Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

C#

Объектно-ориентированный язык программирования

Пособие (дополнение 1) к практическим занятиям - №2

Проф. Забудский Е.И.

Москва 2006

Тема: КОНСТРУКТОРЫ и СВОЙСТВА в C#

Два практических занятия
(4 часа)

Конструкторы создают новые объекты и присваивают значения (см. Листинги: **1…5**).

Посредством свойств осуществляется доступ к закрытым переменным (см. Листинги: **6…9**).

// **КОММЕНТАРИЙ**. На сайте http://www.firststeps.ru/ “Первые шаги” представлено много интересных обучающих материалов по различным интегрированным средам и языкам программирования, в том числе представлены C# и платформа .NET (step by step).

http://ooad.asf.ru/ – **ПОСЕТИТЕ ЭТОТ САЙТ, он посвящен объектно-ориентированному анализу, проектированию и программированию**

Содержание

1. Создание объектов и присвоение значений: КОНСТРУКТОРЫ

В объектно-ориентированном языке **C#** четко различаются понятия «класс» и «объект». **Класс** можно представить себе как **шаблон** (чертеж), по которому организуются его члены, то есть объекты. Таким образом, **объект** – это конкретный **экземпляр класса**. Объект обычно выполняет действия в программе.

1.1. Создание объекта

Единственный способ создания нового объекта в **C#** — использовать ключевое слово **new**. Рассмотрим, как создаются объекты, на следующем примере (**Листинг 1**):

Листинг 1. Троелсен, с.78 **файл Troelsen78.cs**

```
01: using System;
02:
03: namespace ConsAppl_Troelsen78
04: {
05:     // Объекты HelloClass будут созданы при вызове метода Main()
06:
07:     class HelloClass
08:     {
09:         public static int Main(string [ ] args)
10:         {
11:     // Новый объект можно создать: а) в одной строке
12:             HelloClass c1 = new HelloClass();
13:                                 // б) ... или в двух
14:             HelloClass c2 ;
15:             c2 = new HelloClass();
16:             Console.ReadLine(); return 0;
17:         }
18:     }
19: }
```

Ключевое слово **new** означает, что среде выполнения следует выделить необходимое количество оперативной памяти под экземпляр создаваемого объекта. В примере в оперативной памяти созданы два объекта **c1** и **c2**, каждый из которых является экземпляром типа **HelloClass**. В C# переменные класса — это ссылки на объект в памяти, но не сами объекты. Поэтому **c1 и c2** – это ссылки на два отдельных объекта, находящиеся в оперативной памяти. Подробнее о ссылочных типах см. Материалы к практическому занятию №4, стр. 7, сл..

1. 2. Создание объектов и присвоение значений

Объекты создаются:

1) при помощи конструктора по умолчанию (такой конструктор не имеет параметров);

2) при помощи конструктора с параметрами.

Компилятор **C#** автоматически снабжает конструктором по умолчанию любой класс.

В C# конструктор по умолчанию присваивает всем переменным экземпляра (то есть исходным данным, характеризующим объект) значения по умолчанию — в большинстве случаев **0** (в ряде других случаев – **null**).

Как правило, класс снабжается не только конструктором по умолчанию, но и другими конструкторами – принимающими параметры. Использование такого конструктора – самый простой способ присвоить нужные значения всем переменным экземпляра (то есть инициализировать состояние объекта **в момент его создания**).

В качестве второго примера еще раз обратимся к классу **HelloClass**. Придадим этому классу: **1)** несколько переменных экземпляра, значения которых можно устанавливать, **2)** конструктор с параметрами, **3)** а также переопределим конструктор по умолчанию — конструктор без параметров.

Листинг 2. Троелсен, с.79 файл Troelsen79.cs

```
01: using System;
02:
03: namespace ConsAppl_Troelsen79
04: {
05:
06:     public class HelloClass                          // класс HelloClass с конструкторами
07:     {
08:
09:         public HelloClass()  // Это конст-р по умол-ю, он присвоит перем-м знач-я по умол-ю
10:         {                          // Имя конструктора объектов класса совпадает с именем класса
11: Console.WriteLine("Был вызван встроенный конструктор (по умолчанию; БЕЗ ПАРАМЕТРОВ)!");
12:         }
13:                     // Собственный конст-р (с параметрами) присвоит перем-м заданные знач-я
14:         public HelloClass(int x, int y)
15:         {
16: Console.WriteLine("Был вызван собственный конструктор (С ПАРАМЕТРАМИ)!");
17:             intX = x;
18:             intY = y;
19:         }
20:                                                  // Объявляем переменные-члены класса
21:         public int intX, intY;
22:         // Точка входа для программы
23:         public static int Main(string [ ] args)
24:         {
25:                     // Применяем конструктор по умолчанию (c1 – объект класса HelloClass)
26:         HelloClass c1 = new HelloClass();
27:         Console.WriteLine("c1.intX = {0}\nc1.intY = {1}\n", c1.intX, c1.intY);
28:                     // Применяем конструктор с параметрами  (c2 – объект класса HelloClass)
29:         HelloClass c2 = new HelloClass(100, 200);
30:         Console.WriteLine("c2.intX = {0}\nc2.intY = {1}\n", c2.intX, c2.intY);
31:             Console.ReadLine();
32:             return 0;
33:         }
34:     }
35: }
```

Запустив эту программу, можно убедиться, что конструктор по умолчанию действительно присвоил переменным-членам значения по умолчанию (нулевые), а конструктор с параметрами присвоил переменным-членам значения, соответствующие этим параметрам.

