

Лекция 10-11

Классы

1. Основные понятия
2. Данные: поля и константы
3. Методы
4. Параметры методов
5. Конструкторы
6. Свойства

Класс является **типом данных**, определяемым пользователем. Он должен представлять собой одну логическую сущность, например, являться моделью реального объекта или процесса.

Элементами класса являются **данные** и **функции**, предназначенные для их обработки.

Описание класса содержит ключевое слово **class**, за которым следует его имя, а далее в фигурных скобках – тело класса, то есть список его элементов. Кроме того, для класса можно задать его базовые классы (предки) и ряд необязательных атрибутов и спецификаторов, определяющих различные характеристики класса:

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

Как видите, обязательными являются только ключевое слово **class**, а также **имя** и **тело** класса.

Имя класса задается программистом по общим правилам C#.

Тело класса – это список описаний его элементов, заключенный в фигурные скобки.

Список может быть пустым, если класс не содержит ни одного элемента. Таким образом, *простейшее описание класса* может выглядеть так:

```
class Demo {}
```

Спецификаторы определяют **свойства** класса, а также **доступность** класса для других элементов программы. Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется **вложенным**.

В зависимости от места описания класса некоторые из этих спецификаторов могут быть запрещены.

Таблица 1 – Спецификаторы класса

№	Спецификатор	Описание
1	new	Используется для вложенных классов. Задаёт новое описание класса взамен унаследованного от предка. Применяется в иерархиях объектов
2	public	Доступ не ограничен
3	protected	Используется для вложенных классов. Доступ только из элементов данного и производных классов
4	internal	Доступ только из данной программы (сборки)
5	protected internal	Доступ только из данного и производных классов или из данной программы (сборки)
6	private	Используется для вложенных классов. Доступ только из элементов класса, внутри которого описан данный класс
7	abstract	Абстрактный класс. Применяется в иерархиях объектов
8	sealed	Бесплодный класс. Применяется в иерархиях объектов
9	static	Статический класс. Введен в версию языка 2.0. Рассматривается в разделе «Конструкторы»

Спецификаторы 2-6 называются **спецификаторами доступа**. Они определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут присутствовать в описании только в вариантах, приведенных в таблице, а также могут комбинироваться с остальными спецификаторами.

Будем изучать классы, которые описываются в пространстве имен непосредственно (то есть не вложенные классы). Для таких классов допускаются только два спецификатора: `public` и `internal`. По умолчанию, то есть если ни один спецификатор доступа не указан, подразумевается спецификатор `internal`.

Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых *экземплярами*, или *объектами*, класса.

Объекты создаются явным или неявным образом, то есть либо программистом, либо системой.

Программист создает экземпляр класса с помощью операции `new`, например:

```
Demo a = new Demo{}; // создание экземпляра класса Demo
Demo b = new Demo{}; // создание другого экземпляра класса Demo
```

При создании **объекта** в памяти выделяется *отдельная область*, в которой хранятся его данные.

Кроме того, в классе могут присутствовать *статические элементы*, которые существуют в *единственном экземпляре* для всех объектов класса.

Часто *статические данные* называют *данными класса*, а остальные – *данными экземпляра*.

Функциональные элементы класса не тиражируются, то есть всегда хранятся в единственном экземпляре.

Для работы с данными класса используются **методы класса** (статические методы),

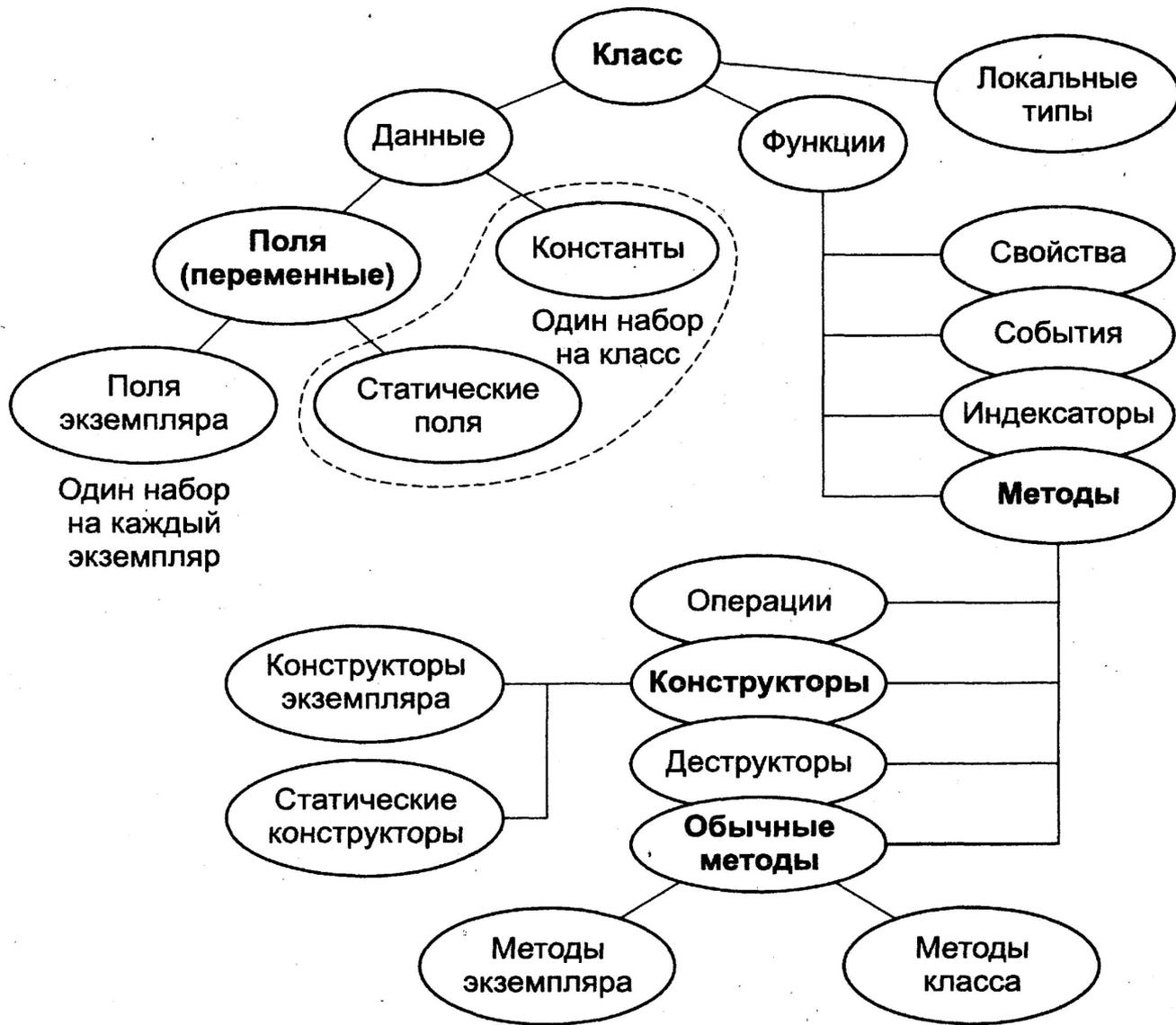
для работы с данными экземпляра – методы экземпляра, или просто **методы**.

