"; }
    public static bool operator >(Point p1, Point p2)
    { return p1.Compare(p2) > 0; }
    public static bool operator <(Point p1, Point p2)
    { return p1.Compare(p2) < 0; }
    public static bool operator ==(Point p1, Point p2)
    { return p1.Compare(p2) == 0; }
    public static bool operator !=(Point p1, Point p2)
    { return p1.Compare(p2) != 0; }
    public override bool Equals(object ob)
    { return ob is Point && Compare((Point)ob) == 0; }
    public override int GetHashCode()
    { return this.x * this.x + this.y * this.y; }
    private int Compare(Point p2)
    {
        double d1 = Math.Sqrt(this.x * this.x + this.y * this.y);
        double d2 = Math.Sqrt(p2.x * p2.x + p2.y * p2.y);
        if (d1 > d2) return 1;
        else if (d1 < d2) return -1; else return 0;
    }
}
class P16
{
    public static void Main()
    {
        Point p1 = new Point(3, 4), p2 = new Point(5, 6);
        Console.WriteLine("{0} < {1}        is {2}",
            p1, p2, p1 < p2);
        Console.WriteLine("{0} Equals ({1}) is {2}",
            p1, p2, p1.Equals(p2));
    }
}
```

с помощью ключевого слова **implicit** (см. листинг 16, рис. 14).

6. ОПРЕДЕЛЕНИЕ И ИСПОЛЬЗОВАНИЕ СВОЙСТВ

Наряду с методами и конструкторами, в классе может быть определена еще одна разновидность функций, которые называются свойствами (**properties**) и предназначены для доступа к закрытым полям класса. Свойство представляет собой функцию, не имеющую параметров.

В заголовке такой функции указывается модификатор доступа, тип возвращаемого значения и имя. Внутри тела функции могут содержаться функции **get** и **set**, которые называются аксессорами (accessors), не имеют параметров и указаний на типы возвращаемых значений. Внутри аксессора **get** содержится оператор **return**, который возвращает значение свойства (тип возвращаемого значения указывается в заголовке свойства). Свойство может возвращать не только значение поля, но и результат вычисления некоторого выражения (то есть может являться виртуальным полем). Внутри аксессора **set** содержится специальная переменная **value**, которая создается компилятором автоматически и содержит значение, присваиваемое свойству.

Свойство может содержать только **get**-аксессор, тогда оно является неизменяемым (**read-only**), или только **set**-аксессор, тогда оно является недоступным. Основным отличием свойств от полей класса является

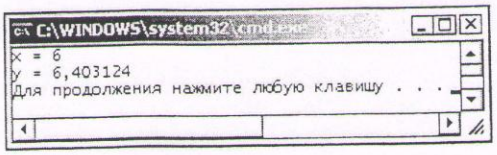

Рис. 14. Пример работы программы с перегруженными операторами преобразования типа

Листинг 16

```
using System;
class Point
{
    private int x, y;
    public Point() { this.x = 0; this.y = 0; }
    public Point(int x, int y) { this.x = x; this.y = y; }
    public override string ToString()
    { return "(" + x + "," + y + ")"; }
    public static explicit operator int(Point p)
    {
        return (int)Math.Sqrt(p.x * p.x + p.y * p.y);
    }
    public static implicit operator float(Point p)
    {
        return (float)Math.Sqrt(p.x * p.x + p.y * p.y);
    }
}
class P17
{
    public static void Main()
    {
        Point p = new Point(4, 5);
        int x = (int)p;
        float y = p;
        Console.WriteLine("x = {0}", x);
        Console.WriteLine("y = {0}", y);
    }
}
```

то, что их нельзя передавать в методы как параметры **ref** или **out**. В качестве примера приведем программу, в которой определено и используется свойство для описания возраста служащего, причем требуется, чтобы служащий был старше 18 лет (см. листинг 17, рис. 15).

Свойства, как и методы, могут быть определены в интерфейсе (см., например, листинг 18).

7. ОПРЕДЕЛЕНИЕ И ИСПОЛЬЗОВАНИЕ ИНДЕКСАТОРОВ

Наряду с методами, конструкторами и свойствами, в классе может быть определена еще одна разновидность функций, которые называются индексаторами и предназначены для доступа к полям класса с использованием индекса (подобно массивам). В заголовке такой функции указывается модификатор доступа, тип возвращаемого значения, ключевое слово **this** и квадратные скобки, содержащие тип и имя индекса. Внутри тела индексатора могут содержаться две функции **get** и **set**, которые называются аксессорами (**accessors**). Они не имеют параметров и указаний на типы возвращаемых значений. Внутри аксессора **get** содержится оператор **return**, который возвращает значение индексатора (тип возвращаемого значения указывается в заголовке индексатора). Внутри аксессора **set** содержится специальная переменная **value**, которая создается компилятором автомати-

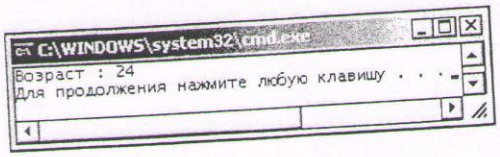

Рис. 15. Пример работы программы, в которой определено и используется свойство

Листинг 17

