



ГЛАВА 8

Введение в классы

За 50—60 лет своего развития программирование прошло очень большой путь: от начала разработки программ в машинных кодах через создание и применение простейших языков символического кодирования до выпуска современных гигантских продуктов, обеспечивающих работу клиента в распределительных сетях на основе новейших подходов. Разработка и создание какой-нибудь стандартной программы перевода данных из одной системы счисления в другую, новой процедуры сортировки массива данных, настраиваемой на потребности заказчика программы обработки анкет, и прочее было заметным событием в те годы. Мысль тех, кто был связан с программированием, неустанно работала в направлении, как упростить очень сложный труд программиста, повысить производительность его труда, добиться надежности и качества программ. Сначала было замечено, что многие части машинных алгоритмов в разных задачах повторяются. Это привело к созданию так называемых стандартных программ — процедур, которые выполняли эти стандартные действия, например, такие как перевод чисел из одной системы счисления в другую, сортировка массива данных и т. п. Каждому программисту уже не надо было, например, беспокоиться о создании участка сортировки данных в своей программе. Он просто пользовался стандартной процедурой, поставляемой вместе с математическим обеспечением вычислительной машины. И это ускоряло процесс разработки программы. Потом пришли к понятию структурного программирования. Это был значительный шаг вперед. Была разработана методология создания программы, в основе которой лежал принцип создания программы как структуры, состоящей из подпрограмм, создаваемых для обработки повторяющихся блоков программы, и принципа выполнения программы "сверху вниз", когда в последовательности выполнения опе-

раторов и отдельных блоков программы не было возврата назад. Умри, но сделай так, чтобы не было возвратов назад! Это обеспечивало более быструю отладку программы. Выход из цикла — только вперед (вспомните оператор `break`)! Возврат назад — только в пределах оператора цикла! Передача управления из оператора `if...else` — только вперед! Долой оператор `goto`! Даже была доказана теорема о том, что любую схему алгоритма можно представить в виде композиции вложенных блоков `begin` и `end`, условных операторов `if`, `then`, `else`, циклов с предусловием (`while`) и, может быть, дополнительных логических переменных (флагов). Подход был серьезный. Но жизнь не стоит на месте. С появлением более новой техники — мощных компьютеров — открылись и новые возможности, о которых раньше даже не мечтали. Да и о чем ты можешь мечтать, когда оперативная память твоего "компьютера" — всего 1024 малюсеньких ячейки (1 К), а внешняя — один лентовод на 64 К, который практически все время не работает! И скорость в 3 тыс. операций в секунду.

Оказалось, что у процедурно-модульно-структурного подхода имеются значительные недостатки, которые можно устранить, используя мощь современных компьютеров. Во-первых, в программе данные и подпрограммы их обработки (процедуры и функции) формально никак не связаны. А хотелось бы, чтобы было наоборот. Например, вы решаете задачу по обработке данных о каком-то доме. Удобнее было бы, чтобы данные по дому и функции по обработке этих данных хранились в одной "коробке". Причем, было бы еще удобнее, если бы эта "коробка" подошла и для решения задачи по другому дому с такими же характеристиками, как и первый дом. То есть, говоря языком стандартизации, чтобы "коробка" была стандартной для данного класса домов. Во-вторых, данные в "коробке" не должны быть доступны всем, кто работает с коробкой. Они должны быть защищены от прямого доступа к ним. Только через функции, которые прячутся в этой "коробке". В-третьих, было бы полезным, чтобы элементы одного дома, спрятанные в "коробке", не хранились там без дела, когда данные дома не обрабатываются, а были бы использованы, если они подходят и к некоторым другим домам. Чтобы можно было брать такие элементы, не создавая их заново для другого класса домов, добавлять к ним новые, необходимые для нового класса, и создавать свою новую "коробку" для нового класса домов. И т. д. Это уже совсем другой подход к программированию. Он ориентирован не на модули-процедуры-функции, а на некие объекты, на то, чтобы в программе они создавались (описывались), чтобы внутри них хранились функции по обработке данных этих объектов (функции, естественно, надо разработать, но в рамках описания объекта), чтобы

данные по объектам вводились и попадали в сам объект и никуда больше, а полезные для других объектов элементы уже готового объекта можно было бы унаследовать другому создаваемому объекту. Раньше, например, могли мы, имея некоторую стандартную процедуру, формально взять из нее какую-то ее часть и применить в другой процедуре? Не могли. А в системе объектно-ориентированного программирования (ООП), к пониманию которого мы подводим читателя, это вполне возможно. Мы видим, что у такого подхода в программировании более высокий уровень, говоря языком политики, обобществления. От простейших стандартных программ по переводу чисел из десятичной системы счисления в двоичную мы приходим к созданию стандартных конструкций по описанию и обработке данных человека, дома, автомобиля, поликлиники, завода, страны.

Ранее мы изучили различные типы данных: целых чисел, чисел с плавающей точкой, строковых данных, типы организации данных, названных массивами, типы организации данных, названных перечислениями. Для ООП характерны типы данных, которые называются *классами*. Например, для типов "массивы" и "перечисления" мы объявляли (описывали по специальной схеме) сам тип, потом объявляли некую переменную этого типа. Затем инициализировали эту переменную некоторыми значениями. У нас существовали правила, как обращаться к элементам объявленного типа данных с помощью имени переменной. Например, если у нас был объявлен массив `M[]` данных, то элемент массива мы доставали, указывая в квадратных скобках номер (индекс) этого элемента в массиве: `M[i]`. Для перечислимого типа данных была своя схема описания и свое правило обращения к элементу этого типа данных. Для типа "класс" соблюдаются такие же правила: этот тип данных описывается по своей схеме (шаблону), переменная этого типа объявляется по общим правилам (`<имя_типа> <имя_переменной>`), по своим правилам переменная инициализируется, т. е. ей придаются некоторые начальные значения. Обратим внимание на два момента: когда мы описываем тип данного (в нашем случае тип "класс"), то описываем фактически схему, шаблон, по которому в дальнейшем будет создан объект данного типа. А когда мы этот шаблон наполняем содержанием, то тем самым создаем объект с данным содержанием, т. е. нечто, что можно пощупать. Объект уже надо размещать в памяти. Он требует пространства. Схема, шаблон тоже где-то хранятся, тоже требуют некоей памяти, но это как бы вспомогательная, не основная память. Вот, например, мы объявили целочисленную переменную `i`. Компилятор для нее создает некий шаблон: выделяет в специальной памяти 4 байта. И все. Но когда мы этой переменной присваиваем, скажем, значение 5, то тем самым создаем кон-

кретный числовой объект, с которым можно работать. Просто с объявлением `i` еще работать нельзя. Не с чем. А с объектом, даже можно сказать, с экземпляром этого шаблона `int i`, который равен 5, работать можно. Если мы присвоим переменной `i` другое значение, например 6, можно сказать, что мы из шаблона `int i` получили (создали) другой объект: число 6.

Точно так же и с классом: мы его описали, получили просто описание, шаблон. И ничего больше. А как только мы по этому шаблону создали переменную и наполнили ее неким содержанием, то получили объект, соответствующий данному содержанию. Говорят, что получили экземпляр класса. Наполнили переменную другим содержанием — получили другой объект с другим содержанием, другой экземпляр класса. Объекты (экземпляры) уже размещаются в памяти. Класс — это ссылочный тип. Поэтому он размещается в динамической куче оператором `new`. А почему сделали класс ссылочным типом данного? Потому что объекты, получаемые из этого класса, могут быть огромными (например, какой-нибудь крупный завод). А перемещать в памяти, как мы видели, ссылочные данные намного проще нессылочных: переслал только ссылку кому надо и не тронул огромный массив. Большой выигрыш в скорости обработки.