Результаты работы программы

Был вызван встроенный конструктор (по умолчанию; без параметров)!
c1.intX = **0**
c1.intY = **0**
Был вызван собственный конструктор (с параметрами)!
c2.intX = **100**
c2.intY = **200**

2. Подробнее о конструкторах

Конструктор используется:

1) для создания объектов:
2) для присвоения значения объектам.

В объектно-ориентированном языке программирования C# конструкторы представляют собой ничего не возвращающие методы, имя которых совпадает с именем класса.

Если включить в код конструктора функцию возврата какого-либо значения, компилятор не будет считать такой метод конструктором.

2.1. Когда вызывается конструктор?

Конструктор вызывается при создании нового объекта. Рассмотрим следующий код:

```
Shape Circle = new Shape();
```

Ключевое слово **new** означает, что сначала создается новый экземпляр класса Shape (это объект Circle), при этом под него выделяется необходимая область памяти. Затем вызывается конструктор со списком необходимых аргументов для его параметров. Необходимо предусматривать необходимые инициализирующие действия в рамках конструктора.

Таким образом, код **new** Shape() создает объект класса Shape (это объект Circle) и вызывает метод Shape(), который является конструктором этого класса.

2.2. Что находится внутри конструктора?

Наиболее важной функцией конструктора является инициализация памяти, выделенной под объект в результате использования ключевого слова **new**. Заключенный внутри конструктора код должен установить только что созданный объект в начальное, устойчивое, безопасное состояние (то есть конструктор должен присвоить объекту соответствующее значение).

2.3. Включение в класс конструктора

Лучше всегда включать в класс конструктор, даже если не планируется выполнять в нем какие-либо действия. Можно включить в класс пустой конструктор и добавлять в него код по мере необходимости.

2.4. Использование нескольких конструкторов

Во многих случаях объект может быть создан более чем одним способом. В такой ситуации необходимо включить в класс несколько конструкторов. Рассмотрим, например класс Count:

```
public class Count
{ int num_count;
    public Count()
    {
        num_count = 0;
     }
 }
```

С одной стороны, нужно присвоить атрибуту num_count нулевое значение. Это легко сделать при помощи следующего конструктора (без параметров):

```
public Count()                  // сигнатура конструктора – Count()
{
num_count = 0;
}        .
```

С другой стороны, может понадобиться добавить в конструктор некоторый параметр, который позволит присваивать атрибуту num_count различные значения:

```
public Count (int number)     // сигнатура конструктора – Count(int number)
{
num_count = number;
 }
```

Это называется перегрузкой конструктора (конструктор – есть метод).

2.5. Перегрузка методов (в том числе конструкторов) ,

Перегрузка позволяет использовать одноименные методы до тех пор, пока их сигнатуры отличаются друг от друга . Сигнатура метода включает в себя: 1) его имя и 2) список параметров:

```
public String getRecor(int kay) .        // заголовок метода
```

Сигнатура метода: 1) имя метода; 2) список параметров

Например, *все* представленные ниже методы имеют различные сигнатуры:

```
public void getCab();  .                        // нет параметров
public void getCab (String cabbieName);         // другой список параметров
public void getCab (int numberOfPassengers);  // другой список параметров
```

Использование в одном классе следующих методов вызовет конфликт, так как у этих методов одинаковые сигнатуры (конфликт возникнет, несмотря на то, что типы возвращаемых методами значений отличаются друг от друга).

```
public void getCab (String cabbieName);
public  int   getCab (String cabbieName);
```

Используя различные сигнатуры, можно создавать различные объекты в зависимости от используемого конструктора.

2.6. Указание типа возвращаемых данных

Тип возвращаемых данных должен указываться для всех методов за исключением конструкторов.

2.7. Присвоение значений с помощью конструктора

Приведенный ниже способ присвоения значений переменным (occupants, area, floors) экземпляра (house) возможен:

```
house.occupants  =  4;
house.area  =  2500;
house.floors  =  2;
```

Однако более удобный способ это сделать – использовать конструктор.

2.7.1. Конструктор без параметров

Конструктор инициализирует объект при его создании. Он имеет такое же имя, что и сам класс, а синтаксически подобен методу. Однако в определении конструкторов не указывается тип возвращаемого значения. Формат записи конструктора такой:

```
доступ имя_класса ()
{
// тело конструктора
)
```

Обычно конструктор используется чтобы: 1) придать переменным экземпляра, определенным в классе, начальные значения или 2) выполнить исходные действия, необходимые для создания полностью сформированного объекта. Кроме того, обычно в качестве элемента «доступ» используется модификатор доступа **public**, поскольку конструкторы, как правило, вызываются вне их класса.

