Керов Леонид Александрович

МЕТОДЫ, КЛАССЫ, ССЫЛКИ И КОНСТРУКТОРЫ В ЯЗЫКЕ C#

Аннотация

Данная статья является пятой из серии статей, посвященных изложению «нулевого уровня» языка C#. Рассматриваются методы, классы, ссылки и конструкторы в языке C#.

Ключевые слова: методы, классы, ссылки, конструкторы, C#.

1. МЕТОДЫ

1.1. ПОНЯТИЕ МЕТОДА. ПЕРЕДАЧА ПАРАМЕТРОВ ПО ЗНАЧЕНИЮ

Метод – это функция, которая определяется внутри класса для выполнения каких-либо действий над данными. Для демонстрации использования метода определим внутри класса `P01` метод `f`, который вычисляет значение функции «факториал» (см. листинг 1, рис. 1).

В определении метода `f` использованы следующие обозначения:

- `public` – модификатор доступа, который означает, что к данной функции можно обращаться как внутри класса, так и из других классов;
- `static` – функция определяется как статическая, то есть обращаться к ней можно без создания объекта данного класса;
- `int` – тип результата функции;
- `f` – имя функции;
- `(int x)` – список параметров функции;
- `{int m=1; for (;x>0;x—)m*=x; return (m);}` – «тело» функции, то есть ее реализация;
- `return m;` – оператор формирования значения функции и прекращения ее работы.

© Л.А. Керов, 2009

Параметр `n` в рассматриваемую функцию передается по значению. Это означает, что внутри тела функции создается локальная переменная `int x`, которой можно пользоваться точно так же, как локальной переменной `int m`. Определение *параметра, передаваемого* методу *по значению*, похоже на определение переменной, то есть в заголовке функции после имени функции в круглых скобках указывается тип параметра и его имя. При завершении работы функции происходит выход из ее тела, поэтому результаты действий над параметром, переданным по значению, теряются.

1.2. ПЕРЕДАЧА ПАРАМЕТРОВ ПО ССЫЛКЕ

Под *ссылкой (reference)* понимают не само значение переменной, а адрес ячейки, содержащей значение переменной. Если параметр нужно передавать по ссылке, то в определении функции и в вызове функ-

Рис. 1. Пример вызова метода с параметром, переданным по значению

Листинг 1

```
using System;
class P01
{
    static void Main(string[] args)
    {
        Console.Write("n  = ");
        int n = int.Parse(Console.ReadLine());
        Console.WriteLine("{0}! = {1}\nn  = {2}", n, f(n), n);
    }
    public static int f(int x) //определение метода "f"
    {
        int m = 1;
        for (; x > 0; x—)
            m *= x;
        return m;
    }
}
```

ции первым указывается слово **ref** (см. листинг 2).

Параметр **n** в рассматриваемую функцию передается по ссылке. Это означает, что внутри тела функции используется переменная **int n**, которая определена в теле функции **Main**. При завершении работы функции происходит выход из ее тела, при этом результаты действий над переданным по ссылке параметром сохраняются (см. рис. 2).

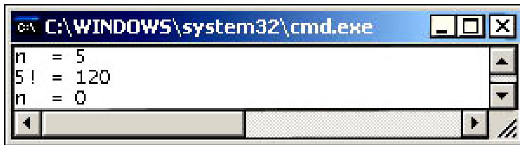


Рис. 2. Пример вызова метода с параметром, переданным по ссылке

1.3. ВОЗВРАЩАЕМЫЕ ПАРАМЕТРЫ

Функция может возвращать только одно значение. Если требуется, чтобы в результате выполнения функции было получено несколько значений, то используются воз-

Листинг 2

```
using System;
public class P02
{
    public static void Main()
    {
        Console.Write("n  = ");
        int n = int.Parse(Console.ReadLine());
        Console.WriteLine("{0}! = {1}\nn  = {2}", n, f(ref n), n);
    }
    public static int f(ref int x)
    {
        int m = 1;
        for (; x > 0; x—)
            m *= x;
        return m;
    }
}
```

вращаемые параметры. В качестве примера приведем программу метода, который вычисляет произведение первых **x** целых чисел, а также вычисляет и возвращает через второй параметр сумму этих чисел (см. листинг 3, рис. 3).