Прежде чем изучать конкретную структуру класса, отметим, что класс как совокупность элементов (членов класса) состоит из членов, которые называются *полями*, и из членов, оперирующих данными этих полей. Эти последние могут быть конструкторами, методами, свойствами, событиями и др. *Методы* — это функции. Так функции называются в классах. *Конструкторы* — это методы, которые позволяют инициализировать класс, тем самым создают из класса объект, размещая его в памяти. То есть конструктор — это обычная (по структуре) функция, получающая на свой вход данные, которые присваиваются полям класса. Иными словами, из пустого шаблона за счет задания полей получается объект.

Например, пусть мы имеем класс `MyCar`. Это тип данного, как мы видели. Объявляем переменную этого типа. Например, `car` (автомобиль): `MyCar car;`. Но это пока ничто: объявление и не более того. С такой переменной работать нельзя. Допустим, у автомобиля, класс которого мы хотим описать, есть такие характеристики (поля): марка (`Type`), имя владельца (`Name`) и скорость (`Speed`). Тогда функция, которая призвана задать эти поля, должна иметь в своем заголовке эти три параметра (`string Type, string Name, float Speed`). А каким должно быть имя у такой функции? Оно специфично и совпадает с именем класса. Впол-

не логично. Ведь конструктор — это то, что создает из класса объект. Итак, для нашего предполагаемого класса его конструктор будет иметь вид

```
MyCar(string Type, string Name, float Speed)
```

А что должно быть в теле конструктора? Операторы, которые присваивают значения полям класса. Если поля в классе описаны как `string type`, `string name`, `float speed`, то в теле должны быть операторы:

```
Type=Type; name=Name; speed=Speed;
```

То есть в итоге конструктор класса `MyCar` будет иметь вид:

```
MyCar(string Type, string Name, float Speed)
{
    type=Type;
    name=Name;
    speed=Speed;
}
```

Если мы теперь из класса хотим создать конкретный объект, например автомобиль Чака Норриса, то должны записать:

```
MyCar Ch_nor_car = new MyCar("Porshe", "Chuck Norris", 250.0);
```

По этому оператору конструктор создаст объект с именем `Ch_nor_car`, оператор `new` разместит объект (или — экземпляр класса `MyCar`) в динамической куче и выдаст адрес начала объекта в этой куче. Адрес будет положен на полочку для переменной `Ch_nor_car`.

Ну а теперь надо бы посмотреть, как создается (описывается) класс. Простейший вид описания класса такой:

```
class Car
{
}
```

Здесь `class` — ключевое слово, `Car` — имя класса. А где же конструктор? Здесь конструктор не указан. Он идет по умолчанию. Вообще, если в классе конструктор не указывается, то он по умолчанию берется из класса `Object` — специального класса, из которого берут свое начало остальные классы. Они, как говорят, потомки класса `Object`. Конструктор по умолчанию инициализирует поля создаваемого класса значениями, принятыми для полей класса по умолчанию. Вообще-то тело класса не пусто. Иначе зачем этот класс создавать? В теле должны быть определены члены класса и методы (т. е. функции), которые работают с полями-членами и другими членами класса. Ну и, конечно, задан конст-

руктор класса. Конструкторов может быть более одного, потому что может потребоваться создавать объекты класса на основе не всех полей, а некоторых. Но все конструкторы должны обязательно иметь имя класса, но с разными параметрами. Члены класса группируются в две секции: общедоступные члены (те, что доступны для обращения к ним не только из методов самого класса, но и из других программ приложения) и приватные члены (к ним имеют доступ только методы данного класса). Члены общедоступной секции помечаются ключевым словом `public`, а приватной (частной) — словом `private`. Есть еще одно ключевое слово, обеспечивающее доступность членов класса. Это слово `protected` (защищенный). У класса есть свойство наследовать члены другого класса (об этом мы узнаем подробнее дальше). Наследуются только те члены класса, которые имеют атрибут `public`. Если мы хотим, чтобы какие-то члены создаваемого нами класса, попавшие в приватную секцию, тоже наследовались, им надо присвоить атрибут `protected`. Тогда такие члены будут попадать в другие классы только наследным путем.

Пример программы, в которой создается класс с двумя конструкторами и одним методом обработки полей класса, приведен в листинге 8.1.

Листинг 8.1

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 24.11.2012  
 * Time: 19:51  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace app21_class  
{  
    class MyCar  
    {  
        private string name;  
        private int speed;  
        private string owner_name;  
  
        // Конструктор класса  
        public MyCar(string Name,int Speed,string Owner_name)
```

```
{
    name=Name;
    speed=Speed;
    owner_name=Owner_name;
}

// Второй конструктор класса
public MyCar(string Name,int Speed)
{
    name=Name;
    speed=Speed;
}

// Метод1 класса
public int M1()
{
    speed=speed*2;
    return(0);
}
} // class

class Program
{
    public static void Main()
    {
        // Чтобы создать объект (с помощью конструктора),
        // надо, чтобы конструктор был доступен в программе
        // Main(), которая находится в другом классе.
        // Поэтому конструктор должен иметь атрибут public.
        // Это же касается и метода M1()

        MyCar car = new MyCar("Лада",120,"Иванов");
        car.M1();

        // car уже выше объявлена как тип MyCar
        car = new MyCar("Лада",120);
        car.M1();

        Console.WriteLine("Press any key to continue... ");
        Console.Read();
    }
}
```

Итак, в этой программе создается класс `MyCar` с тремя приватными полями (имя автомобиля, скорость и имя владельца).

```
private string name;  
private int speed;  
private string owner_name;
```

Так как все поля приватные, то из внешней программы, в частности из `Main()`, к ним доступа нет. Но есть доступ через метод `M1()`, который имеет атрибут `public` (общедоступный). То есть в данном случае, если мы хотим что-то с полями делать, то имеем только одно средство, которое в классе определено. Это метод `M1()`. Другого не дано. В классе определены два конструктора, которые позволяют создать два разных объекта. Один — на основании инициализации трех полей, другой — двух полей. Конструкторы должны быть общедоступными, чтобы вне класса с их помощью создавать объекты. Поэтому у конструкторов имеется атрибут `public`. Единственный метод класса (`M1()`) увеличивает скорость автомобиля в два раза. В основной программе `Main()` создаются два объекта двумя конструкторами, и в каждом случае выполняется метод `M1()`. Чтобы было видно, что же получается при вычислениях, программа была запущена в режиме отладки. Точки останова были поставлены так, чтобы можно было пошагово следить, как формируется объект `car` разными конструкторами. Значения, получаемые полями объектов в результате их инициализации конструкторами и воздействия на них метода `M1()`, показаны на рис. 8.1 и 8.2.

Заметим, что для обращения к элементу класса надо назвать класс и через точку написать имя требуемого элемента. Отметим также, что после определения полей класса, которые в объектах, полученных из этого класса, будут использоваться для представления их состояния, определяют другие члены, которые моделируют поведение объектов. В нашем случае — это метод `M1()`. Размещение одной строкой объекта, как показано в предыдущем листинге (`MyCar car = new MyCar("Лада", 120, "Иванов");`), не обязательно: можно сначала объявить тип переменной (`MyCar car;`), а потом создать экземпляр класса (объект):