Вот пример использования простого конструктора (без параметров):

Листинг 3. Шилдт, с.145 файл Schildt_145.cs

```
01: // Использование простого конструктора.
02: using System;
03:     class MyClass
04:     {
05:     public int x;
06:             public MyClass ()   // простой конструктор (без параметров)
07:             {
08:                     x = 10;
09:             }
10:     }
11:     class ConsDemo
12:     {
13:             public static void Main()
14:             {
15:             MyClass c1  = new MyClass();
16:             MyClass c2 = new MyClass();
17:             Console.WriteLine(c1.x + "  " + c2.x);
18:             }
19:     }
```

В этом примере конструктор класса **MyClass** имеет следующий вид:

```
06:             public MyClass ()   // простой конструктор (без параметров)
07:             {
08:                     x = 10;
09:             }
```

Обратите внимание на **public**-определение конструктора, которое позволяет вызывать его из кода, определенного вне класса **MyClass**. Этот конструктор присваивает переменной экземп-

ляра **x** значение **10**. Конструктор **MyClass()** вызывается оператором **new** при создании объекта класса **MyClass**. Например, при выполнении строки

```
MyClass  c1  = new MyClass();
```

для объекта **c1** вызывается конструктор **MyClass()**, который присваивает переменной экземпляра **c1.x** значение **10**. То же самое справедливо и в отношении объекта **c2**, т.е. в результате создания объекта **c2** значение переменной экземпляра **c2.x** также станет равным **10**. Таким образом, после выполнения этой программы получаем следующий результат**:** **10 10**

2.7.2. Конструкторы с параметрами

В предыдущем примере использовался конструктор без параметров. Но чаще приходится иметь дело с конструкторами, которые принимают один или несколько параметров. Параметры вносятся в конструктор точно так же, как в метод**:** для этого достаточно объявить их внутри круглых скобок после имени конструктора.

Вот пример использования конструктора с параметрами:

Листинг 4. Шилдт, с.146 файл Schildt_146.cs

```
01: // Использование  конструктора с параметрами
02: using System;
03:     class MyClass
04:     {
05:     public  int  x;
06:             public MyClass (int param)   //  конструктор с параметрами
07:             {
08:                     x =  param;
09:             }
10:     }
11:     class  ParamConsDemo
12:     {
13:             public  static void Main()
14:             {
15:             MyClass  c1   =  new MyClass(15);
16:             MyClass  c2  =   new MyClass(30);
17:             Console.WriteLine(c1.x  +  "   "  +  c2.x);
18:             }
19:     }
```

После выполнения этой программы получаем следующий результат**:** **15 30**

В конструкторе **MyClass()** этой версии программы определен один параметре с именем **param**, который используется для инициализации переменной экземпляра **x**. Таким образом, при выполнении строки кода

```
MyClass  c1  = new MyClass(15);
```

параметру **param** передается значение **15**, которое затем присваивается переменной экземпляра

c1.x. А при выполнении строки кода

MyClass **c2** = new MyClass(**30**);

параметру **param** передается значение **30**, которое затем присваивается переменной экземпляра **c2.x**.

3. Добавление конструктора в класс Building

Добавим в класс Building конструктор, который при создании объекта автоматически инициализирует поля (т.е. переменные экземпляра) floors, area и occupants. Обратите особое внимание на то, как создаются объекты класса Building.

Листинг 5. Шилдт, с.147 файл Schildt_147.cs

```
01:                    // Добавление конструктора в класс Building.
02: using System;
03: class Building
04: {
05: public int floors;        //  количество этажей
06: public int area;          //  общая площадь основания здания
07: public int occupants;     //  количество жильцов
08:     public Building(int f,  int a,  int o)  // конструктор с параметрами
09:     {
10:     floors = f ;
11:     area = a ;
12:     occupants = o ;
13:     }
14:             public int areaPerPerson( )
15:             {               // Этот метод возвр-ет значение площади, кот-я прих-ся на одного чел-ка.
16:             return area / occupants;
17:              }
18:                         /*  А этот метод возвр-ет макс-ное возм-ное кол-во человек в здании,  если
19:                               на каждого должна приходиться заданная минимальная площадь.  */
20:     public int maxOccupant(int minArea)
21:     {
22:     return area / minArea;
23:     }
24: }
25: class BuildingDemo
26: {
27:     public static void Main()
28:     {                   // Дважды используется конструктор Building( ) (с разными параметрами)
29:     Building house = new Building(2,  2500,  4);  // объект house
30:     Building office = new Building(3,  4200,  25);  // объект office
```

```
31:     Console.WriteLine("Максимальное число человек для дома,  \n" +
32:                       "если на  каждого должно приходиться  "  +
33:                       30  +  "  квадратных  метров:  "  +
34:                       house.maxOccupant(30)) ;
35:     Console.WriteLine("Максимальное число человек для офиса,  \n" +
36:                       "если на  каждого должно приходиться  " +
37:                       30  +  "  квадратных метров:  "  +
38:                       office.maxOccupant(30));
39:     }
40: }
```

Результаты выполнения этой программы:

```
Максимальное число человек для дома,
если на  каждого должно приходиться  30  квадратных  метров:  83  = [ 2500 / 30 ]
Максимальное число человек для офиса,
если на  каждого должно приходиться  30  квадратных метров:  140  = [ 4200 / 30 ]
```

Оба объекта, **house** и **office**, в момент создания инициализируются в программе конструктором **Building()**. Каждый объект инициализируется в соответствии с тем, как заданы параметры, передаваемые конструктору. Например, при выполнении строки

```
Building house = new Building(2,  2500,  4);
```

конструктору **Building()** передаются значения **2, 2500** и **4** в момент, когда оператор **new** создает объект класса **Building** (имя этого объекта house). В результате этого копии переменных floors, area и occupants, принадлежащие объекту house, будут содержать значения **2**, **2500** и **4**, соответственно.