До сих пор мы использовали в программах только один вид функциональных элементов класса – **методы**.

Поля и методы являются основными элементами класса.

Кроме того, в классе можно задавать целую гамму других элементов:

свойства, события, индексы, операции, конструкторы, деструкторы, а также *типы* (рис. 1).



Состав класса

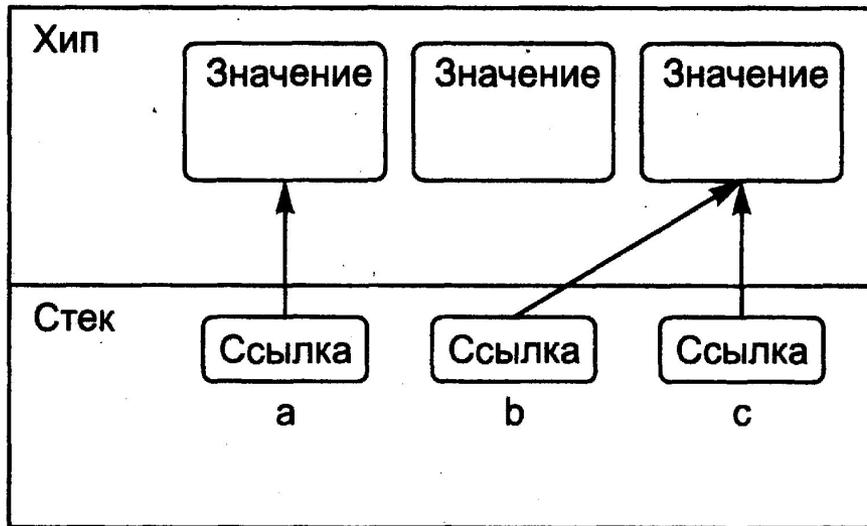
Приведем краткое описание элементов класса:

- *Константы* класса хранят неизменяемые значения, связанные с классом.
- *Поля* содержат данные класса.
- *Методы* реализуют вычисления или другие действия, выполняемые классом или экземпляром.
- *Свойства* определяют характеристики класса в совокупности со способами их задания и получения, то есть методами записи и чтения.
- *Конструкторы* реализуют действия по инициализации экземпляров или класса в целом.
- *Деструкторы* определяют действия, которые необходимо выполнить до того, как объект будет уничтожен.
- *Индексаторы* обеспечивают возможность доступа к элементам класса по их порядковому номеру.
- *Операции* задают действия с объектами с помощью знаков операций.
- *События* определяют уведомления, которые может генерировать класс.
- *Типы* – это типы данных, внутренние по отношению к классу.

Первые пять видов элементов класса рассмотрим сейчас, остальные – позже.

Присваивание и сравнение объектов. Механизм выполнения присваивания один и тот же для величин любого типа, как ссылочного, так и значимого, однако результаты различаются.

При присваивании значения копируется значение, а при присваивании ссылки – ссылка, поэтому после присваивания одного объекта другому мы получим две ссылки, указывающие на одну и ту же область памяти:



Объекты

Присваивание объектов

Создано три объекта, a, b и c, затем выполнено *присваивание* $b = c$. Старое значение b недоступно и очищается сборщиком мусора. Если изменить значение одной величины ссылочного типа, это может отразиться на другой: если изменить объект через ссылку c, объект b также изменит значение.

Аналогичная ситуация с операцией *проверки на равенство*.

Величины значимого типа равны, если равны их значения.

Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке: объекты b и c равны, но a не равно b даже при равенстве их значений или если они обе равны null).

2. Данные: поля и константы

Данные, содержащиеся в классе, могут быть переменными или константами и задаются в соответствии с правилами, рассмотренными ранее.

Переменные, описанные в классе, называются *полями* класса.

При описании элементов класса можно также указывать *атрибуты* и *спецификаторы*, задающие различные характеристики элементов.

Синтаксис описания элемента данных:

**[атрибуты] [спецификаторы] [const] тип имя
[= начальное_значение]**

Возможные спецификаторы полей и констант перечислены в табл. 2.

Для констант можно использовать только спецификаторы 1-6.

Таблица 2 – Спецификаторы полей и констант класса

№	Спецификатор	Описание
1	new	Новое описание поля, скрывающее унаследованный элемент класса
2	public	Доступ к элементу не ограничен
3	protected	Доступ только из данного и производных классов
4	internal	Доступ только из данной сборки
5	protected internal	Доступ только из данного и производных классов и из данной сборки
6	private	Доступ только из данного класса
7	static	Одно поле для всех экземпляров класса
8	readonly	Поле доступно только для чтения
9	volatile	Поле может изменяться другим процессом или системой

По умолчанию элементы класса считаются закрытыми (`private`). Для полей класса этот вид доступа является предпочтительным, поскольку поля определяют *внутреннее строение класса, которое должно быть скрыто от пользователя*.

Все методы класса имеют непосредственный доступ к его закрытым полям.

- Поля, описанные со спецификатором **static**,
- константы

существуют *в единственном экземпляре для всех объектов класса*, поэтому к ним обращаются не через имя экземпляра, а через имя класса.

Если класс содержит только статические элементы, экземпляр класса создавать не требуется.

Именно этим фактом мы пользовались в предыдущих листингах.