```
car = new MyCar("Лада", 120, "Иванов");
```

Конструктор класса — это специальный метод, который вызывается при создании объекта. Этот метод не имеет, в отличие от других методов, никакого возвращаемого значения: ни типа `int`, ни типа `double`, и даже типа `void` не имеет. В `C#` каждый класс снабжается своим конструктором по умолчанию, хотите вы этого или нет. Это метод с именем класса,

```

37 class Program
38 {
39
40
41 public static void Main()
42 {
43     //Чтобы создать объект (с помощью конструктора), надо, чтобы конструктор был доступен
44     // в программе Main(), которая находится в другом классе.
45     //Поэтому конструктор должен иметь атрибут public.
46     //Это же касается и метода M1()
47
48     MyCar car = new MyCar("Лада",120,"Иванов");
49     car.M1();
50     Console.WriteLine(car.ToString());
51     Console.ReadLine();
52 }
53
54 }
55
56
57

```

Object state after first constructor and M1():

Field	Value
name	"Лада"
owner_name	"Иванов"
speed	120 (0x78)

Рис. 8.1. Значения полей объекта после работы первого конструктора и метода M1 ()

```

53
54     MyCar car = new MyCar("Лада",120,"Иванов");
55     car.M1();
56
57     //car уже выше объявлена как тип MyCar
58     car = new MyCar("Лада",120);
59     car.M1();
60
61     Console.WriteLine(car.ToString());
62     Console.ReadLine();
63 }
64
65
66

```

Object state after second constructor and M1():

Field	Value
name	"Лада"
owner_name	null
speed	120 (0x78)

Рис. 8.2. Значения полей объекта после работы второго конструктора и метода M1 ()

но без параметров. То есть, когда вы создаете экземпляр класса (или объект, что то же самое), вы должны написать, например,

```
MyCar car = new MyCar();
```

В этом случае полям класса будут присвоены значения, принятые в языке по умолчанию. Если же вам не подходят значения по умолчанию, то вы сами можете создать свой конструктор по умолчанию (т. е. без параметров), задав в его теле свои, нужные вам, значения полей класса. Например, для нашего случая можно было бы задать такой конструктор по умолчанию:

```
public MyCar()  
{  
    name="Лада";  
    speed=120;  
    owner_name="Иванов";  
}
```

В нашем примере в классе определены два конструктора. У них разное количество параметров, но типы параметров одинаковы. Но может быть и так, что один конструктор от другого отличается не только количеством параметров, но и их типами. Определение методов с одним и тем же именем, но с разным количеством и типом параметров называют *перегрузкой метода*. Таким образом, класс `MyCar` имеет перегруженный конструктор, чтобы предоставить более одного способа создания из этого класса объекта.

Ключевое слово *this*

В переменной с этим именем хранится адрес текущего объекта, т. е. объекта, с которым в данном месте программы происходит работа. Этот элемент введен в язык для разрешения неоднозначных ситуаций, когда, например, вы в конструкторе назвали параметр таким же именем, как поле. Формально компилятор пропустит этот вариант, но при выполнении программы можно получить не тот результат. Чтобы в таких случаях различать, какая переменная к чему относится при одинаковых именах, и введено это ключевое слово. Например, в случае нашего примера мы могли бы определить заголовок конструктора как

```
public MyCar(string name, int Speed, string Owner_name)
```

Здесь первый параметр по имени совпадает с именем поля класса. Поэтому в теле конструктора мы вынуждены записать строку

```
{  
    name=name;  
    ...  
}
```

Компилятор посчитает, что строка присваивается самой себе, и полю `name` класса ничего не присвоит. Поэтому в таких ситуациях надо использовать слово `this`, которое в данном случае подскажет компилятору, что `name` в левой части от знака присвоения принадлежит текущему объекту, а именно тому, который создается из класса, а не конструктору. В этом случае конструктор будет выглядеть так:

```
public MyCar(string name, int Speed, string Owner_name)  
{  
    this.name=Name;  
    speed=Speed;  
    owner_name=Owner_name;  
}
```

ПРИМЕЧАНИЕ

Члены класса могут иметь атрибут `static` (об этом будет сказано далее). В этом случае применение `this` вызовет ошибку компиляции, потому что статические члены действуют на уровне класса, а не объекта (они для того и создаются, чтобы быть доступными всем объектам), а на уровне класса `this` не существует.

Ключевое слово *static*

Члены класса могут иметь атрибут `static`. Необходимость его введения была вызвана теми обстоятельствами, что создаваемые из класса объекты должны были иметь по некоторым полям, например, общие значения или обладать общими методами. Это удобно, т. к. нет необходимости иметь в каждом объекте один и тот же метод, достаточно получить его на уровне класса, из которого создается этот метод. Например, вы создаете класс по персоналу предприятия. Допустим, каждый объект, получаемый из этого класса, — это отдельный работник со всеми своими характеристиками. У всех работников могут быть одинаковые, например, некоторые процентные надбавки. Такие поля следует делать статическими, и они будут во всех объектах одинаковы. Достаточно изменить их на уровне класса, как изменения отразятся во всех объектах. Или методы некоторых расчетов: изменился алгоритм, метод изменяется на уровне класса, и для всех сотрудников обновился метод расчета.

Ранее мы пользовались методом `WriteLine()`. Но мы ничего не говорили, откуда он и почему так записывается (`Console.WriteLine()`). Теперь мы можем сказать, что этот метод из класса `Console`, и он имеет атрибут `static`. Он не вызывается на уровне объекта (мы не можем написать `Console con = new Console();`, создавая объект из класса, а потом писать `con.WriteLine();`; компилятор не пропустит), а вызывается напрямую из класса `Console`. Если в некотором классе определены только статические члены, такой класс можно назвать обслуживающим. Пример тому — класс `Console`. Имеется, например, и класс `Math`, предоставляющий возможность работы с математическими величинами и функциями.

Статические члены класса могут оперировать только другими статическими членами, в противном случае на этапе компиляции возникнет ошибка.

Для обращения к статическому члену обязательно надо указывать имя класса, которому этот член принадлежит (как, например, мы это делали в случае применения функции (а теперь мы знаем — статического метода) консольного вывода).

Итак, при создании статических полей класса они распределяются по всем экземплярам класса. Для них в памяти выделяется специальный участок, который не сбрасывается, когда объект уничтожается. В то время как нестатические поля — это независимые копии полей для каждого объекта. И они при ликвидации объекта уничтожаются вместе с ним, т. е. память от них освобождается. Статические поля сохраняются, а нестатические — не сохраняются. Поэтому при создании класса одной из задач разработчика является определение, какие поля будут статическими, а какие нет.

Для вящей убедительности еще один пример. Пусть у работника требуется рассчитать некую процентную ставку. Она для всех работников рассчитывается по одному и тому же алгоритму, не зависящему от работника. Пусть работников на заводе три тысячи. То есть создано три тысячи экземпляров класса `Персонал`. И в каждом будет присутствовать метод расчета процентной ставки. Если алгоритм расчета изменился, надо будет из класса снова формировать три тысячи экземпляров с новым методом, если только метод не имеет атрибута `static`. А с этим атрибутом формировать заново экземпляры нет надобности, потому что в каждом экземпляре обращение к этому методу идет с указанием класса, а не объекта, поскольку метод — статический. А в классе метод уже изменен. И оттуда он будет доставаться уже измененным.

Статический конструктор

При создании класса создаются статические поля, которые прямо при создании получают конкретные (статические) значения. Например, `static Rate=0.12;`. А как быть в случаях, когда значение статического поля определяется не в момент создания класса, а в момент работы приложения, когда из приложения надо обратиться к базе данных, чтобы взять оттуда, например, значение процентной ставки? И это все надо сделать до создания объекта. И поле процентной ставки имеет атрибут `static`. Такие проблемы решает так называемый *статический конструктор*, который выполняется всего один раз перед конструктором, создающим объекты. Статический конструктор используется для инициализации любых статических данных или для определенного действия, которое требуется выполнить только один раз. Он вызывается автоматически перед созданием первого экземпляра или когда идет первое обращение к статическому члену. Вот пример:

```
class SimpleClass
{
    // Статическая переменная, которая должна быть
    // инициализирована во время выполнения программы:
    static long dt;

    // Статический конструктор, который вызывается только
    // один раз перед любым конструктором, создающим
    // экземпляр класса, или при первой встрече
    // переменной dt:

    static SimpleClass()
    {
        dt = DateTime.Now.Ticks;
    }
}
```

Исполняющая среда вызовет этот конструктор перед созданием экземпляра класса `SimpleClass` и перед первым обращением к переменной `dt`.

Статические классы

Если все члены класса — статические, то имеет смысл вынести слово `static` "за скобки" и объявить весь класс статическим. Например, запись `static class MyClass {}` будет правильной.

Принципы объектно-ориентированного программирования

Таких принципов — три: инкапсуляция, наследование и полиморфизм. В предыдущем материале мы частично касались этих понятий, но не заостряли на них внимание. Теперь настало время рассмотреть сущность этих принципов подробнее.

Инкапсуляция

Капсула по латыни означает "коробочка", "ин" — предлог "в". Поэтому дословный перевод первого принципа ООП — "в коробочке". В нашем случае это условная коробочка, черный ящик, в котором прячутся все детали реализации проблемы, но с помощью этого ящика программист может решить саму проблему, не отвлекаясь на детали ее реализации. Кроме того, "коробочка" содержит (хранит) данные, причем таким образом, что к ним нет прямого доступа извне. Есть доступ только с помощью методов, хранящихся тоже в коробочке. Таким способом обеспечивается защита данных от внешнего вмешательства. Тот, кто в старые времена программировал ввод-вывод, знает, какая это была морока. Сегодня есть такой класс, как, например, `Console`, в котором содержатся (спрятаны в деталях) методы `Read()`, `ReadLines()`, обеспечивающие ввод данных с клавиатуры, и ни у одного программиста не болит голова, что надо будет делать в программе блок ввода-вывода. Все спрятано в классе `Console`. Пиши себе строку `string s = ReadLines();`, и твоя проблема ввода с клавиатуры решена.

Другая сторона — в коробочке упрятаны данные о состоянии объекта, который описан в ней. К этим данным просто так не добаться, если им присвоить атрибут `private`. Только через специальные методы, находящиеся там же, в коробочке (как мы увидим — это методы с именами `get` (получить данное) и `set` (установить данное в определенное значение)). То есть просто так изменить напрямую состояние объекта и тем самым, возможно, разрушить его не получится. Это все мы наблюдали, когда создавали простейший класс, правила построения которого включают в себя соблюдение принципа инкапсуляции.

Чем же, какими синтаксическими средствами языка обеспечивается соблюдение принципа инкапсуляции при создании класса? Это использование ключевых слов `public`, `private`, `protected` (с ними мы уже встре-

чались) и `internal`. Смысл последнего будет рассмотрен позже. Ранее мы уже обращали внимание на тот факт, что когда задаются поля класса (а именно они определяют своими значениями состояние в данный момент объекта класса), эти поля должны задаваться с атрибутом `private`, чтобы к ним извне не было доступа. В таком случае они доступны только для методов, определенных в данном классе.

Что касается общедоступных (имеющих атрибут `public`) данных, то такими данными могут быть общедоступные константы, другие поля, которые определены лишь для чтения. Последние имеют специальный атрибут `readonly`. Но чем отличаются константы от полей "только для чтения"? Дело в том, что константы не всегда соответствуют всем требованиям ситуации с реализацией алгоритма. Часто случается так, что переменную нужно получить в результате расчетов, а потом сделать ее "только для чтения". В С# именно для таких случаев предусмотрен тип переменных `readonly`. Переменные поля `readonly` имеют большую гибкость, нежели `const` (атрибут, с которым объявляется константа), потому что позволяют перед присваиванием производить различные вычисления значения, которое должно быть "только для чтения". Правило использования таких полей говорит, что вы можете присваивать им значение только в конструкторе и нигде более (лишь конструктор инициализирует поля, это его функция). Одной из основных особенностей таких полей является то, что они могут принадлежать и экземплярам класса, а не быть статическими, как константы (при объявлении константы атрибут `static` запрещен, т. к. по своей сути константа — уже сама по себе статический элемент). Это позволяет получать различные значения полей "только для чтения" в разных экземплярах классов. Но если вы хотите сделать поле `readonly` статическим, то должны явно объявить его таковым, в отличие от полей `const`.

Рассмотрим пример. Пусть в некотором классе `A` имеется метод вычисления количества поставщиков `supp()`. Объекты, получаемые из класса `A`, должны работать какое-то время с одним количеством поставщиков, а через некоторый промежуток — с другим количеством. То есть на определенном отрезке времени поле "Количество поставщиков", назовем его `NumSupp`, должно быть как бы константой.

В описанном примере код может выглядеть так:

```
public class A
{
    public static readonly uint NumSupp;
    static A() // статический конструктор
```

```
{  
    NumSupp = supp (Data);  
}  
}
```

В данном конкретном примере мы объявляем нашу переменную как статическую и используем ее в экземпляре класса при каждом запуске программы. Функция `supp(Data)` вычислит количество поставщиков на данную дату, и с помощью конструктора поле `NumSupp` получит заданное значение, с которым станут работать все экземпляры класса.

Мы говорили ранее, что управление приватными данными класса осуществляется двумя методами с именами `get` и `set`. Однако есть еще один способ управления этими данными: определение свойства. Рассмотрим инкапсуляцию с использованием методов `get` и `set`.

Инкапсуляция с использованием методов *get* и *set*

Построим класс `Employee`, моделирующий сотрудника некоторого предприятия. Сначала определим в нем поле "Имя сотрудника". Мы хотим, чтобы вне класса к этому полю не было доступа. То есть это поле должно иметь атрибут доступа `private`. Вид объявления в классе будет `private string empName;` (имя — строковое данные, т. е. данные типа `string`). Чтобы прочитать это данные из класса, надо создать метод, который имел бы право читать это данные и выдавал бы его значение. Начинаться он должен с части `Get` (получить). То есть его имя будет, например, `GetName()`. Какой атрибут доступа ему присвоить? Этот метод будет вызываться из исполняющей программы, а не в самом классе. Поэтому он должен быть известен в исполняющей программе, т. е. быть общедоступным, не спрятанным в классе. Значит, его атрибут доступа — `public`. Он должен возвращать имя сотрудника. Следовательно, тип возвращаемого значения у метода будет `string` (как и тип имени сотрудника). Окончательно метод чтения поля `empName` будет выглядеть так:

```
public string GetName()  
{  
    return empName;  
}
```

Напомним, что `return` — это оператор возврата результата работы функции (в данном случае — метода класса). Поэтому в исполняющей программе обращение к методу можно было бы записать, например, так:

```
string s = GetName();
```

если бы все происходило не с членом класса. Но при работе с классом предварительно надо создать из класса объект по конкретному сотруднику (т. е. некоторой переменной `emp` присвоить тип класса `Employee`). Так что предыдущая запись на самом деле будет выглядеть следующим образом:

```
Employee emp = new Employee("Иванов И. И.", остальные аргументы  
конструктора);  
string s = emp.GetName();
```

В `s` появится "Иванов И. И.".

Теперь надо создать метод, который изменяет поле из внешней программы. Назовем его `SetName()` (установить значение поля `Name`). Здесь надо подумать, какие действия следует предварительно выполнить в теле этого метода, прежде чем изменить поле. Мы возьмем самое простое действие: проверим значение, которое присвоится полю, на его длину. Пусть, например, имя работника не должно превышать девяти символов. Вот эту-то проверку и зададим в методе. Ясно, что метод должен иметь общедоступность, т. е. должен быть снабжен атрибутом `public`. Возвращать по оператору `return` ничего не требуется, поэтому метод имеет тип возвращаемого значения `void`. А у себя на входе метод должен иметь параметр типа `string`, т. к. он должен принимать некую строку текста, которой заменит значение поля в классе. Таким образом, метод может иметь вид:

```
public void SetName(string Name)  
{  
    if(Name.Length > 9)  
        WriteLine("Ошибка: длина имени больше 9 ");  
    else  
        empName = Name;  
}
```

Предполагается, что метод находится в классе, поэтому для него переменная `empName` доступна.

В итоге для задания одного поля класса `Employee` в соответствии с принципом инкапсуляции получим программу, представленную в листинге 8.2.

Листинг 8.2

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 27.11.2012
```

```
* Time: 16:39
*
* To change this template use Tools | Options | Coding |
* Edit Standard Headers. */
using System;

namespace app23_set_get
{
    class Employee
    {
        private string empName;    // поле
        // Конструктор: должен иметь атрибут public,
        // иначе его нельзя будет вызвать
        // из основной программы
        public Employee(string name)
        {
            empName=name;
        }
        // Методы класса:
        public string GetName()
        {
            return empName;
        }
        public int SetName(string Name)
        {
            if(Name.Length > 9)
            {
                Console.WriteLine("Ошибка: длина имени " +
                                   "больше 9, имя = {0}", Name);
                return(0);
            }
            else
            {
                empName = Name;
                return(1);
            }
        } // if
    } // конец класса

    class Program
    {
        public static void Main()
```

```
{
    // Создаем объект из класса:
    Employee emp = new Employee("Иванов И.И.");
    // Достаем поле name из объекта (если бы поле
    // имело атрибут static, мы бы его достали
    // прямо из класса):
    string s=emp.GetName();
    Console.WriteLine("Поле name = {0}",s);

    Console.Write("Продолжить: <Enter> >");
    // Изменяем значение поля name:
    if(emp.SetName("Петров П.П.") == 0)
        emp.SetName("Петров П.");
    // Так, как выше, не делают, но это просто
    // для примера
    s=emp.GetName();
    Console.WriteLine("Поле name = {0}",s);
    Console.Write("Продолжить: <Enter> >");
    Console.ReadKey(true);
}
}
```

Результат работы программы представлен на рис. 8.3.

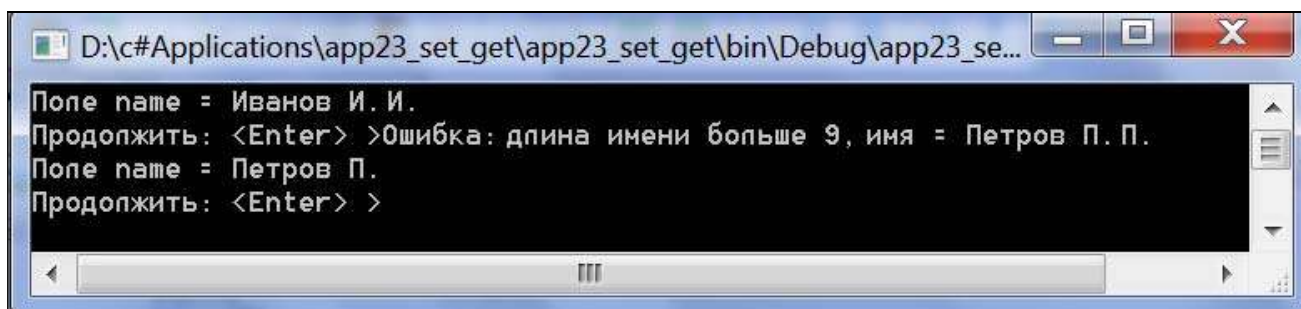


Рис. 8.3. Применение методов get, set для управления полем класса

Инкапсуляция с использованием свойств

В С# имеется другой, более удобный способ инкапсулирования данных — это использование так называемых свойств. *Свойства* — это упрощенное синтаксическое представление методов доступа к полю. Добавим к классу `Employee` еще одно поле: табельный номер работника. Определение свойств представлено в программе листинга 8.3.

Листинг 8.3

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 27.11.2012  
 * Time: 18:12  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace app24_properties  
{  
    class Employee  
    {  
        private string empName;  
        private int empID;  
        private float empPay;  
  
        // Конструктор с полями  
        public Employee(string Name, int ID, float Pay)  
        {  
            empName=Name; empID=ID; empPay=Pay;  
        }  
  
        // Конструктор со свойствами  
        /*  
        Чтобы конструкторы различались, параметр Pay взят  
        другого типа. Но поле empPay объявлено типа float,  
        поэтому понадобилось принудительное приведение  
        параметра Pay к типу float. this взят из-за  
        одинаковости имен параметров и свойств  
        */  
        public Employee(string Name,int ID, double Pay)  
        {  
            this.Name=Name; this.ID=ID; this.Pay=(float)Pay;  
        }  
        // Так определяются свойства Name, ID, Pay:  
        public string Name  
        {  
            get { return empName; }  
        }  
    }  
}
```



```
set
{
    if (value.Length > 9)
        Console.WriteLine ("Ошибка: длина имени > 9. "+
                             "Имя = {0}", value);
    else
        empName = value;
} // set
} // Name
public int ID
{
    get { return empID; }
    set { empID = value; }
} // ID
public float Pay
{
    get {return empPay;}
    set { empPay = value;}
}
} // Employee
class Program
{
    public static void Main(string[] args)
    {
        Employee emp = new Employee("Иванов", 411, 15000);

        // Вывод полей объекта через вывод свойств:
        Console.WriteLine("empName = {0}", emp.Name);
        Console.WriteLine("empID = {0}", emp.ID);
        Console.WriteLine("empPay = {0}", emp.Pay);

        // emp.SetPay(emp.GetPay() + 1000);
        // Без свойств надо было бы так делать
        emp.Pay += 1000; // emp.Pay будет равно 16000
        Console.WriteLine("empPay после добавления 1000 к свойству
Pay  = {0}", emp.Pay);