(см. также Материалы к Практ. Зан. № 3 C#, стр. 21)

4. Доступ к закрытым переменным: СВОЙСТВА

Свойство (properties) используется для обращения к закрытым членам класса.

Свойство включает: 1) поле[1] и 2) методы доступа к этому полю.

Свойство состоит: 1) из имени и 2) двух аксессоров (**get** и **set** – это внутренние методы свойства).

Аксессоры (accessor – средство доступа) используются: 1) для чтения (**get**) содержимого переменной и 2) для записи (**set**) в нее нового значения.

Основное достоинство свойства состоит в том, что его имя можно использовать в выражениях и инструкциях присваивания подобно обычной переменной, хотя в действительности здесь будут автоматически вызываться **get**- и **set**-аксессоры (см. далее стр. **28…34**).

Формат записи свойства таков:

```
тип  идентификатор
{
   get                          // get (получить)– этот метод получает  значения данных
    {
     // код аксессора чтения поля
    }
  set                           // set (задать ) – этот метод изменяет значения данных
  {
    // код аксессора записи поля
  }
}
```

Здесь **тип** – это **тип** свойства (например, **int**), а **идентификатор** – его **имя**. После определения свойства любое использование его имени означает вызов соответствующего аксессора. Аксессор **set** автоматически принимает параметр с именем **value**, который содержит значение, присваиваемое свойству.

Важно понимать, что свойства не определяют область памяти. Следовательно, свойство управляет доступом к полю, но самого поля не обеспечивает. Это поле должно быть задано независимо от свойства (см. далее стр. **07**).

4.1. Свойства – пример 1

Рассмотрим пример, в котором определяется свойство **Prop**, используемое для доступа к полю **prop**. Это свойство (в данном примере) позволяет присваивать полю только положительные числа.

Листинг 6. Файл **Schildt_267_properties.cs**

```
01: using System;
02: //  Пример использования свойства.
03: namespace ConsAppl_Schildt_267_properties
04: {
05:  class SimpProp
06:   {
07:      private int prop;                          //  Это поле управляется свойством Prop.
```

[1] Поля, то есть переменные экземпляра

```
08:     public SimpProp() { prop = 0; }              // Конструктор инициализирует переменную prop
09:           /*  Это  свойство (его имя Prop) поддер-ет доступ  к ЗАКРЫТОЙ переменной экземпляра
              prop. В данном контексте св-во Prop позволяет присваивать  ей только полож-е числа. */
10:      public int Prop
11:     {
12:        get
13:        {
14:           return prop;
15:        }
16:        set
17:         {
18:           if (value >= 0) prop = value;
19:         }
20:     }
21:  }
22:                                                  // Демонстрируем использование  свойства Prop
23:   class PropertyDemo
24:   {
25:     public static void Main()
26:     {
27:        SimpProp ob = new SimpProp();          // Конструктор создает объект ob класса SimpProp
28:       Console.WriteLine("Исходное  значение  ob.Prop:   "  + ob.Prop);
29:        ob.Prop =  100;                              //  Присваиваем значение.
30:        Console.WriteLine("Значение  ob.Prop:   "  +  ob.Prop);
31:                              //  Переменной prop невозможно присвоить  отрицательное  значение.
32:     Console.WriteLine("Попытка  присвоить -10 свойству ob.Prop");
33:     ob.Prop  =  -10;
34:        Console.WriteLine("Значение  ob.Prop:   " + ob.Prop);
35:     Console.ReadLine();
36:     }
37:   }
38: }
```

Результаты выполнения этой программы выглядят так:

```
Исходное  значение  ob.Prop:   0
Значение  ob.Prop:   100
Попытка присвоить  -10  свойству ob.Prop
Значение ob.Prop::   100
```

В классе **SimpProp** определяется закрытое поле **prop** (строка **07**) и свойство **Prop** (строки **10…20**), которое управляет доступом к закрытому полю **prop**. Свойство само не определяет область хранения поля, а лишь управляет доступом к нему. Поэтому без определения базового поля (в строке **07**) определение свойства теряет всякий смысл. Более того, поскольку поле **prop** закрытое, к нему можно получить доступ только посредством свойства **Prop**.

Свойство **Prop** определяется как **public**-член класса (строка **10**), чтобы к нему можно было обратиться с помощью кода (строки **28…34**) вне класса, включающего это свойство. В этом есть своя логика, поскольку свойство предоставляет доступ к закрытому полю **prop** с помощью аксессоров**: get**-аксессор просто возвращает значение **prop**, а **set**-аксессор устанавливает новое значение **prop**, если оно положительно. Таким образом, свойство **Prop** управляет тем, какие значения может содержать поле **prop**. В этом и состоит важность свойств.

Свойство **Prop** предназначено для чтения и записи, поскольку позволяет как прочитать содержимое своего базового поля, так и записать в него новое значение. Но можно создавать свойства, предназначенные только для чтения (определив лишь **get**-аксессор), либо только для записи (определив лишь **set**-аксессор).

4.2. Свойства – пример 2

Методы доступа к свойствам вызываются: **1)** для чтения или **2)** записи значения свойства. Метод для чтения значения свойства обозначается ключевым словом **get**, а метод для изменения значения свойства обозначается **set**.

В примере, приведенном на листинге 7, для свойства **SquareFeet** реализованы методы доступа **get** и **set**.