Обращение к полю класса выполняется с помощью операции доступа – **точка**:

- справа от точки задается *имя поля*,
- слева – *имя экземпляра* для обычных полей или *имя класса* для статических:

имя экземпляра (класса).имя поля

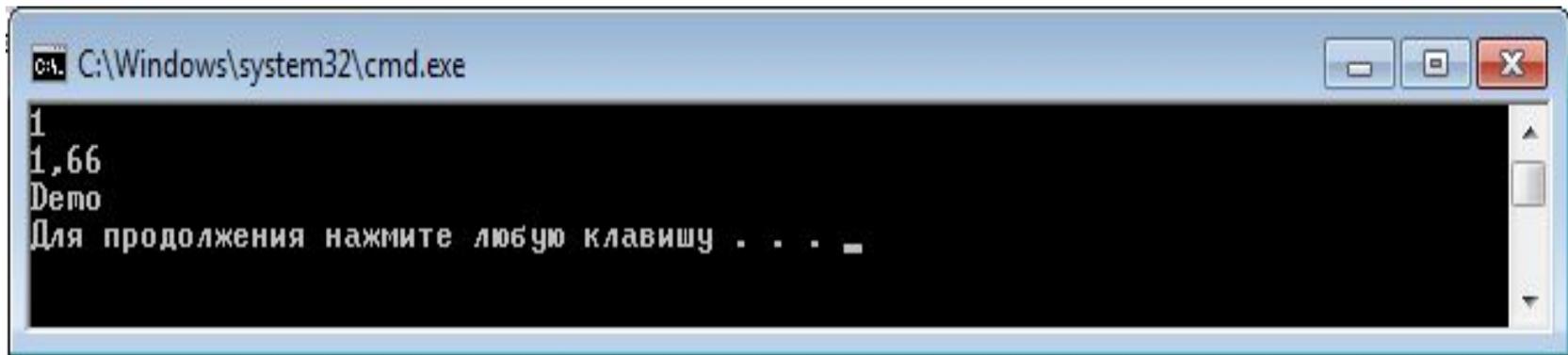
В листинге 1 приведены:

- *пример простого класса Demo*;
- *два способа обращения к его полям*.

Листинг 1 – Класс **Demo**, содержащий поля и константу
Добавим в проект класс **Demo**.

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
        public int a = 1;           // поле данных
        public const double c = 1.66; // константа
        public static string s = "Demo"; // статическое поле класса
        double y;                   // закрытое поле данных
    }
    class Program
    { static void Main()
        {
            Demo x = new Demo(); // создание экземпляра класса Demo
            Console.WriteLine(x.a); // x.a - обращение к полю класса
            Console.WriteLine(Demo.c); // Demo.c - обращение к константе
            Console.WriteLine( Demo.s); // обращение к статическому
        }
    }
}
```

полю



```
C:\Windows\system32\cmd.exe
1
1,66
Demo
Для продолжения нажмите любую клавишу . . .
```

Поле `y` вывести на экран аналогичным образом не удастся: оно является закрытым, то есть недоступно извне (из класса `Program`). Поскольку значение этому полю явным образом не присвоено, среда присваивает ему значение ноль.

Все **поля сначала автоматически инициализируются нулем соответствующего типа**, например:

- полям типа `int` присваивается `0`,
- ссылкам на объекты – значение `null`).

После этого полю присваивается значение, заданное при его явной инициализации. Задание начальных значений для статических полей выполняется при инициализации класса, а обычных – при создании экземпляра.

Поля со спецификатором `readonly` предназначены только для чтения. Установить значение такого поля можно либо при его описании, либо в конструкторе (конструкторы рассматриваются далее).

3. Методы

Метод – это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или экземпляром. Методы определяют поведение класса.

Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может столько раз, сколько необходимо. Один и тот же метод может обрабатывать различные данные, переданные ему в качестве аргументов.

Синтаксис метода:

заголовок_метода

тело_метода

где **заголовок_метода:**

[атрибуты] [спецификаторы] тип имя_метода ([параметры])

тело_метода задает действия, выполняемые методом:

чаще всего представляет собой **блок** – последовательность операторов в фигурных скобках.

При описании методов можно использовать **спецификаторы** 1-7 из табл. 2, имеющие тот же смысл, что и для полей, а также спецификаторы **virtual**, **sealed**, **override**, **abstract** и **extern**.

Чаще всего для методов задается спецификатор доступа **public**, ведь методы составляют **интерфейс класса** – то, с чем работает пользователь, поэтому они должны быть доступны.

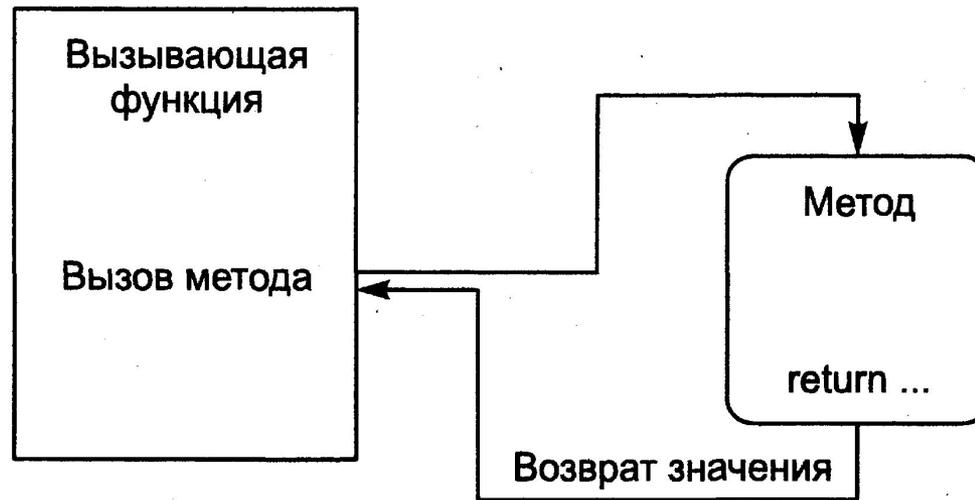
Внимание – Статические (**static**) методы, или методы класса, можно вызывать, не создавая экземпляр объекта. Именно таким образом используется метод **Main**.

Пример простейшего метода:

```
public double Gety() // метод для получения поля y из листинга
1
{
    return y;
}
```

Тип определяет, значение какого типа вычисляется с помощью метода.

Часто употребляется термин «метод возвращает значение», поскольку после выполнения метода происходит возврат в то место вызывающей функции, откуда был вызван метод, и передача туда значения выражения, записанного в операторе **return**:



Вызов метода

Если метод не возвращает никакого значения, в его заголовке задается тип **void**, а оператор **return** отсутствует.

Параметры используются для обмена информацией с методом.

Параметр – это локальная переменная, которая при вызове метода принимает значение соответствующего аргумента.

Область действия параметра – весь *метод*.

Пример. Чтобы вычислить значение синуса для вещественной величины **x**, мы передаем ее в качестве аргумента в метод **Sin** класса **Math**, а чтобы вывести значение этой переменной на экран, мы передаем ее в метод **WriteLine** класса **Console**:

```
double x = 0.1;  
double y = Math.Sin(x);  
Console.WriteLine(x);
```

При этом метод **Sin** возвращает в точку своего вызова вещественное значение синуса, которое присваивается переменной **y**, а метод **WriteLine** ничего не возвращает.

Метод, не возвращающий значение, вызывается отдельным оператором,

метод, возвращающий значение, – в составе выражения в правой части оператора присваивания.

Параметры, описываемые в заголовке метода, определяют множество значений **аргументов**, которые можно передавать в метод. Список аргументов при вызове как бы накладывается на список параметров, поэтому они должны попарно соответствовать друг другу. Правила соответствия подробно рассматриваются далее.

Для каждого **параметра** должны задаваться его **тип** и **имя**.

Например, заголовок метода **Sin** выглядит следующим образом:

```
public static double Sin(double a);
```

Имя метода вместе с количеством, типами и спецификаторами его параметров представляет собой **сигнатуру метода** – то, чем один метод отличают от других. В классе не должно быть методов с одинаковыми сигнатурами.

В листинге 2 в класс **Demo** добавлены методы установки и получения значения поля **y** (на самом деле для подобных целей используются не методы, а свойства, которые рассматриваются чуть позже).

Кроме того, статическое поле **s** закрыто, то есть определено по умолчанию как **private**, а для его получения описан метод **Gets** – пример статического метода.

Листинг 2 – Простейшие методы

```
using System;
namespace ConsoleApplication1
{
class Demo
    {
    public int a = 1;
    public const double c = 1.66;
    static string s = "Demo";
    double y;

    public double Gety(); // метод получения поля y
    {
        return y;
    }
    public void Sety(double y_) // метод установки поля y
    {
        y = y_;
    }
    public static string Gets() // метод получения поля s
    {
        return s;
    }
}
}
```

```

class Class1
{ static void.Main()
  {
    Demo x = new Demo();
    x.Sety(0.12);           // вызов метода установки поля у
    Console.WriteLine(x.Gety()); // вызов метода получения поля
у
    Console.WriteLine(Demo.Gety()); // вызов метода получения поля с
//Console.WriteLine(Gety()); // при вызове из др. метода этого
объекта
  }
}
}

```

```

C:\Windows\system32\cmd.exe
0,12
Demo
Для продолжения нажмите любую клавишу . . .

```

Как видите, методы класса имеют непосредственный доступ к его закрытым полям. Метод, описанный со спецификатором **static**, должен обращаться только к статическим полям класса.

Обратите внимание: *статический метод* вызывается через имя класса, а *обычный* – через имя экземпляра.

Примечание – При вызове метода из другого метода того же класса имя класса/экземпляра можно не указывать.

4. Параметры методов

Рассмотрим более подробно, каким образом метод обменивается информацией с вызвавшим его кодом.

При вызове метода выполняются следующие действия:

1. Вычисляются выражения, стоящие на месте аргументов.
2. Выделяется память под параметры метода в соответствии с их типом.
3. Каждому из параметров сопоставляется соответствующий аргумент (аргументы как бы накладываются на параметры и замещают их).
4. Выполняется тело метода.
5. Если метод возвращает значение, оно передается в точку вызова; если метод имеет тип `void`, управление передается оператору, следующему после вызова.

При этом проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение.

Листинг 3 иллюстрирует этот процесс.

Листинг 3 – Передача параметров методу

```
using System;
namespace ConsoleApplication1
{
    class Program
    {
static int Max(int a, int b) // метод выбора максимального
значения
        {
            if (a > b) return a;
            else return b;
        }
static void Main(string[] args)
        {
            int a = 2, b = 4;
            int x = Max(a, b);      // вызов метода Max
            Console.WriteLine(x);   // результат: 4
            short t1 = 3, t2 = 4;
            int y = Max(t1, t2);    // вызов метода Max
            Console.WriteLine(y);   // результат: 4
            int z = Max(a + t1, t1 / 2 * b); // вызов метода Max
            Console.WriteLine(z);   // результат: 5
        }
    }
}
```

Результат выполнения:



```
C:\Windows\system32\cmd.exe
4
4
5
Для продолжения нажмите любую клавишу . . .
```

Главное требование при передаче параметров: аргументы при вызове метода должны записываться в том же порядке, что и параметры в заголовке метода.

Правила соответствия типов аргументов и параметров описаны далее.

Количество аргументов должно соответствовать количеству параметров.

На имена никаких ограничений не накладывается: имена аргументов могут как совпадать, так и не совпадать с именами параметров.

Существуют **два способа передачи параметров**:

- по значению;
- по ссылке.

При передаче по значению метод получает копии значений аргументов, и операторы метода работают с этими копиями. Доступа к исходным значениям аргументов у метода нет, а следовательно, нет и возможности их изменить.

При передаче по ссылке (по адресу) метод получает копии адресов аргументов, он осуществляет доступ к ячейкам памяти по этим адресам и может изменять исходные значения аргументов, модифицируя параметры.

В С# для обмена данными между вызывающей и вызываемой функциями предусмотрено **четыре типа параметров**, описываемые:

- параметры-ссылки – с помощью ключевого слова **ref**;
- выходные параметры – с помощью ключевого слова **out**;
- параметры-массивы – с помощью ключевого слова **params**;
- параметры-значения – без помощи ключевых слов.

Ключевое слово предшествует описанию типа параметра.

Если оно *опущено*, параметр считается параметром-значением.

Параметр-массив может быть только один и должен располагаться последним в списке, например:

```
public int Calculate(int a, ref int b, out int c, params int[] d)  
...
```

Рассмотрим остальные типы параметров.

О параметрах-массивах поговорим позже.

4.1. Параметры-значения

Параметр-значение описывается в заголовке метода следующим образом:

тип имя

Пример. Заголовок метода, имеющего один параметр-значение целого типа:

```
void P(int x)
```

Имя параметра может быть произвольным. Параметр *x* – локальная переменная, которая получает свое значение из вызывающей функции при вызове метода. В метод передается копия значения аргумента.

Механизм передачи следующий:

из ячейки памяти, в которой хранится переменная, передаваемая в метод, берется ее значение и копируется в специальную область памяти – область параметров.

Метод работает с этой копией, следовательно, доступа к ячейке, где хранится сама переменная, не имеет.

По завершении работы метода область параметров освобождается.

Таким образом, для параметров-значений используется, как вы догадались, ***передача по значению***.

Ясно, что этот способ годится только для величин, которые не должны измениться после выполнения метода, то есть для его ***исходных данных***.

При вызове метода на месте параметра, передаваемого по значению, может находиться:

- *выражение*,
- а также, конечно, его частные случаи – *переменная* или *константа*.

Должно существовать неявное преобразование типа выражения к типу параметра.

Пример. Пусть в вызывающей функции описаны переменные и им до вызова метода присвоены значения:

```
int x = 1;  
sbyte c = 1;  
ushort y = 1;
```

Тогда следующие вызовы метода **P**, заголовок которого был описан ранее, будут синтаксически правильными:

```
P(x); P(c); P(y); P(200); P(x/4 + 1);
```

4.2. Параметры-ссылки

Во многих методах все величины, которые метод должен получить в качестве исходных данных, описываются в списке параметров, а величина, которую вычисляет метод как результат своей работы, возвращается в вызывающий код с помощью оператора **return**. Очевидно, что если метод должен возвращать более одной величины, такой способ не годится. Еще одна проблема возникает, если в методе требуется изменить значение каких-либо передаваемых в него величин. В этих случаях используются **параметры-ссылки**.

Признаком параметра-ссылки является ключевое слово **ref** перед описанием параметра:

ref тип имя

Пример. Заголовок метода, имеющего один параметр-ссылку целого типа:

void P(ref int x)

При вызове метода в область параметров *копируется не значение аргумента, а его адрес*, и метод через него имеет доступ к ячейке, в которой хранится аргумент.

Таким образом, *параметры-ссылки передаются по адресу* («передача по ссылке»). Метод работает непосредственно с переменной из вызывающей функции и, следовательно, может ее изменить, поэтому, если в методе требуется изменить значения параметров, они должны передаваться только по ссылке.

Внимание – При вызове метода на месте параметра-ссылки может находиться только ссылка на инициализированную переменную точно того же типа. Перед именем параметра указывается ключевое слово `ref`.

Исходные данные передавать в метод по ссылке не рекомендуется, чтобы исключить возможность их непреднамеренного изменения.

Листинг 4 – Параметры-значения и параметры-ссылки

```
using System;
namespace ConsoleApplication1
{ class Class1
    {
        static void P(int a, ref int b)
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b);
        }
    static void Main()
    { int a = 2, b = 4;
      Console.WriteLine("до вызова {0} {1}", a, b);
      P(a, ref b);
      Console.WriteLine("после вызова {0} {1}", a, b );
    }
}
}
```

```
using System;
```

```
namespace ConsoleApplication1
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void P(int a, ref int b)
```

```
        {
```

```
            a = 44; b = 33;
```

```
            Console.WriteLine("внутри метода {0} {1}", a, b);
```

```
        }
```

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

```
        {
```

```
            int a = 2, b = 4;
```

```
            Console.WriteLine("до вызова {0} {1}", a, b);
```

```
            P(a, ref b);
```

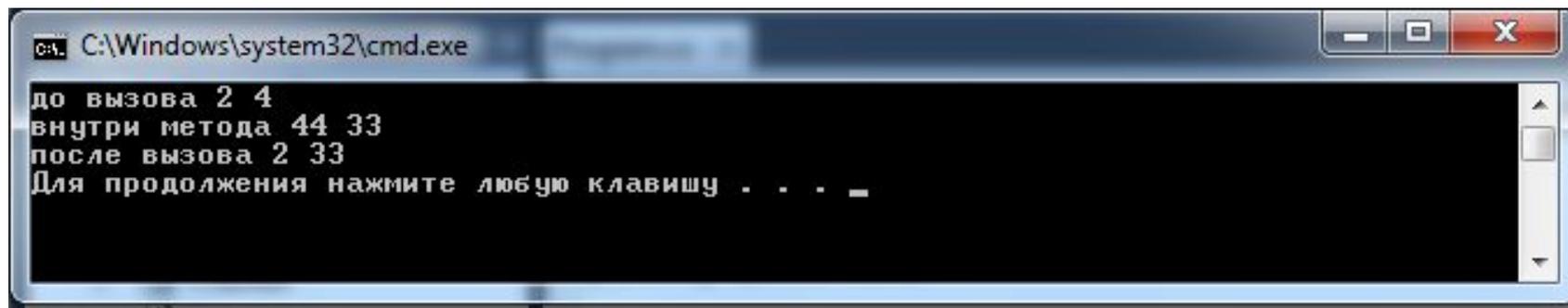
```
            Console.WriteLine("после вызова {0} {1}", a, b);
```

```
        }
```

```
    }
```

```
}
```

Результаты работы этой программы:



```
C:\Windows\system32\cmd.exe
до вызова 2 4
внутри метода 44 33
после вызова 2 33
Для продолжения нажмите любую клавишу . . . _
```

Значение переменной **a** в функции **Main** не изменилось: она передавалась *по значению*, а значение переменной **b** изменилось: она передана *по ссылке*.

Если *передать в метод* не величины значимых типов, а *экземпляры классов*, то есть величины ссылочных типов, то метод может их изменить:.

переменная-объект на самом деле хранит ссылку на данные, расположенные в динамической памяти, и именно эта ссылка передается в метод либо по адресу, либо по значению. В обоих случаях метод получает в свое распоряжение фактический адрес данных и, следовательно, может их изменить.

Совет – Для простоты можно считать, что объекты всегда передаются по ссылке.

Разница между передачей объектов *по значению* и *по ссылке*:

в последнем случае можно изменить саму ссылку,

то есть после вызова метода она может указывать на другой объект.

4.3. Выходные параметры

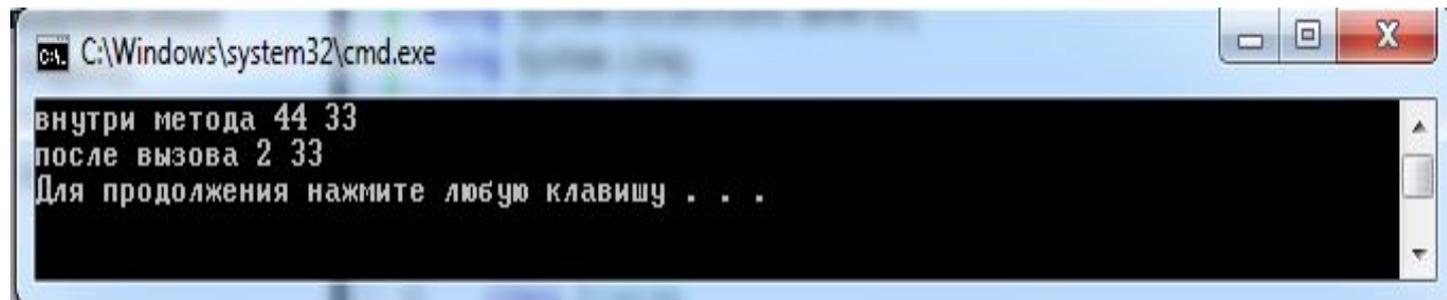
Часто нужны *методы, формирующие несколько величин*, например, если в методе создаются объекты или инициализируются ресурсы.

Тогда становится неудобным *ограничение параметров-ссылок: необходимость присваивания значения аргументу до вызова метода*.

Это ограничение снимает спецификатор **out**.

out-параметру *нужно присвоить значение внутри метода* (компилятор за этим следит), зато в вызывающем коде можно ограничиться описанием переменной без инициализации.

Изменим описание второго параметра в листинге 4 так, чтобы он стал выходным (листинг 5, следующий слайд). Результат:



```
C:\Windows\system32\cmd.exe
внутри метода 44 33
после вызова 2 33
Для продолжения нажмите любую клавишу . . .
```

При *вызове метода* перед соответствующим параметром тоже указывается ключевое слово **out**.

Совет – В списке параметров записывайте сначала все входные параметры, затем – все ссылки и выходные параметры. Давайте параметрам имена, по которым можно получить представление об их назначении.

Листинг 5 – Выходные параметры

```
using System;

namespace ConsoleApplication3
{
    class Program
    {
        static void P(int a, out int b) // out-параметр b
        {
            a = 44; b = 33;
            Console.WriteLine("внутри метода {0} {1}", a, b);
        }

        static void Main(string[] args)
        {
            int a = 2, b; // параметр b не инициализируется
            P(a, out b); // при вызове метода - out перед b
            Console.WriteLine("после вызова {0} {1}", a, b);
        }
    }
}
```

4.4. Ключевое слово **this**

Каждый *объект содержит свой экземпляр полей класса*.

Методы находятся в памяти в *единственном экземпляре* и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается *скрытый параметр **this***, в котором хранится *ссылка на вызвавший функцию экземпляр*.

В *явном виде* параметр **this** применяется для того, чтобы вернуть из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

```
class Demo
{
    double y;
    public Demo T()// метод T возвращает ссылку на экземпляр
    {
        return this;
    }
    public void Sety(double y)
    {
        this.y = y; // полю y присваивается значение параметра y
    }
}
```

5. Конструкторы

Конструктор предназначен для *инициализации объекта*. Он вызывается автоматически при создании объекта класса с помощью операции **new**.

Имя конструктора совпадает с *именем класса*.

Свойства конструкторов:

- Конструктор **не возвращает значение**, даже типа **void**.
- Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.
- Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается ноль, полям ссылочных типов – значение **null**.
- Конструктор, вызываемый без параметров, называется **конструктором по умолчанию**.

До сих пор мы задавали *начальные значения полей класса при описании класса* (см., например, листинг 1). Это удобно в том случае, когда *для всех экземпляров класса начальные значения некоторого поля одинаковы*.

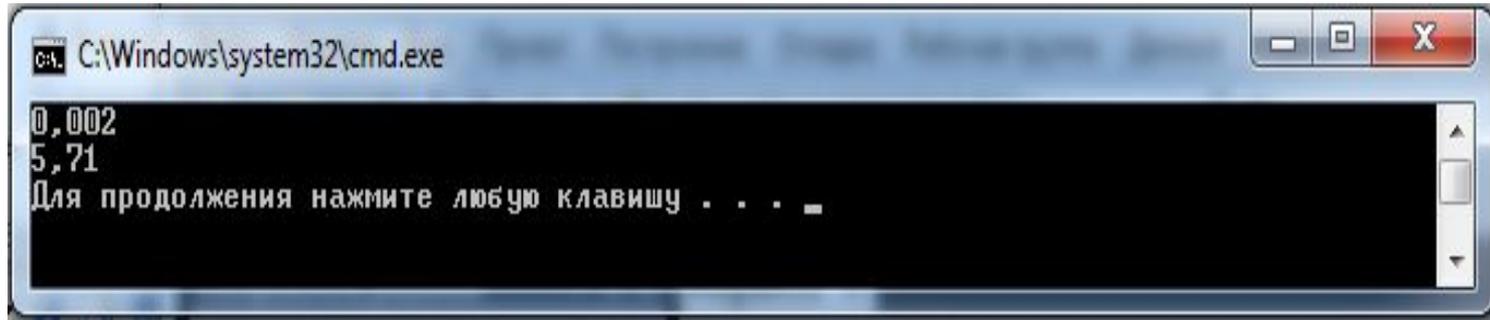
Если же при создании объектов требуется присваивать полю разные значения, это следует делать в конструкторе.

В листинге 6 в класс **Demo** добавлен **конструктор**, а **поля сделаны закрытыми** (ненужные в данный момент элементы опущены). В программе создаются два объекта с различными значениями полей.

Листинг 6 – Класс с конструктором

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
        public Demo(int a, double y) // конструктор с параметрами
        {
            this.a = a;
            this.y = y;
        }
        public double Gety() // метод получения поля y
        {
            return y;
        }
        int a;
        double y;
    }
    class Program
    {
        static void Main(string[] args)
        {
            Demo a = new Demo(300, 0.002); // ВЫЗОВ КОНСТРУКТОРА
            Console.WriteLine(a.Gety()); // результат: 0.002
            Demo b = new Demo(1, 5.71); // ВЫЗОВ КОНСТРУКТОРА
            Console.WriteLine(b.Gety()); // результат: 5.71
        }
    }
}
```

Результат:



```
C:\Windows\system32\cmd.exe
0,002
5,71
Для продолжения нажмите любую клавишу . . .
```

Часто бывает удобно задать в классе **несколько конструкторов**, чтобы обеспечить возможность инициализации объектов разными способами:

```
class Demo
{
    public Demo(int a) // конструктор 1
    {   this.a = a;
        this.y = 0.002;
    }
    public Demo(double y) // конструктор 2
    {   this.a = 1;
        this.y = y;
    }
    ...
}
...
Demo x = new Demo(300); // вызов конструктора 1
Demo y = new Demo(5.71); // вызов конструктора 2
```

Все конструкторы должны иметь *разные сигнатуры*.

Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-нибудь, удобно вызвать *первый конструктор из второго*.

Для этого используется уже известное вам ключевое слово **this** в другом контексте, например:

```
class Demo
{
    public Demo(int a) // конструктор 1
    {
        this.a = a;
    }
    public Demo(int a, double y) : this(a)// вызов конструктора 1
    {
        this.y = y;
    }
    ...
}
```

Конструкция, находящаяся после двоеточия, называется *инициализатором*, то есть тем кодом, который исполняется до начала выполнения тела конструктора.

Как известно, все классы в C# имеют общего предка – класс **object**.
Конструктор любого класса, если не указан инициализатор, автоматически вызывает конструктор своего предка.

Это можно сделать и явным образом с помощью ключевого слова **base**, обозначающего *конструктор базового класса*.

Таким образом, первый конструктор из предыдущего примера можно записать и так:

```
public Demo(int a): base()    // конструктор 1
{
this.a = a;
}
```

Примечание – Конструктор базового класса вызывается явным образом в тех случаях, когда ему требуется передать параметры.

До сих пор речь шла об «обычных» конструкторах, или **конструкторах экземпляра**.

Существует *второй тип конструкторов* – **статические конструкторы**, или **конструкторы класса**.