Возвращаемые параметры похожи на параметры, передаваемые по ссылке, однако имеются следующие отличия:

– вместо ключевого слова **ref** используется ключевое слово **out**;

– возвращаемые параметры не могут передавать значения в метод;

– возвращаемые параметры не могут использоваться самим методом иначе, кроме как для возврата значения из метода.

1.4. ПЕРЕГРУЗКА МЕТОДОВ

Методы можно перегружать, то есть в одном классе можно определить несколько методов, которые имеют одно и то же имя, но при этом различаются типом или числом параметров. В качестве примера приведем три определения метода **h**, в первом из которых метод имеет пустой список параметров, во втором – один параметр, в третьем – два параметра (см. листинг 4, рис. 4).

2. КЛАССЫ

2.1. ОПРЕДЕЛЕНИЕ КЛАССА. КЛЮЧЕВОЕ СЛОВО «THIS»

Определение класса вводит в программу новый тип, имя которого совпадает с

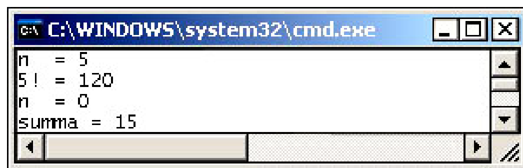


Рис. 3. Пример вызова метода с возвращаемым параметром

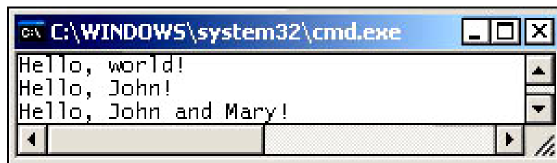


Рис. 4. Примеры вызовов перегруженного метода

Листинг 3

```
using System;
public class P03
{
    public static void Main()
    {
        Console.Write("n  = ");
        int n = int.Parse(Console.ReadLine());
        int p;
        Console.WriteLine("{0}! = {1}\nn  = {2}\nsumma = {3}",
            n, f(ref n, out p), n, p);
    }

    public static int f(ref int x, out int g)
    {
        int m = 1, s = 0;
        for (; x > 0; x–)
        {
            m *= x;
            s += x;
        }
        g = s;
        return m;
    }
}
```

Листинг 4

```
using System;
public class P04
{
    public static void Main()
    {
        h();
        h("John");
        h("John", "Mary");
    }
    public static void h()
    {
        Console.WriteLine("Hello, world!");
    }
    public static void h(string name)
    {
        Console.WriteLine("Hello, {0}!", name);
    }
    public static void h(string name1, string name2)
    {
        Console.WriteLine("Hello, {0} and {1}!", name1, name2);
    }
}
```

именем класса. Тип, который определен посредством класса, можно использовать для объявления переменных, которые называются экземплярами данного типа (или объектами данного класса). В качестве примера рассмотрим программу, в которой определяется тип **Point**, соответствующий точке на экране монитора (см. листинг 5, рис. 5, 6).

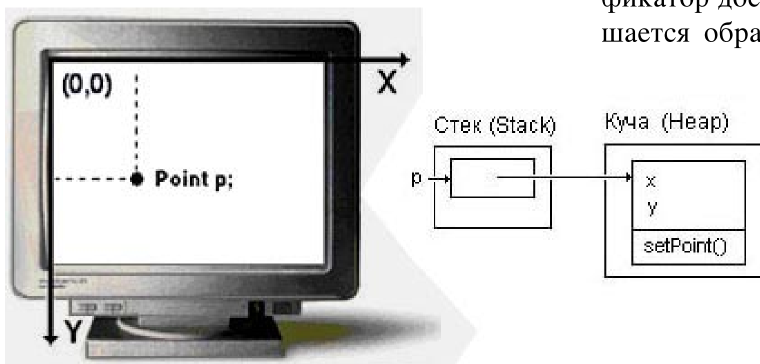


Рис. 5. Тип Point соответствует точке на экране монитора

Рис. 6. Примеры попыток создания объектов класса **Point**

В классе можно определять данные, которые называются *полями* класса (например **x, y**), и функции, которые называются методами класса (например **setPoint()**). Поля и функции класса называются его *членами (members)* и имеют модификаторы доступа; *в частности:*