Листинг 7. Реализация методов доступа к свойствам. Файл **Wil_63.cs**

```
01: using System;                    // Свойства класса.  Реализация методов доступа к свойствам
02: namespace ConsAppl_Wil_63
03: {
04:    public class House
05:    {
06:       private int squareMetre;                    // переменная экземпляра squareMetre
07:       public int SquareMetre                      // свойство SquareMetre
08:       {
09:          get { return squareMetre; }              // получить значение
10:          set { squareMetre = value; }             // задать значение   (150)
11:       }
12:    }
13:   class TestApp
14:   {
15:      public static void Main()
16:      {
17:         House myHouse = new House();     // конструктор создает объект myHouse класса House
18:          myHouse.SquareMetre = 150;      // 1-е обращение к свойству SquareMetre: метод set
19: Console.WriteLine(" Площадь дома {0} кв.м", myHouse.SquareMetre); // 2-е обр-ние:  м-д get
20:          Console.ReadLine();
21 :      }
22 :   }
23: }
```

Результаты выполнения этой программы выглядят так: Площадь дома **150** кв.м

Класс **House** имеет всего одно свойство - **SquareFeet** (площадь), которое может: **1)** считываться и **2)** записываться (строки **07...11**). Само значение представлено в переменной **squareMetre** (строки **06**), доступной только внутри класса **House**. И если вы хотите записать его в виде поля, все, что вам понадобится для этого сделать, – убрать методы доступа к свойству и переопределить переменную как публичную:

```
public int squareMetre;   // совет : поэкспериментируйте
```

Для такой простой переменной это вполне допустимо. Но, например, если вы хотите скрыть детали внутреннего устройства вашего класса, следует использовать методы доступа к свойствам. В этом случае новое значение переменной передается в качестве параметра **value** метода **set** (этот параметр не может иметь другое имя, см. стр. **10**).

Вы можете не только скрыть детали реализации, но также определить, какие из следующих операций будут разрешены:

1) реализованы оба метода: разрешено **чтение** (**get**) и **запись** (**set**) свойства;

2) реализован только метод **get:** разрешено **чтение** свойства;

3) реализован только метод **set:** разрешена **запись**.

Кроме того, вы можете реализовать в методе **set** проверку значения (см. строку **18** в программе Листинга 6). Например, вы можете по какой-то причине отвергнуть новое значение свойства. Свойство особенно замечательно тем, что может быть динамическим, то есть проявляться только при первом запросе и не занимать до этого ресурсы.

4.3. Свойства – пример 3

Свойства можно понимать как интеллектуальные поля, поддерживающие инкапсуляцию типа (см. Материалы к Практ. Зан. №3, стр. 4, сл.). С точки зрения практического применения свойство доступно так же, как открытое поле в типе. Единственным ограничением является то, что свойство не может передаваться в метод как параметр.

Листинг 8. Файл **Mayo_67_properties.cs**

```
01: using System;
02: namespace ConsAppl_Mayo_67_properties
03: {
04:   class Properties
05:   {                    // объявлена переменная myMoney, которой управляет свойство MyMoney
06:     private static decimal myMoney;
07:     public static decimal MyMoney                    // свойство MyMoney
08:     {
09:       get                                  // возврат (получение) значения переменной myMoney
10:       {
11:       return myMoney;
12:       }
13:       set                                  // установить (задать) значение переменной myMoney
14:       {
15:        myMoney = value;              // имя “value” задано компилятором, его нельзя изменить
16:       }
17:     }
18:     static void Main()
19:     {
20:     MyMoney = 1000000.00m;                          // слева от знака "равно" имя свойства - NB
21:     Console.WriteLine("myMoney: {0:$#,###.00}", MyMoney); // в списке вывода имя свойства
22:     Console.ReadLine();
23:     }
24:   }
25: }
```

Результаты выполнения этой программы выглядят так: myMoney: $**1 000 000,00**

В большинстве случаев свойства объявляются так, как объявлено свойство **MyMoney** в программе. Свойство без функции **get** доступно только для записи, а свойство без функции **set** доступно только для чтения. В строке **20** – первое обращение к свойству **MyMoney,** в результате работает метод **set** и переменной **myMoney** присвоено значение. В строке **21** – второе обращение к свойству **MyMoney,** в результате работает метод **get** и получено значение переменной **myMoney**.

4.4. Свойства – пример 4

Хотя **свойство** для внешнего мира представляет собой переменную экземпляра **public**, оно реализовано способом, похожим на **метод** (ведь свойство тоже может исполнять операторы).

Свойство (как структурный элемент языка **C#**) позволяет:

а) внешнему пользователю объекта обращаться к переменной экземпляра,

б) переменным экземпляра оставаться закрытыми. В то же время, свойство может содержать операторы.