Конструктор экземпляра инициализирует **данные экземпляра**, **конструктор класса** – **данные класса**.

Статический конструктор *не имеет параметров*, его *нельзя вызвать явным образом*.

Система сама определяет момент, в который требуется его выполнить.

Гарантируется только, что это происходит до создания первого экземпляра объекта и до вызова любого статического метода.

Если классы содержат только статические данные, то создавать экземпляры таких объектов не имеет смысла.

Примечание – В классе, состоящем только из статических элементов (полей и констант), описывать статический конструктор не обязательно, начальные значения полей удобнее задать при их описании.

Чтобы подчеркнуть этот факт, в первой версии C# описывали пустой **закрытый (private) конструктор**. Это предотвращало попытки создания экземпляров класса. В листинге 5.7 приведен пример класса, который служит для группировки величин. Создавать экземпляры этого класса запрещено.

В версию 2.0 введена возможность описывать статический класс, то есть класс с модификатором **static**. Экземпляры такого класса создавать запрещено, и кроме того, от него запрещено наследовать. Все элементы такого класса должны явным образом объявляться с модификатором **static** (константы и вложенные типы классифицируются как статические элементы автоматически). Конструктор экземпляра для статического класса задавать, естественно, запрещается.

Листинг 7 – Пример статического класса (начиная с версии 2.0)

```
using System;
namespace ConsoleApplication1
{
    static class D
    {
        static int a = 200;
        static double b = 0.002;
        public static void Print()
        {
            Console.WriteLine("a = " + a);
            Console.WriteLine("b = " + b);
        }
    }
}
```

```
class Class1
{ static void Main()
  {
    D.Print();
  }
}
```

В качестве «сквозного» примера для демонстрации работы с различными элементами класса, создадим *класс, моделирующий персонаж компьютерной игры*. Для этого требуется задать его *свойства* (например, количество щупальцев, силу или наличие гранатомета) и *поведение*. Естественно, пример (листинг 8) является схематичным, поскольку приводится лишь для демонстрации синтаксиса.

Листинг 8 – Класс **Monster**

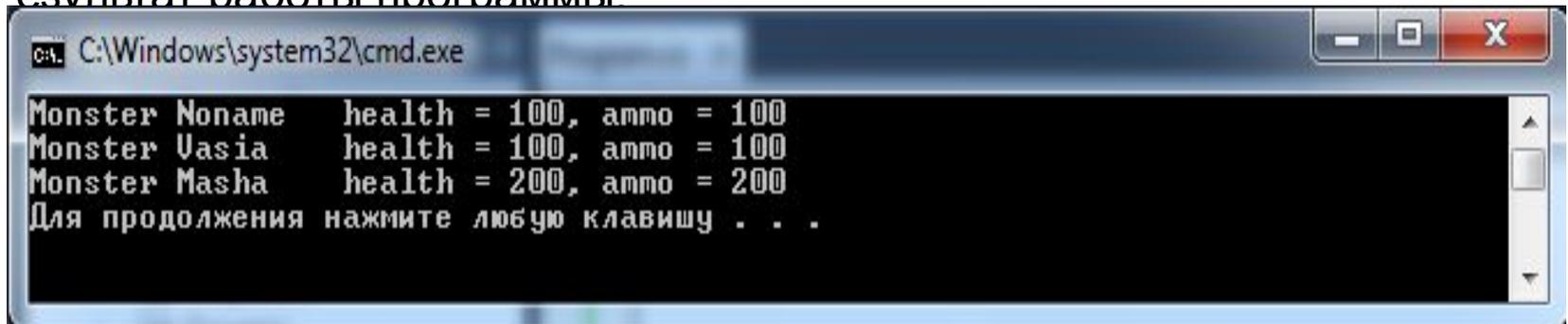
```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        public Monster()
        {
            this.name = "Noname";
            this.health = 100;
            this.ammo = 100;
        }
        public Monster(string name):this()
        {
            this.name = name;
        }
    }
}
```

```
public Monster(int
health, int ammo, string
name)
{
    this.name = name;
    this.health = health;
    this.ammo = ammo;
}
```

```
public string GetName()
{
    return name;
}
public int GetHealth()
{
    return health;
}
public int GetAmmo()
{
    return ammo;
}
public void Passport()
{
    Console.WriteLine("Monster {0} \t health = {1}, ammo = {2}",
name, health, ammo);
}
string name; // закрытые поля
int health, ammo;
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Monster X = new Monster();
        X.Passport();
        Monster Vasia = new Monster("Vasia");
        Vasia.Passport();
        Monster Masha = new Monster(200, 200, "Masha");
        Masha.Passport();
    } // В классе Monster три закрытых поля (name, health,
аммо),
    } // четыре метода (GetName, GetHealth, GetAmmo и Passport)
} // три конструктора, позволяющие задать при создании объекта
// ни одного, один или три параметра.
```

Результат работы программы:



```
C:\Windows\system32\cmd.exe
Monster Noname health = 100, ammo = 100
Monster Vasia health = 100, ammo = 100
Monster Masha health = 200, ammo = 200
Для продолжения нажмите любую клавишу . . .
```

6. Свойства

Свойства служат для организации доступа к полям класса.

Как правило, свойство:

- связано с *закрытым полем* класса;
- 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 версии C# 2.0 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,

если свойство имеет доступ **protected internal**,

его части могут объявляться как **internal**, **protected** или **private**.

Синтаксис свойства в версии 2.0 имеет вид:

```
[атрибуты] [спецификаторы] тип имя_свойства
{
    [[атрибуты] [спецификаторы] get код_доступа]
    [[атрибуты] [спецификаторы] set код_доступа]
}
```

Пример описания свойств:

```
public class Button: Control
{
    private string caption;//закрытое поле, с которым связано
    СВОЙСТВО
    public string Caption {// свойство
        get { // способ получения свойства
            return caption;
        }
        set { // способ установки свойства
            if (caption != value)
            {
                caption = value;
            }
        }
    }
}
...
```

Примечание – Двоеточие между именами **Button** и **Control** в заголовке класса **Button** означает, что класс **Button** является производным от класса **Control**.

Метод записи обычно содержит действия по проверке допустимости устанавливаемого значения, метод чтения может содержать, например, поддержку счетчика обращений к полю.