        // Создание объекта из конструктора с помощью
        // задания свойств
        Console.WriteLine("Создание объекта через свойства:");
        Employee emp1 = new Employee("Петров", 001, 25000);
```

```
// Вывод полей объекта через вывод свойств:
Console.WriteLine("empName = {0}", emp1.Name);
Console.WriteLine("empID = {0}", emp1.ID);
Console.WriteLine("empPay = {0}", emp1.Pay);

Console.Write("Press any key to continue... ");
Console.Read();
    }
}
}
```

Мы видим, что свойства имеют такие же имена, как и соответствующие поля, для обработки которых создаются свойства. Тип доступа к свойствам — `public`, т. к. это фактически методы, которые станут вызывать-ся вне данного класса, поэтому и должны быть общедоступными. У свойства есть тело, ограниченное фигурными скобками, в которое помещены два специальных метода управления соответствующим свойству полем. У методов заданные в среде исполнения имена `get` (получить значение поля) и `set` (установить значение поля). Метод `get()` возвращает значение поля, метод `set()` присваивает полю значение переменной `value`, которое определяется в основной программе, исполняющей метод `set()`. При этом тип `value` всегда совпадает с типом свойства. Свойства хороши не только более короткой формой записи, но и также тем, что они могут участвовать в операциях внутри класса. Например, если в класс добавить поле `private float empPay` (зарплата), которому будет соответствовать свойство, определенное как

```
public float Pay
{
    get {return empPay;}
    set { empPay = value;}
}
```

(компилятор потом преобразует эту короткую запись в обыкновенные функции-методы), то если потребуется увеличить зарплату, скажем, на 1000, то при отсутствии аппарата свойств надо было бы писать операторы:

```
Employee emp = new Employee("Иванов");
emp.SetPay(emp.GetPay() + 1000);
```

а с использованием свойства `Pay` можно записать

```
emp.Pay += 1000;
```

Намного проще.

Как доставать значения полей, если мы ввели понятия свойств? А мы уже показали частично в операторе

```
emp.Pay += 1000;
```

То есть вместо имени поля после точки надо поставить имя свойства. В данном случае компилятором будет построено обращение к методу `get()`, который выведет нас на поле. А обращение к полю для вывода его содержимого (раньше мы обращались через метод `GetИмяПоля()`) будет таким: `emp.Pay`. Замечательно! С вводом понятия свойств мы получили возможность в основных программах, использующих классы, даже не догадываться, что существуют какие-то методы `get` и `set`, хотя раньше в основной программе, как мы видели, надо было писать обращение к методам `SetName()`, `GetName()` и т. п. И еще одна возможность появилась после ввода понятия свойств: в конструкторе теперь можно не писать присвоение параметров полям класса, а сразу — свойствам. Если раньше мы писали, например, конструктор в виде:

```
public Employee(string name)
{
    empName=name;
}
```

то теперь можно писать в виде:

```
public Employee(string name)
{
    Name=name;
}
```

где `Name` — это имя свойства, связанного с полем `empName`, например. В конструкторе же можно задавать операторы, проверяющие на достоверность вводимые данные (это и раньше никто не мешал делать). Рабо-

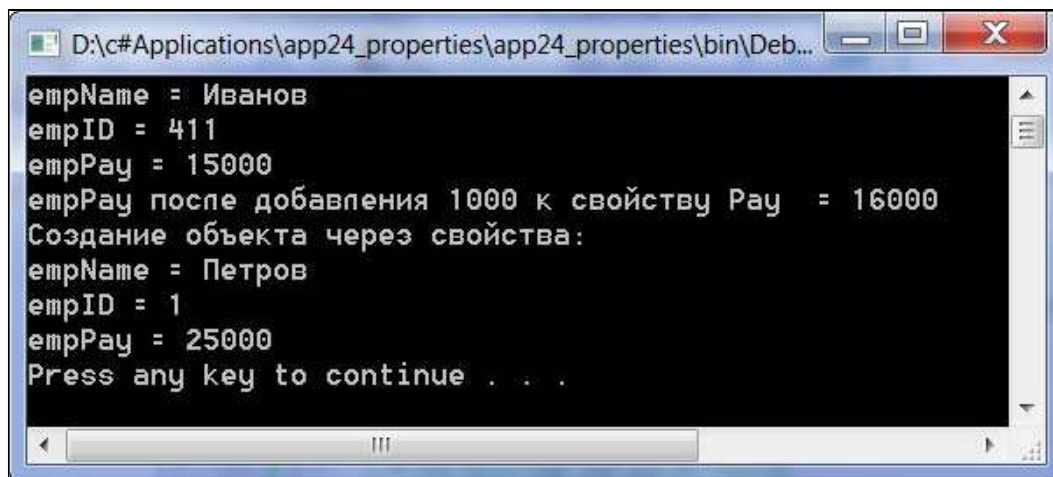


Рис. 8.4. Работа со свойствами класса

та со свойствами отражена в программе листинга 8.3. Результат представлен на рис. 8.4.

О доступности и статичности свойств

Когда при определении свойства заданы оба метода `get`, `set`, то такое свойство доступно как для чтения, так и для записи (модификации). Если какой-то из методов не указать, то свойство станет либо доступным только для чтения (присутствует только метод `get`), либо только для записи (присутствует только метод `set`).

Если поле класса имеет атрибут `static`, то вполне естественно, что и свойство для этого поля надо объявлять с таким же атрибутом.

Автоматические свойства

При определении свойства мы видели, что оно задается парой методов `get-set`, в телах которых (в фигурных скобках) только простейшие операторы — возврата и присвоения. В большинстве случаев описание `get-set` этим и ограничивается. Если в классе много свойств, то записывать почти одно и то же содержимое в фигурных скобках становится утомительным. Поэтому разработчики языка придумали, как обойти эту проблему. Они дали возможность отказаться от записи полей, а потом на их основе — записи свойств, а ввели возможность сразу определять свойства без описания полей (фактически была двойная работа), и свойства писать покороче.

Вот как это делается. Воспользуемся данными программы из листинга 8.3. Мы описывали поле `empPay` и свойство к нему `Pay` таким образом:

```
private float empPay; // поле
public float Pay      // свойство
{
    get { return empPay; }
    set { empPay = value; }
}
```

Теперь поле не определяют, а пишут сразу вид свойства:

```
public float Pay {get; set;}
```

(без точки с запятой в конце). Дальше во всем этом разбирается компилятор, который автоматически создает свойство таким, каким ему положено быть из этой синтаксической конструкции, и обеспечивает работу программиста только со свойством. Полей нет! Таким образом определенные свойства называют *автоматическими*.

Инициализация объекта

Мы знаем, что инициализация объекта происходит через конструктор, в котором при создании объекта задаются конкретные значения параметров. Существует еще один способ инициализации, синтаксис которого — список значений общедоступных полей и/или свойств (свойства по определению общедоступны), заключенный в фигурные скобки. Например, имеем класс `p`, описанный как

```
class p
{
    // Свойства (автоматические)
    public int x {get; set;}
    public int y {get; set;}

    // Конструкторы
    public p()
    {
    }
    public p(int v1, int v2)
    {
        x=v1;
        y=v2;
    }
}
```

С помощью этих конструкторов можно создать объекты обычным способом:

```
p ob1=new p(); ob1.x=10; ob1.y=20;
```

или

```
p ob2=new p(10,20);
```

А с помощью синтаксиса инициализации можно написать:

```
p ob=new p {x=10, y=20};
```

При таком синтаксисе конструктор класса — конструктор по умолчанию — вызывается неявно. Но можно его вызывать и явно, записав

```
p ob=new p() {x=10, y=20};
```

При инициализации объекта таким образом можно вызывать не только конструктор по умолчанию, но и любой другой определенный в классе. Например, в нашем случае можно записать:

```
p ob2=new p(10,20) {x=30, y=40};
```

Свойства класса получают, естественно, последние значения (30, 40).

Применение синтаксиса инициализации объектов более наглядно и удобно при большом количестве свойств, полей и конструкторов в классе. В этом случае не надо думать, какой конструктор выбрать для данного объекта: бери конструктор по умолчанию и в фигурных скобках пиши названия нужных свойств и/или полей и присваиваемые им значения. Программа по примерам инициализации приведена в листинге 8.4. Результат ее работы представлен на рис. 8.5.

Листинг 8.4

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 30.11.2012  
 * Time: 13:27  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace app25_obj_ini  
{  
    class p  
    {  
        // Свойства (автоматические)  
        public int x {get; set;}  
        public int y {get; set;}  
  
        // Конструкторы  
        public p()  
        {  
        }  
        public p(int v1, int v2)  
        {  
            x=v1;  
            y=v2;  
        }  
    }  
    class Program  
    {  
        public static void Main()  
        {  
            p ob1=new p(); ob1.x=10; ob1.y=20;
```

```
Console.WriteLine("Инициализация свойств через "+  
    "конструктор по умолчанию: x={0} y={1}",  
    ob1.x, ob1.y);  
p ob2=new p(10,20);  
Console.WriteLine("Инициализация свойств через " +  
    "двухаргументный конструктор: x={0} y={1}",  
    ob2.x, ob2.y);  
p ob=new p {x=10, y=20};  
Console.WriteLine("Инициализация свойств через "+  
    "синтаксис инициализации: x={0} y={1}",  
    ob.x, ob.y);  
Console.Write("Press any key to continue... ");  
Console.Read();  
}  
}  
}
```

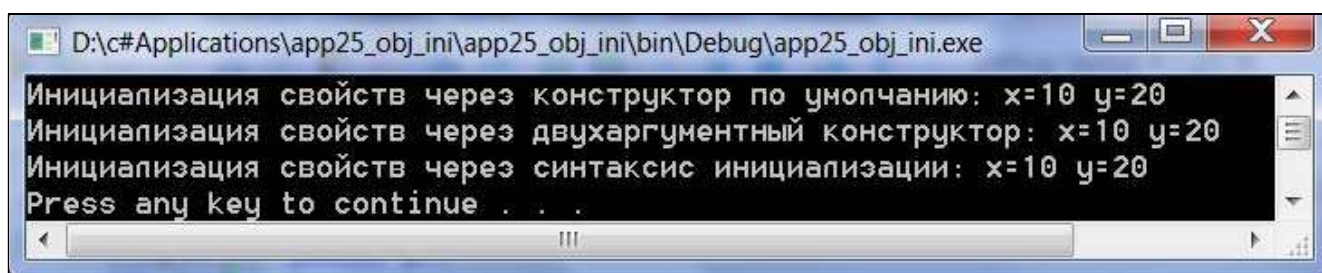


Рис. 8.5. Различные виды инициализации объекта

Организация работ при описании класса.

Атрибут *partial*

При разработке, а особенно при эксплуатации класса (когда он участвует в эксплуатации какого-то приложения), существуют участки описания класса, которые относительно постоянны. Например, такие элементы класса, как описание полей, свойства и конструкторы, довольно постоянны, чего нельзя сказать о методах, которые могут довольно часто изменяться, особенно во время эксплуатации. В этих условиях имеет смысл большие классы разбивать на части, присваивая частям атрибут *partial* (частичный). Тогда можно отдельно заниматься только той частью, которая требует модификации, не трогая остальные части. При компиляции компилятор соберет все частичные классы (так они станут называться) в один общий класс. Вот как можно создавать частичные классы: разносят общий (длинный) файл с расширением *cs* на несколько файлов с таким же расширением и именами, помещая в них необходи-

мые элементы (конструкторы, методы, поля свойства). Каждому файлу добавляют описатель `partial`. Потом эти файлы подключают к общему проекту (создают проект с программой `Main()`). Файлы с расширением `cs` формируют с помощью программы WordPad (при записи текстового файла дают ему расширение `cs`). Объединяют эти файлы в один, зайдя в меню среды программирования: **Project | Add | Existing Item**. Откроется диалоговое окно для поиска файлов. Надо найти ранее сформированные `cs`-файлы и подключить. Пример программы с частичными классами представлен в листингах 8.5—8.7. Результат приведен на рис. 8.6.

Листинг 8.5

```
// Первый частичный класс

partial class Employee
{
    // Поля класса
    private string empName;    // имя
    private int empID;        // табельный номер
    private float currPay;    // зарплата
    private int empAge;       // возраст
    private string empSSN;    // номер медицинского полиса
    private static string companyName; // название компании
    // Конструкторы
    public Employee() { }
    public Employee(string name, int age, int id,
                    float pay, string ssn)
    {
        // Инициализация свойств
        Name = name;
        Age = age;
        ID = id;
        Pay = pay;
        SocialSecurityNumber = ssn;
    }
    static Employee() // Статический конструктор
    {
        companyName = "Union Fenosa";
    }
}
```


Листинг 8.6

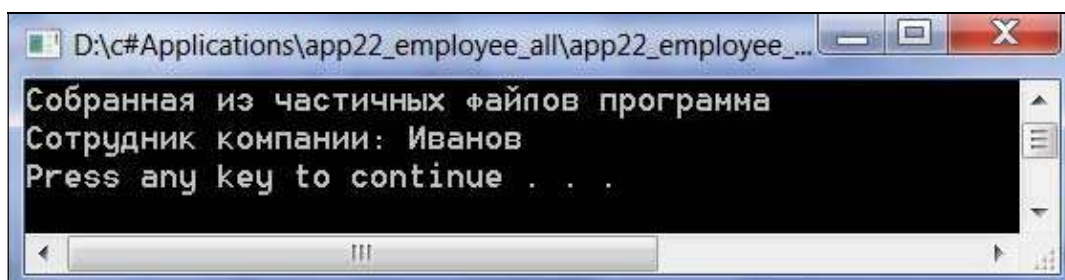
```
// Второй частичный класс
partial class Employee
{
    // Свойства
    public static string Company
    {
        get { return companyName; }
        set { companyName = value; }
    }
    public string Name
    {
        get { return empName; }
        set
        {
            if (value.Length > 9)
                ; // Console.WriteLine("Имя больше 9 СИМВОЛОВ");
            else
                empName = value;
        }
    }
    public int ID
    {
        get { return empID; }
        set { empID = value; }
    }
    public float Pay
    {
        get { return currPay; }
        set { currPay = value; }
    }
    public int Age
    {
        get { return empAge; }
        set { empAge = value; }
    }
    public string SocialSecurityNumber
    {
        get { return empSSN; }
        set { empSSN = value; }
    }
} // class
```

Листинг 8.7

```
/* Основная программа
 * Created by SharpDevelop.
 * User: user
 * Date: 30.11.2012
 * Time: 15:43
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app22_employee_all
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Собранная из частичных файлов "+
                              "программа");

            Employee emp = new Employee {Name="Иванов"};
            // Инициализация свойства через синтаксис
            // инициализации
            Console.WriteLine("Сотрудник компании: {0}",
                              emp.Name);
            Console.Write("Press any key to continue... ");
            Console.ReadKey(true);
        }
    }
}
```

**Рис. 8.6.** Результат работы программы, собранной из частичных классов

Наследование

Наследование — это возможность использования существующих классов для создания новых классов. Новые классы могут создаваться на основе существующих классов, наследуя их функциональность, т. е. то, что умеют делать старые классы. При этом старые классы называют *родителями* или *базовыми классами*, а новые — *потомками* или *дочерними классами* (почему не сыновьями — не понятно), или *производными классами*. Новые классы фактически расширяют функциональность старых, потому что умеют делать то, что умеют старые, но и еще кое-что, чего нет в старых.