- **private** – если указан этот модификатор доступа, то к члену класса разрешается обращаться только внутри класса (этот модификатор доступа используется по умолчанию);
- **public** – если указан этот модификатор доступа, то к члену класса разрешается обращаться как внутри класса, так и извне класса.

Ограничение доступа к членам класса – это один из принципов объектно-ориентированного программирования, который называется *инкапсуляцией*. Для обращения к члену объекта класса нужно указать:

Листинг 5

```
using System;
class Point //определение типа "Point"
{
    private int x;
    private int y;
    public void setPoint(int a, int b)
    {
        if (a >= 0 & a < 800 & b >= 0 & b < 600)
        {
            this.x = a; this.y = b;
            Console.WriteLine("Объект создан: " +
            "x={0}\t y={1}", x, y);
        }
        else
            Console.WriteLine("Объект создать нельзя: " +
            "точка ({0},{1}) вне экрана", a, b);
    }
}
class P05
{
    public static void Main()
    {
        Point p = new Point();
        p.setPoint(100, 50);
        p.setPoint(-100, 50);
    }
}
```

имя объекта, точку, имя члена класса; например

```
p.setPoint(100,50);
```

Внутри определения класса можно использовать ключевое слово **this**. Оно обозначает ссылку на тот объект того класса, к члену которого указано обращение (например **this.x**).

2.2. СТАТИЧЕСКИЕ ПОЛЯ И СТАТИЧЕСКИЕ МЕТОДЫ КЛАССА

Если поле класса должно использоваться совместно всеми объектами этого класса, то оно объявляется с модификатором **static**. Метод для работы со статическим полем также должен быть объявлен с модификатором **static** (см. листинг 6, рис. 7).

Память для статического поля выделяется в отдельной области, предназначенной для класса. Внутри класса к статическому полю обращаются просто по имени (перед ним нельзя указывать ключевое слово **this**).

Рис. 7. Примеры использования статического поля и статического метода класса

При вызове статического метода перед ним указывают (через точку) имя класса, а при вызове нестатического метода – имя объекта. Статический метод не может обращаться к нестатическим методам или полям своего класса.

3. ССЫЛКИ И ОБЪЕКТЫ

3.1. ОБЪЯВЛЕНИЕ ССЫЛОК И СОЗДАНИЕ ОБЪЕКТОВ

Все переменные в C# можно разделить на два вида:

– *переменные-значения* – это переменные простых типов (**int**, **char** и т. п.),

структуры и перечисления; значениями таких переменных являются непосредственно данные, которые хранятся в стеке;

– *переменные-ссылки* – их значениями являются адреса ячеек памяти, где хранятся собственно данные; адреса ячеек хранятся в стеке, а сами данные хранятся в куче.

Переменная, тип которой определен с помощью класса, – это переменная-ссылка: объявление переменной-ссылки выделяет память в стеке только под ссылку; для выделения памяти в куче для объекта осуществляет оператор **new**. Переменной-ссылке (например **p2**) можно присвоить другую переменную-ссылку (например **p1**; см. листинг 7, рис. 8).