Листинг 9. Файл **Mic_544_properties.cs**

```
01: using System;
02: namespace ConsAppl_Mic_544_properties
03: {
04:   class Bicycle
05:   {
06:     const byte MaxSpeed = 40;      // максимальное значение скорости (ограничение) – 40 км/час
07:     private byte speed = 0;                  // значение скорости
08:     private int speedAccessCounter = 0;  // счетчик кол-ва обращений к методу get св-ва Speed
09:     public byte Speed                        // первое свойство Speed
10:     {
11:       get
12:       {
13:         speedAccessCounter++;
14:         return speed;
15:       }
16:       set
17:       {
18:           if (value > MaxSpeed)
19:     Console.WriteLine("Ошибка. {0} превышает ограничение скорости {1}", value, MaxSpeed);
20:           else if (value < 0)
21:         Console.WriteLine("Ошибка. {0} менее чем 0", value);
22:           else
23:             speed = value;
24:       }
25:     }
26:     public int SpeedAccessCounter                  // второе свойство SpeedAccessCounter
27:     {
28:       get
29:       {
30:         return speedAccessCounter;
31:       }
32:     }
33:   }
34:   class BicycleTester
35:   {
36:     public static void Main()
37:     {
38:       byte speedInMilesPerHour;                              // скорость в милях за час
39:       Bicycle myBike = new Bicycle();              // создан объект myBike класса Bicycle
40:       myBike.Speed = 60;  // первое обращение к методу set свойства Speed – 60 км/час
41:       myBike.Speed = 30;  // второе обращение к методу set свойства Speed  – 30 км/час
```

```
42:          // первое обращение к методу get свойства Speed
43:          Console.WriteLine("Текущая скорость объекта myBike: {0} км/час", myBike.Speed);
44:       // второе обращение к методу get свойства Speed
45:          speedInMilesPerHour = (byte)(myBike.Speed * 0.621);
46:              Console.WriteLine("Текущая скорость объекта myBike: {0} миль/час",
47:                                  speedInMilesPerHour);
48:          // обращение к методу get свойства SpeedAccessCounter
49:          Console.WriteLine("Количество определений скорости велосипеда: {0}",
50:                                  myBike.SpeedAccessCounter);
51:          Console.ReadLine();
52:       }
53:    }
54: }
```

Результаты работы программы

```
Ошибка. 60 превышает ограничение скорости 40
Текущая скорость объекта myBike: 30 км/час                    (это значение speed)
Текущая скорость объекта myBike: 18 миль/час                  (это значение speedInMilesPerHour)
Количество определений скорости велосипеда: 2                 (это значение speedAccessCounter)
```

Класс **Bicycle** содержит два свойства. Одно из них – **Speed** (строки **9…27**) – обеспечивает доступ к переменной экземпляра **speed**, объявленной в строке **7**. Другое – **SpeedAccessorCounter** (строки **26…32**) – предназначено для доступа к **speedAccessCounter**. Рассмотрим вначале свойство **Speed**. Свойство состоит из заголовка (строка **9**) и блока (строки **10…25**), заключенного в фигурные скобки (строки **10** и **25**). В заголовке можно определить спецификатор доступности (в данном случае **public**), тип свойства (**byte**) и его имя (**Speed**). Блок свойства состоит из блока **get** (строки **11…15**) и блока **set** (строки **16…24**).

Пользователь класса **Bicycle**, в данном случае это метод **Main,** может теперь вызвать свойство **Speed** как собственную переменную экземпляра. Это показано в строках **40…45**. Когда свойству **Speed** присваивается значение (например, в строке **41**):

```
myBike.Speed = 30;
```

исполняются операторы блока **set**. Специальный параметр **value**, встречающийся в строках **18**, **19, 20**, **21** и **23**, представляет присваиваемое значение (в данном случае **30**). Среда исполнения автоматически присваивает параметру **value** значение **30**. Таким образом, в строке **23** значение **30** присваивается переменной экземпляра **speed**. Чтение значения **Speed** (в строке **45**)

```
speedInMilesPerHour = (byte)(myBike.Speed *0.621);
```

1 миля ≈ 1610 м

инициирует исполнение операторов блока **get**. После исполнения оператора с ключевым словом **return** (строка **14**) блок **get** завершается и возвращает выражение, следующее за словом **return**. Значение **return** в таком контексте идентично его значению в обычном методе.

Если требуется только установить значение переменной экземпляра (когда она предназначена только для записи) или только прочесть его (переменная только для чтения), можно определить только один блок: **set** или **get**. Важно, что хотя бы один из них должен присутствовать в свойстве. К примеру, свойство **SpeedAccessCounter** (строки **26…32**) предназначено только для чтения: в нем опущен блок **set**.

П р и м е ч а н и я

1. Тип переменной экземпляра, расположенной после ключевого слова **return** в блоке **get** (строки **14 и 07**)

должен совпадать с типом в заголовке свойства (строка **09**); в программе – это тип **byte** .

2. Параметр **value** в блоке_**set** (строки **16…24**) представляет значение, присваиваемое свойству, и принадлежит типу из заголовка свойства (строка **09**); в программе – это тип **byte**. Имя **value** задано компилятором – его нельзя изменить.

3. Свойство может содержать только блок_**get**, только блок_**set** или оба блока.

4. Зачастую свойство представляет определенную переменную экземпляра (хотя это не обязательно).

5. Клиенту (клиентом называется часть программы, которая вызывает свойства и методы класса или объекта) удобно использовать свойства по следующим причинам.

5.1. Свойства обеспечивают согласованный синтаксис при обращении к переменной экземпляра внутри объекта и за его пределами.

5.2. Свойства позволяют программисту не заботиться о том, является переменная экземпляра **public** или **private**.

6. Рекомендуется стиль написания идентификатора **С**войств - **Pascal**,

7. Рекомендуется стиль написания идентификатора **п**ЕРЕМЕННЫХ **э**КЗЕМПЛЯРА –**вЕРБЛЮЖИЙ**

Резюме

Рассказано о создании объектов, в частности о конструкторах и их возможностях при создании объекта.