В программе свойство выглядит как поле класса, например:

```
Button ok = new Button();  
ok.Caption = "OK";    // вызывается метод установки свойства  
string s = ok.Caption; // вызывается метод получения свойства
```

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)
        {
            x = new ComplexObject();// создание объекта при 1-м обращении
        }
        return x;
    }
}
...
```

//Добавим в класс **Monster**, описанный в листинге 8, свойства, позволяющие работать с закрытыми полями этого класса. Свойство **Name** сделаем доступным только для чтения, поскольку имя объекта задается в конструкторе и его изменение не предусмотрено, в свойствах **Health** и **Ammo** введем проверку на положительность устанавливаемой величины. Код класса несколько разрастется, зато упростится его использование.

ЛИСТИНГ 9 – Класс **Monster** со свойствами

```
using System;
namespace ConsoleApplication1
{
    class Monster
    { public Monster()
      {
          this.health = 100;
          this.ammo    = 100;
          this.name     = "Noname";
      }
    public Monster(string name) : this()
      {
          this.name = name;
      }
    public Monster(int health, int ammo, string name)
      {
          this.health = health;
          this.ammo    = ammo;
          this.name     = name;
      }
    }
```

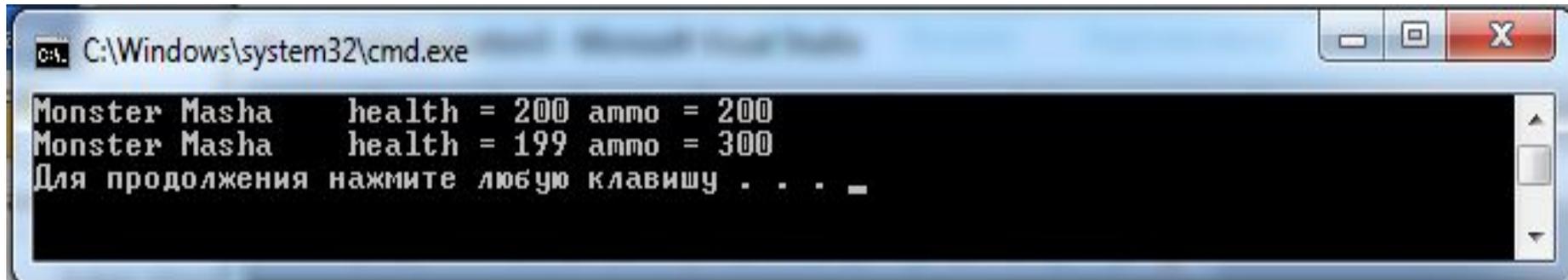
```
public int Health // СВОЙСТВО Health связано с полем health
{
    get
    { return health;
    }
    set
    { if (value > 0) health = value;
      else health =0;
    }
}
public int Ammo // СВОЙСТВО Ammo связано с полем ammo
{
    get
    {
        return ammo;
    }

    set
    {
        if (value > 0) ammo = value;
        else ammo = 0;
    }
}
```

```
public string Name// СВОЙСТВО Name СВЯЗАНО С ПОЛЕМ name
{
    get
        {return name;
        }
}
public void Passport()
    {Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
name, health, ammo);
    }
string name; // закрытые поля
int health, ammo;
}
class Program
    {
        static void Main(string[] args)
        {Monster Masha = new Monster(200, 200, "Masha");
        Masha. Passport();
        --Masha.Health; // ИСПОЛЬЗОВАНИЕ СВОЙСТВ
        Masha.Ammo += 100; // ИСПОЛЬЗОВАНИЕ СВОЙСТВ
        Masha.Passport();}
    }
}
```

Примечание – Вообще говоря, в данном случае логичнее использовать не свойство, а просто поле со спецификатором **readonly**. Свойство требуется для демонстрации синтаксиса.

Результат работы программы:



The image shows a screenshot of a Windows command prompt window. The title bar indicates the path 'C:\Windows\system32\cmd.exe'. The window contains the following text:

```
Monster Masha health = 200 ammo = 200  
Monster Masha health = 199 ammo = 300  
Для продолжения нажмите любую клавишу . . . _
```

Рекомендации по программированию

При создании класса, то есть нового типа данных, следует хорошо продумать его **интерфейс** – средства работы с классом, доступные использующим его программистам. Интерфейс хорошо спроектированного класса интуитивно ясен, непротиворечив и обозрим. Как правило, он не должен включать поля данных.

Поля предпочтительнее делать **закрытыми** (private). Это дает возможность впоследствии изменить реализацию класса без изменений в его интерфейсе, а также регулировать доступ к полям класса с помощью набора предоставляемых пользователю свойств и методов. Важно помнить, что поля класса вводятся только для того, чтобы **реализовать характеристики класса, представленные в его интерфейсе с помощью свойств и методов**.

Не нужно расширять интерфейс класса без необходимости, «на всякий случай», поскольку **увеличение количества методов затрудняет понимание класса пользователем** (под пользователем имеется в виду программист, применяющий класс).

В идеале **интерфейс должен быть полным**, то есть:

- предоставлять возможность выполнять любые разумные действия с классом, и одновременно
- минимально необходимым – без дублирования и пересечения возможностей методов.

Методы определяют *поведение класса*.

Каждый метод класса должен решать только *одну задачу* (не надо объединять два коротких независимых фрагмента кода в один метод). Размер метода может варьироваться в широких пределах, все зависит от того, какие функции он выполняет.

Желательно, чтобы тело метода помещалось на 1-2 экрана: одинаково сложно разбираться в программе, содержащей несколько необъятных функций, и в россыпи из сотен единиц по несколько строк каждая.

Если метод реализует *сложные действия*, следует разбить его на последовательность шагов, и каждый шаг оформить в виде вспомогательной функции (метода со спецификатором `private`).

Если некоторые *действия встречаются в коде хотя бы дважды*, их также нужно оформить в виде отдельной функции.

Создание любой функции следует начинать с ее интерфейса, то есть заголовка.

Необходимо четко представлять себе:

- какие параметры функция должна *получать*;
- какие результаты *формировать*.

Входные параметры обычно перечисляют в начале *списка параметров*.

Предпочтительнее, чтобы каждая функция вычисляла ровно один результат, однако это не всегда оправдано.

Если величина вычисляется внутри функции и возвращается из нее через список параметров, необходимо использовать перед соответствующим параметром ключевое слово `out`.

Если параметр значимого типа может изменить свою величину внутри функции, его предваряют ключевым словом `ref`. Величины ссылочного типа всегда передаются по адресу и, следовательно, могут изменить внутри функции свое значение.

Необходимо стремиться к максимальному сокращению области действия каждой переменной, то есть к *реализации принципа инкапсуляции*. Это упрощает отладку программы, поскольку ограничивает область поиска ошибки. Следовательно, величины, используемые только в функции, следует описывать внутри нее как локальные переменные.

Поля, характеризующие класс в целом, то есть имеющие одно и то же значение для всех экземпляров, следует описывать как *статические*.

Все литералы, связанные с классом (числовые и строковые константы), описываются как поля-константы с именами, отражающими их смысл.