Объект, созданный перед этим для **p2** с помощью оператора **new**, объявляется «мусором» и будет автоматически уничтожен программой *сборки мусора (garbage collection).*

3.2. КЛАСС SYSTEM.OBJECT

Независимо от того, определяете ли вы в программе собственный класс или используете библиотечный, все они явно или неявно порождаются от класса **System.Object**. Как и встроенные классы, он имеет псевдоним: **object**. Класс **System.Object** содержит в себе определение нескольких методов, которые, по принципу наследования, можно использовать во всех классах; в частности:

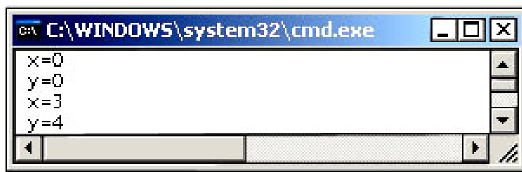
- **public virtual string ToString()** – возвращает представление объекта в виде строки (по умолчанию возвращает имя класса);
- **public static Equals(object obj)** – в качестве параметра принимает ссылку на другой объект и, если значения ссылок совпадают, возвращает **true**; в противном случае – **false**;
- **public Type GetType()** – возвращает объект типа Type, который содержит информацию о типе объекта, а также обо всех его элементах.

Листинг 6

```
using System;
class StaticPoint
{
    private static int x;  //статическое поле
    private int y;
    public static void setX(int a) //статический метод
    {
        if (a >= 0 & a < 800) { x = a; }
    }
    public void setY(int b)
    {
        if (b >= 0 & b < 600) this.y = b;
    }
    public void writePoint()
    { Console.WriteLine("(" + x + "," + this.y + ")"); }
}
class P06
{
    public static void Main()
    {
        StaticPoint.setX(100);
        StaticPoint sp1 = new StaticPoint();
        sp1.setY(50); sp1.writePoint();
        StaticPoint sp2 = new StaticPoint();
        sp2.setY(70); sp2.writePoint();
        sp2.setY(-30); sp2.writePoint();
    }
}
```

Листинг 7

```
using System;
class P07
{
    class Point
    {
        public int x;
        public int y;
    }
    public static void Main()
    {
        Point p1;           //определение ссылки
        p1 = new Point(); //создание объекта
        Console.WriteLine(" x={0}\n y={1}", p1.x, p1.y);
       Point p2 = new Point();//определение ссылки и создание объекта
        p2 = p1;
        p2.x = 3;
        p2.y = 4;
        Console.WriteLine(" x={0}\n y={1}", p1.x, p1.y);
    }
}
```



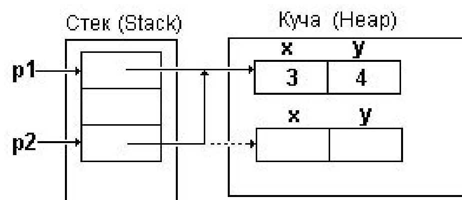


Рис. 8. Примеры объявления и использования ссылочных переменных

Приведем пример программы, в которой используются указанные выше методы (см. листинг 8, рис. 9).

3.3. ПРЕОБРАЗОВАНИЕ К БАЗОВОМУ КЛАССУ

Язык C# разрешает ссылке на объект базового класса присвоить ссылку на объект производного класса. Так как все классы явно или неявно порождаются от класса **`System.Object`**, то ссылке на тип **`object`** можно присвоить ссылку на любой объект.

Допускается и обратное присваивание, но для этого нужно явно преобразовать ссылку на объект типа **`object`** к типу объекта, указанного в левой части оператора присваивания. Проверка выполнимости такого преобразования происходит на этапе выполнения; в случае невозможности выполнить преобразование генерируется исключение **`InvalidCastException`** (см. листинг 9, рис. 10).

3.4. ОПЕРАТОР is

Вместо перехвата исключения **`InvalidCastException`** можно использовать оператор **`is`** для проверки типа. Этот оператор обычно используют, когда необходимо привести объект к одному из ти-

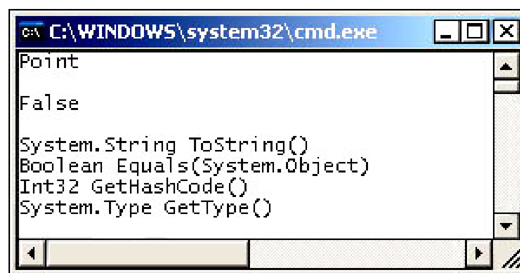


Рис. 9. Примеры использования методов класса **`System.Object`**

Листинг 8

```
using System;
using System.Reflection;
class Point
{
    public int x;
    public int y;
}
class P08
{
    public static void Main()
    {
        Point p1 = new Point();
        Console.WriteLine(p1.ToString());
        Console.WriteLine();

        Point p2 = new Point();
        Console.WriteLine(p1.Equals(p2));
        Console.WriteLine();

        Type t = p1.GetType();
        MethodInfo[] m = t.GetMethods();
        foreach (MethodInfo mi in m)
            Console.WriteLine(mi);
        Console.WriteLine();
    }
}
```