```
using System;
class Employee
{
    private int age;
    private string name;
    private int id;
    public int Age
    {
        get
        {
            return age;
        }
        set
        {
            if (value > 18) this.age = value;
        }
    }
}
class P18
{
    static void Main(string[] args)
    {
        Employee e = new Employee();
        e.Age = 23; e.Age++;
        Console.WriteLine("Возраст : {0}", e.Age);
    }
}
```

чески и содержит значение, присваиваемое индексатору (см., например, листинг 19, рис. 16).

Как и свойства, индексаторы нельзя передавать в методы как параметры **ref** или **out**. В отличие от свойств, индексаторы не могут быть статическими. Индексатор может не использовать базовый массив. Его назначение может заключаться в том, чтобы обеспечивать такое функционирование, которое для пользователя выглядело бы как использование массива (см., например, листинг 20, рис. 17).

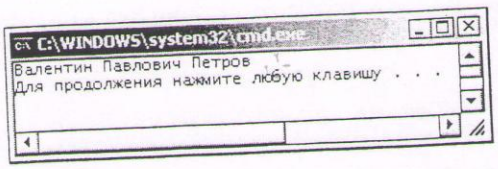

Рис. 16. Пример работы программы, в которой определен и используется индексатор

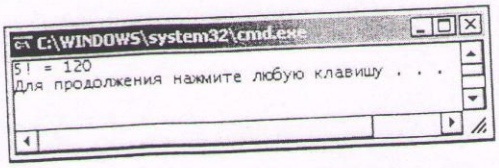

Рис. 17. Пример работы программы с индексатором без базового массива

Листинг 18

```
using System;
interface IEmployee
{
    int Age
    {
        get;
        set;
    }
}
class Employee : IEmployee
{
    private int age;
    private string name;
    private int id;
    public int Age
    {
        get
        {
            return age;
        }
        set
        {
            if (value > 18) this.age = value;
        }
    }
}
class P19
{
    public static void Main()
    {
        Employee e = new Employee();
        e.Age = 23; e.Age++;
        Console.WriteLine("Возраст : {0}", e.Age);
    }
}
```

Листинг 19

```
using System;
class FullName
{
    private string[] name;
    public FullName(int i)
    {
        name = new string[i];
    }
    public string this[int i]
    {
        get
        {
            return name[i];
        }
        set
        {
            name[i] = value;
        }
    }
}
class P20
{
    public static void Main()
    {
        FullName n = new FullName(3);
        n[0] = "Валентин"; n[1] = "Павлович"; n[2] = "Петров";
        Console.WriteLine("{0} {1} {2}", n[0], n[1], n[2]);
    }
}
```

Листинг 20

```
using System;
class Fact
{
    public int this[int i]
    {
        get
        {
            if (i < 0)
                return -1;
            else
            {
                int f = 1;
                for (int j = 1; j <= i; j++)
                    f *= j;
                return f;
            }
        }
    }
}
class P21
{
    public static void Main()
    {
        Fact g = new Fact();
        Console.WriteLine("{0}! = {1}", 5, g[5]);
    }
}
```

Литература

1. *Керов Л.А.* Методы объектно-ориентированного программирования на C# 2005: Учебное пособие. СПб: Издательство «ЮТАС», 2007. 164 с.

2. *Нэш Т.* C# 2008: ускоренный курс для профессионалов: Пер. с англ. М.: ООО «И.Д. Вильямс», 2008. 576 с.

3. *Павловская Т.А.* C#. Программирование на языке высокого уровня. Учебник для вузов. СПб: Питер, 2007. 432 с.

Abstract

The article is sixth of a series of articles, devoted to «a zero level» of language C#. Use of the inheritance mechanism, definition and use of virtual methods, interfaces, abstract classes and methods, purpose and the announcement of namespaces, an overloading of operators, declarations and use of properties and indexers are considered.

Керов Леонид Александрович,
кандидат технических наук,
старший научный сотрудник,
доцент, заведующий кафедрой
бизнес-информатики СПб филиала
ГУ–ВШЭ при Правительстве РФ,

kerov@hse.spb.ru