Ниже перечислены важные рассмотренные моменты.

Переменная экземпляра может быть инициализирована тремя способами.

1. Автоматически — средой исполнения. В этом случае переменной экземпляра присваивается значение по умолчанию, которое зависит от ее типа.
2. С использованием объявления инициализации, что позволяет программисту задавать нужное значение.
3. С использованием конструктора – функции-члена, исполняющейся при создании объекта.

Часто для выполнения инициализации, где требуется запуск нескольких операторов, применяются конструкторы. Их можно использовать и для других действий при создании объекта.

Конструктор экземпляра вызывается тогда, когда для создания объекта применяется ключевое слово **new**.

Если в классе не определен ни один конструктор, компилятор автоматически включает конструктор по умолчанию. Последний не имеет аргументов.

В один и тот же класс можно включить несколько конструкторов. В этом случае они становятся перегруженными подобно методам. Формальные параметры перегруженных конструкторов (как и перегруженных методов) различаются по типу или количеству.

Также рассмотрены функции-члены классов – **свойства**.

Для получения доступа к переменным экземпляра лучше использовать свойства, а не аксессор и мутатор. Обращение к свойству происходит так же, как к открытой переменной экземпляра, однако при чтении или присваивании значения исполняются блоки **get** и **set**, соответственно. Свойство может не только представлять какую-то переменную экземпляра, но и возвращать вычисленное значение.

Свойства исполняются так же быстро, как и аксессор и мутатор, поскольку при компиляции в **MSIL** как свойств, так и методов, получается похожий код. При определенных обстоятельствах свойства могут исполняться с той же скоростью, что непосредственный доступ к переменной экземпляра. Это достигается благодаря подстановочной оптимизации.

Если переменная экземпляра — ресурс, требующий инициализации и обновления, можно использовать свойства вместе с двумя подходами: 1) отложенной инициализацией и 2) медленным обновлением. Они позволяют сократить потребление ресурсов процессора и памяти.

Контрольные вопросы

1. а) Если **Speed** — свойство, определенное в объекте **myRocket**, каково общее название блока операторов в свойстве **Speed**, который запускается при исполнении строки:

myRocket.Speed = **40;**

б) при исполнении строки:

travelTime = distance/**myRocket.Speed;**

2. Какой из стилей капитализации рекомендуется применять для**:**

а) переменных экземпляра?

б) методов?

в) свойств?

3. Ниже приведено объявление переменной экземпляра **speed** и нечто, задуманное как свойство для доступа к ней извне объекта. Однако чтобы свойство работало, необходимо устранить четыре ошибки.

```
private double speed;
private int Speed()
{ get
        {
        return Speed;
         }
}
```

Задача по программированию

1. Расширьте класс **Bicycle** (листинг 9), чтобы он содержал переменную экземпляра **age**, представляющую срок службы объекта **Bicycle**. Создайте свойство, которое позволит устанавливать (**set**) и читать (**get**) ее значение. Включите код, предотвращающий присваивание **age** значения отрицательного или большего **200**. Включите в класс **Bicycle** переменную экземпляра **numberOfAgeAccesses**, хранящую число обращений к переменной **age**. Создайте свойство для чтения **numberOfAgeAccesses**. Должно ли оно содержать оба блока, **set** и **get**?

Рекомедуемая литература

Мэтт Вайсфельд. Объектно-ориентированный подход**:** Java**, .**Net**,** C++ . Второе издание / Пер. с англ. - М: КУДИЦ-ОБРАЗ, 2005. - 336 с.

Что значит освоить объектно-ориентированное программирование? Для этого недостаточно выучить синтаксис языка C#, Java или C++. Нужно разобраться в принципиальных положениях объектного подхода, понять, чем он отличается от других. И предлагаемая книга будет в этом отличным помощником. В ней на конкретных примерах разбираются все основные понятия объектно-ориентированного подхода. Советую прочесть эту книгу.

Анатомия класса

Класс является для объекта тем же, чем строительный чертеж для дома. Класс — это абстракция (реальная или концептуальная) объекта, принадлежащего какой-либо предметной области. Один шаблон класса можно использовать для создания нескольких объектов (экземпляров класса), которые обладают свойствами, определенными в классе.

При решении разных вычислительных задач объекты различных классов взаимодействуют друг с другом, внося свои уникальные свойства в общую программу. Конструкция класса позволяет объединять данные (называемые состоянием объекта) с функциями (представляющими его поведение) для создания объектов, составляющих структуру разрабатываемого ПО. Классы, в простейшем варианте, состоят из переменных и методов экземпляра (см. Материалы к Практ. Зан. №1, рис. 1.5).

Элементы класса являются языковыми конструкциями, составляющими тело класса, к примеру, переменные и методы экземпляра представляют собой два фундаментальных элемента класса. Однако классы настолько разнообразны, что **C#** содержит и несколько других элементов, придающих классу гибкость и расширяющих его возможности по взаимодействию с другими классами программы.

П.1. Обзор возможных элементов класса

В синтаксическом блоке **П.1** расширен синтаксис, показанный на рис. 1.5 (Практ. Зан. №1), и описаны элементы, которые можно включать в определение класса. В первых строках отображен уже знакомый синтаксис: 1) ключевое слово **class**, 2) имя (идентификатор) класса и 3) фигурные скобки, формирующие тело класса.