Листинг 9

```
using System;
class Point
{
    public int x;
    public int y;
}
class P09
{
    public static void Main()
    {
        Point p1 = new Point();
        object ob1 = p1; //преобразование к базовому классу
        Console.WriteLine("{0}\n",ob1.ToString());
        object ob2 = new object();
        try
        {
            Point p2 = (Point)ob1; //преобразование к классу "Point"
             Console.WriteLine("{0}\n",p2.ToString());
             Point p3 = (Point)ob2; //Строка 19:некорректное преобр-е
             Console.WriteLine("{0}\n",p3.ToString());
        }
        catch (InvalidCastException e)
        {
            Console.WriteLine("{0}\n",e);
        }
    }
}
```

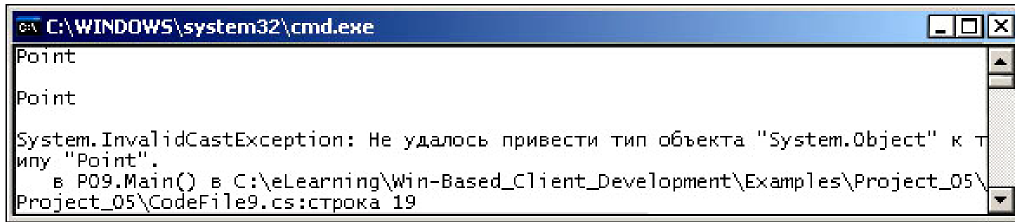


Рис. 10. Примеры попыток преобразования ссылок

пов, но заранее неизвестно к какому типу (см. листинг 10, рис. 11).

3.5. ОПЕРАТОР `as`

Рассмотренный выше оператор **`is`** только проверял возможность такого преобразования. Оператор **`as`** делает попытку выполнить преобразование типа. Если этого сделать нельзя, то результатом его выполнения будет пустая ссылка **`null`** (см. листнг 11, рис. 12).

3.6. УПАКОВКА И РАСПАКОВКА

В **`.NET Framework`** все простые типы данных определены с помощью структур.

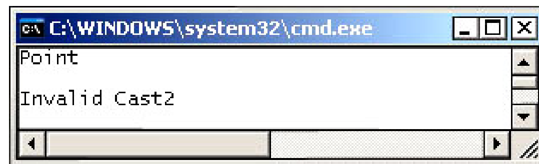


Рис. 11. Примеры использования оператора **`is`**

Листинг 10

```
using System;
class Point
{
    public int x;
    public int y;
}
class P10
{
    public static void Main()
    {
        Point p1 = new Point();
        object ob1 = p1;
        object ob2 = new object();
        if (ob1 is Point)  //использование оператора "is"
        {
            p1 = (Point)ob1;
            Console.WriteLine("{0}\n", p1.ToString());
        }
        else
            Console.WriteLine("{0}\n", "Invalid Cast1");
        if (ob2 is Point) //использование оператора "is"
        {
            Point p2 = (Point)ob2;
            Console.WriteLine("{0}\n", p2.ToString());
        }
        else
            Console.WriteLine("{0}\n", "Invalid Cast2");
    }
}
```

Листинг 11

```
using System;
class Point
{
    public int x;
    public int y;
}

class P11
{
    public static void Main()
    {
        object ob = new object();
        Point p = ob as Point; //использование оператора «as»
        if (p != null)
        {
            Console.WriteLine("{0}\n", p.ToString());
        }
        else
            Console.WriteLine("{0}\n", "Invalid Cast");
    }
}
```

Рис. 12. Примеры использования оператора **as**

Все структуры неявно получаются из класса **System.ValueType**, который является наследником базового **System.Object** (см. рис. 13).

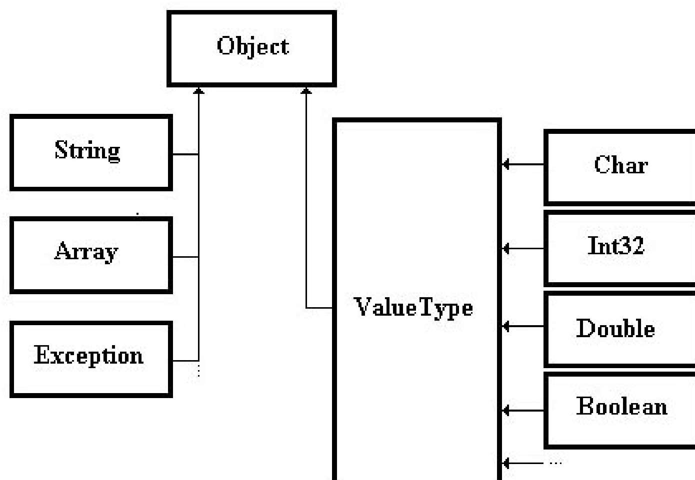


Рис. 13. Иерархия наследования типов в .NET Framework

В .NET имеются две категории типов (типы значений и типы ссылок), поэтому может понадобиться представление переменной одной категории в виде переменной другой категории. Если ссылке на объект типа **object** присвоить переменную встроенного типа, то программа автоматически создаст в куче *ячейку (box)*, куда поместит значение простой переменной, а ссылке на объект типа **object** присвоит адрес этой ячейки; например:

```
object ob1 = 5;
```

Эта процедура называется *упаковкой (boxing)*. При обратном преобразовании выполняется *распаковка (unboxing)*: значение из бокса копируется в переменную; например

```
int x = (int)ob1;
```

Заметим, что распаковку нужно выполнять в соответствующий тип данных. Если это нарушается, генерируется исключение **InvalidCastExeption** (см. листинг 12, рис. 14).

4. КОНСТРУКТОРЫ

4.1 НАЗНАЧЕНИЕ КОНСТРУКТОРА

Конструктор (constructor) – это определенная в классе функция, которая предназначена для инициализации полей класса и которая имеет следующие отличительные признаки:

– имеет имя, совпадающее с именем класса;

– не содержит типа возвращаемого значения.

Конструктор класса вызывается при создании объекта этого класса сразу же после выделения в куче памяти для объекта (см. листинг 13, рис. 15).

4.2. КОНСТРУКТОР ПО УМОЛЧАНИЮ

Если в классе не определен конструктор, то используется конструктор «по умолчанию», который не имеет параметров и инициализирует поля класса следующим образом (см. листинг 14, рис. 16).

– числовые элементы инициализируются нулями;

– булевские элементы инициализируются **false**;

– ссылочные элементы инициализируются пустой ссылкой **null**.

Если в классе определен хотя бы один конструктор, то конструктором «по умолчанию» пользоваться нельзя (см. листинг 15, рис. 17).