Элементы (члены) класса можно разделить на три широкие категории.

1. Данные-члены.
2. Функции-члены.
3. Вложенные типы.

Далее приведено их краткое описание.

П.1.1. Данные-члены состоят из **:**

1.1. Переменных-членов.

1.2. Констант.

1.3. Событий.

1.1. Переменные-члены (называемые также полями) используются для представления данных. Такая переменная может принадлежать**: а)** или конкретному экземпляру (объекту) класса — в этом случае она называется переменной экземпляра, **б)** или самому классу — тогда она называется статической (объявленной **static**) переменной (или переменной класса). Напомним, что статический метод принадлежит классу, а не объекту (и вызывается для класса). То же самое справедливо и для статической переменной.

Переменная-член может быть объявлена только для чтения (с ключевым словом **readonly**). Такие переменные тесно связаны с константами-членами (**1.2**), но существует важное различие — значения последних устанавливаются в программе в процессе компиляции и существуют на протяжении ее исполнения. А значения **readonly** переменных-членов присваиваются

им при создании объекта, и поэтому существуют, пока существует объект.

1.3. События объявляются подобно переменным-членам, но используются по-другому. При щелчке на кнопке (к примеру, ОК) в графическом пользовательском интерфейсе (**GUI**) соответствующий объект в программе генерирует (или возбуждает) событие (скажем, **OKButtonClick**), по которому определенная часть программы реагирует на действие пользователя. О приложении такого типа говорят, что оно управляется событиями, поскольку его следующее действие зависит от генерируемого события. Здесь уже неприменимо понятие программы, исполняющейся в той последовательности, в которой написаны ее операторы. Такая асинхронная способность используется в **GUI** (и других важных типах приложений), поскольку пользователь в любой момент может щелкнуть кнопкой мыши или нажать клавишу на клавиатуре.

СИНТАКСИЧЕСКИЙ БЛОК **П.1. Обзор элементов класса**

Определение_класса

```
class <Идентификатор_класса>
{
<Члены_класса>
}  ,
```

где
Члены_класса:

1. Данные-члены
2. Функции-члены
3. Вложенные_типы

Данные-члены

1.1. Переменные-члены
1.2. Константы .
1.3. События

Функции-члены

2.1. Методы
2.2. Конструкторы
2.3. Деструктор
2.4. Свойства
2.5. Индексаторы
2.6. Операции

Вложенные_типы

3.1. Вложенные_классы
3.2. Вложенные_структуры
3.3. Вложенные_перечисления

Примечания:

- В приведенном здесь определении класса основное место занимают его внутренние особенности, поэтому в нем опущены синтаксические элементы, связанные с модификацией доступа, наследованием и интерфейсами (см. Материалы к Практ. Зан. №**7**).
- Порядок элементов класса может быть любым внутри него, он не меняет семантики.

➢ <Функция-член> может быть: **а)** либо <Функция_экземпляра>, **б)** либо <Статическая_функция> (называемая также <Функция_класса>). Функция экземпляра всегда исполняется по отношению к конкретному объекту, поэтому последний необходим для ее вызова.

П.1.2. Функции-члены могут быть методами, конструкторами, деструкторами, свойствами, индексаторами и операциями:

2.1. Метод – знакомая конструкция, которая, тем не менее, обладает многими свойствами, как, например, ссылочные и выходные параметры, массивы параметров, перегрузка метода и ключевое слово **this**.

2.2. Конструкторы (рассмотрены выше).

2.3. Деструктор (называемый также завершающим методом). Объекты создаются и размещаются в определенной области памяти, где хранятся значения их переменных экземпляра и другие данные. Когда объект становится ненужным программе, занимаемую им память следует освободить для других объектов (иначе программа быстро начнет испытывать дефицит памяти). Деструктор, подобно своему собрату-конструктору, может содержать набор операторов, которые вызываются средой исполнения (их нельзя вызвать непосредственно из программы), когда происходит переполнение памяти.

2.4. Свойства (рассмотрены выше). Доступ к свойствам осуществляется так же, как к переменным-членам. Различие состоит в том, что свойство содержит операторы, которые исполняются подобно операторам метода, когда происходит обращение к нему. Свойства часто используются [вместо аксессоров и мутаторов (которые обычно имеют имена, наподобие **GetDistance** и **SetDistance**)] для доступа к закрытым переменным, поддерживая, таким образом, принцип инкапсуляции.

2.5. Индексаторы используются с классами, представляющими массивы. Так же, как свойства обеспечивают удобный доступ к отдельным переменным-членам, индексаторы выполняют эту функцию для массивов, размещенных внутри классов.

2.6. Операции. Иногда имеет смысл (для повышения читаемости кода) объединить два объекта с помощью операции. Например, можно написать оператор наподобие totalTime = myTime + yourTime, где все три переменных – объекты класса TimeInterval. Для достижения такой функциональности в классы включают операции-члены, которые задают набор команд, исполняющийся при объединении объектов в выражении с данной операцией. Этот процесс называют перегрузкой операции.

П.1.3. Вложенные типы — это классы, структуры, перечисления и интерфейсы, определенные в пределах тела класса. Они позволяют скрыть типы, которые используются только в пределах класса. Подобно тому, как вспомогательные методы объявляются закрытыми для уменьшения внешней сложности класса, вложенные типы помогают снизить количество классов, необходимых для сборки программы.