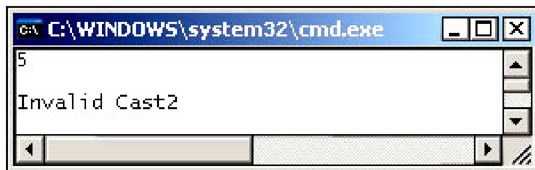


Рис. 14. Примеры выполнения упаковки и распаковки

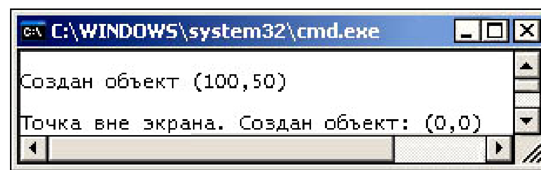


Рис. 15. Примеры вызовов конструктора класса

Листинг 12

```
using System;
class P12
{
    public static void Main()
    {
        int x = 5;
        object ob1 = x;  //выполнение упаковки
        object ob2 = new object();
        if (ob1 is int)
        {
            x = (int)ob1;  //выполнение распаковки
            Console.WriteLine("{0}\n", x.ToString());
        }
        else
            Console.WriteLine("{0}\n", "Invalid Cast1");
        if (ob2 is int)
        {
            int y = (int)ob2;  //выполнение распаковки
            Console.WriteLine("{0}\n", y.ToString());
        }
        else
            Console.WriteLine("{0}\n", "Invalid Cast2");
    }
}
```

Листинг 13

```
using System;
class Point
{
    private int x;
    private int y;
    public Point(int a, int b) //конструктор класса Point
    {
        if (a >= 0 & a < 800 & b >= 0 & b < 600)
        {
            this.x = a; this.y = b;
            Console.WriteLine(
                "\nСоздан объект ({0},{1})", this.x, this.y);
        }
        else
        {
            this.x = 0; this.y = 0;
            Console.WriteLine("\nТочка вне экрана. " +
                "Создан объект: ({0},{1})", this.x, this.y);
        }
    }
}
class P13
{
    static void Main(string[] args)
    {
        Point p1 = new Point(100, 50);  //Вызов конструктора
        Point p2 = new Point(-100, 50); //Вызов конструктора
    }
}
```

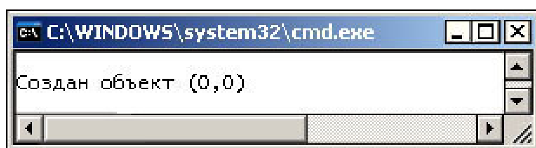


Рис. 16. Пример вызова конструктора по умолчанию

Если при этом необходим конструктор без параметров, то его нужно определить. Заметим, что определенный в классе конструктор без параметров не обязательно должен инициализировать поля объекта нулями (см. листинг 16, рис. 18).

Листинг 14

```
using System;
class Point
{   private int x;
    private int y;
    public int getX() { return x; }
    public int getY() { return y; }
}
class P14
{   public static void Main()
    {   Point p1 = new Point(); //вызов конструктора по умолчанию
        Console.WriteLine("\nСоздан объект ({0},{1})\n",
        p1.getX(), p1.getY());
    }
}
```

Листинг 15

```
using System;
class Point
{
    private int x;
    private int y;
    public Point(int a, int b)
    {
        if (a >= 0 & a < 800 & b >= 0 & b < 600)
        {
            this.x = a; this.y = b;
            Console.WriteLine(
                "\nСоздан объект ({0},{1})", this.x, this.y);
        }
        else
        {
            this.x = 0; this.y = 0;
            Console.WriteLine("\nТочка вне экрана. " +
                "Создан объект: ({0},{1})", this.x, this.y);
        }
    }
}
class P15
{
    public static void Main()
    {
        Point p1 = new Point(100, 50);
        Point p2 = new Point();
    }
}
```

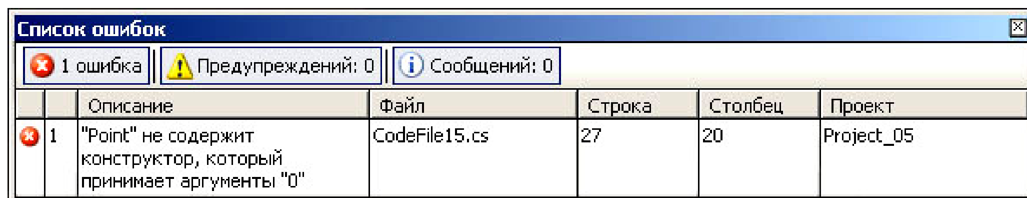


Рис. 17. Некорректная попытка вызова конструктора по умолчанию

4.3. СПИСОК ИНИЦИАЛИЗАЦИИ

C# позволяет из одного конструктора вызывать другой конструктор этого же класса с использованием ключевого слова **this**. Такой вызов называется списком инициализации (см. листинг 17, рис. 19).

4.4. ПОЛЯ-КОНСТАНТЫ И ПОЛЯ ТОЛЬКО ДЛЯ ЧТЕНИЯ

Поле может быть объявлено с ключевым словом **const**. Значение такого поля не может быть изменено во время работы программы. Значение поля-константы должно быть известно на этапе компиляции, поэтому для инициализации такого поля

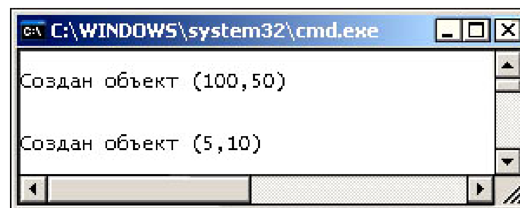


Рис. 18. Вызов конструктора с параметрами и конструктора без параметров

Листинг 16

```
using System;
class Point
{
    private int x;
    private int y;
    public Point()
    {
        this.x = 5; this.y = 10;
        Console.WriteLine(
            "\nСоздан объект ({0},{1})\n", this.x, this.y);
    }
    public Point(int x, int y)
    {
        this.x = x; this.y = y;
        Console.WriteLine(
            "\nСоздан объект ({0},{1})\n", this.x, this.y);
    }
}
class P16
{
    public static void Main()
    {
        Point p1 = new Point(100, 50);
        Point p2 = new Point();
    }
}
```

можно использовать только константные выражения. Если нужно сослаться на константу, определенную внешним типом, нужно добавить префикс имени типа (например, **Point.x**), так как поля-константы являются неявно статическими.

Поле может быть объявлено с ключевым словом **readonly**. Значение такого поля также не может быть изменено во время работы программы. Однако оно инициализируется на этапе выполнения программы посредством конструктора (см. листинг 18, рис. 20).

Листинг 17

```
using System;
class Point
{   private int x;
    private int y;
    public Point() : this(5, 10) //список инициализации
    { }
    public Point(int x, int y)
    {   this.x = x; this.y = y;
        Console.WriteLine(
        "\nСоздан объект ({0},{1})", this.x, this.y);
    }
}
class P17
{   public static void Main()
    {   Point p1 = new Point(100, 50);
        Point p2 = new Point();
    }
}
```

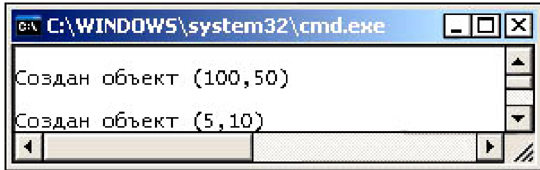


Рис. 19. Конструктор без параметров использует список инициализации

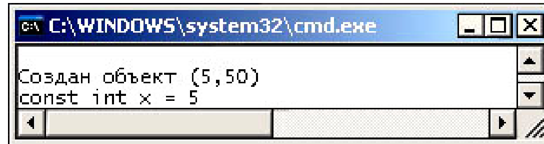


Рис. 20. Создание объекта с полем-константой и с полем только для чтения

В отличие от данных-констант, поля только для чтения не причисляются автоматически к группе статических. Если требуется использовать значение поля только для чтения на уровне класса, то следует указать ключевое слово **static**. Заметим, что статические члены могут воздействовать только на статические члены. Если, например, в статическом методе будет сделана попытка использовать нестатические члены, то компилятор выдаст сообщение об ошибке.

Листинг 18

```
using System;
class Point
{    public const int x = 5; //поле-константа
     private readonly int y; //поле только для чтения
     public Point(int y)
     {    this.y = y;
          Console.WriteLine(
               "\nСоздан объект ({0},{1})", Point.x, this.y);
     }
}
class P18
{
     public static void Main()
     {    Point p1 = new Point(50);
          Console.WriteLine("const int x = {0}", Point.x);
     }
}
```

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 fifth of a series of articles, devoted to «a zero level» of language C#. Methods, classes, references, and constructors in C# 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