При наследовании в дочернем классе становятся доступными все члены родительского класса с атрибутом `public` или `protected`.

Зададим базовый класс в виде, показанном в листинге 8.8.

Листинг 8.8

```
class car                // Автомобиль
{
    const int maxSpeed = 90; // Макс. скорость автомобиля
    int carSpeed;           // Поле
    public int Speed        // Свойство (должно быть
                           // общедоступным)

    {
        get {return carSpeed;}
        set
        {
            carSpeed = value; // Одна из форм синтаксиса
                              // задания свойства
            if(carSpeed > maxSpeed) // Контроль задания скорости
                                    // автомобиля пользователем
                carSpeed = maxSpeed;
        }
    } // Speed
} // car
```

Здесь задан класс `car` (автомобиль) всего с одним полем — скорость автомобиля (`carSpeed`). На его основе создано свойство `Speed`, в котором в методе установки свойства (`set`) проверяется, не введет ли пользователь недопустимую величину максимальной скорости, заданной константой `maxSpeed`. Если такое произойдет, то метод `get` возьмет за вве-

денную скорость величину максимальной скорости. Допустим, вы хотите построить другой класс автомобиля (`My_car`) и такой, у которого были бы те же поля и свойства, что и у `car`, но чтобы новый класс еще содержал в себе марку автомобиля. Пусть это новое поле будет называться `carName`. Ясно, что нелогично снова создавать класс с теми данными, что уже разработаны, и добавлять в него новые данные. Лучше бы было воспользоваться уже существующим, а затем добавить свои. То есть желательно унаследовать уже готовую функциональность и добавить свою. Это и позволяет делать принцип наследования, заложенный в C#. Точнее, в его компилятор. В нашем случае мы можем спокойно написать синтаксическую конструкцию, представленную в листинге 8.9.

Листинг 8.9

```
class My_car : car
{
    const int maxName = 20;
    string carName;      // Поле — марка автомобиля
    public string Name    // Свойство (должно быть
                          // общедоступным)

    {
        get {return carName;}
        set
        {
            carName = value;
            if(carName.Length > maxName)
                carName="Ошибка: марка должна иметь " +
                    "не более 20-ти символов";
        }
    }
}
```

Конструкция похожа на ту, что показана в листинге 8.8, поэтому поясним только ее первую строку: так задается класс-наследник — через двоеточие от его имени пишется класс-родитель.

Сведем теперь данные листингов 8.8 и 8.9 в одну программу, добавив к этому метод `Main()`, чтобы проверить, работает ли вновь созданный класс `My_car`. Мы должны убедиться, можно ли из этого класса достать свойство `Speed` из класса-родителя, можно ли изменять его в дочернем классе и работать с маркой автомобиля в дочернем классе. Вся программа полностью приведена в листинге 8.10, а результат ее работы показан на рис. 8.7.

Листинг 8.10

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 02.12.2012  
 * Time: 12:42  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace app27_inherit  
{  
    class car // Автомобиль  
    {  
        const int maxSpeed = 90; // Макс. скорость автомобиля  
        int carSpeed; // Поле  
        public int Speed // Свойство  
        {  
            get {return carSpeed;}  
            set  
            {  
                carSpeed = value; // Одна из форм  
                                   // задания свойства  
                if(carSpeed > maxSpeed) // Контроль задания  
                                   // скорости автомобиля пользователем  
                    carSpeed = maxSpeed;  
            }  
        } // Speed  
    } // car  
  
    class My_car : car  
    {  
        // Здесь нет конструктора: он берется по умолчанию  
        const int maxName = 20;  
        string carName; // Поле — марка автомобиля  
        public string Name // Свойство  
        {  
            get {return carName;}  
            set  
            {  
                carName = value;  
            }  
        }  
    }  
}
```

```
        if(carName.Length > maxName)
            carName="Ошибка: марка должна иметь " +
                "не более 20-ти символов";
    }
}

class Program
{
    public static void Main()
    {
        Console.WriteLine("Данные по классу-наследнику:");

        car obj_car = new car {Speed = 85}; // Инициализация
                                           // объекта – базового класса
        Console.WriteLine("Установленная скорость в " +
            "родительском классе: {0}",
            obj_car.Speed);

        My_car cr = new My_car {Name = "Volvo"};
                    // Инициализация объекта
        cr.Speed=88;
        Console.WriteLine("Марка автомобиля: " +
            "{0}\nСкорость в дочернем классе: {1}",
            cr.Name, cr.Speed);

        Console.Write("Press any key to continue... ");
        Console.Read();
    }
}
```

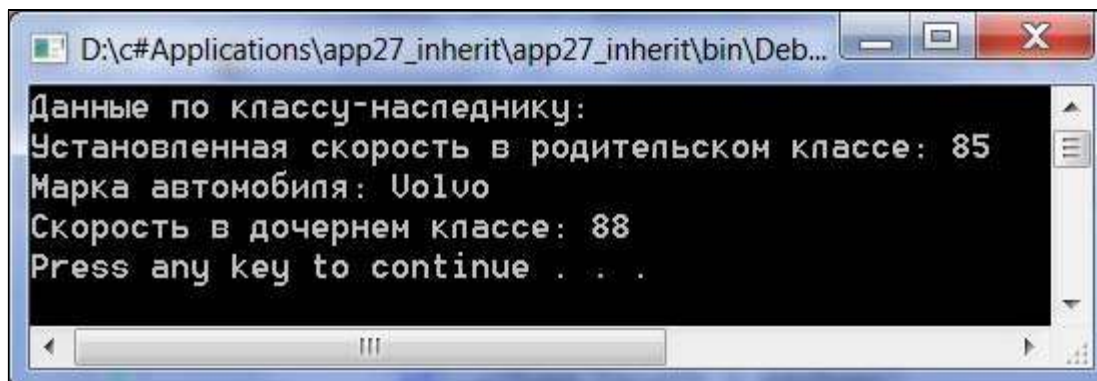


Рис. 8.7. Результаты работы с наследуемыми и родительскими свойствами

Во-первых, надо не забывать, что когда вы создаете свойства, те должны иметь атрибут `public`, т. е. быть общедоступными. Если атрибут `public` не указан, компилятор такому члену класса автоматически присваивает атрибут `private`. Но если свойства — автоматические, тогда им присваивается `public` по умолчанию. В нашем случае свойства объявлены не автоматические, а традиционным способом, правда, с усовершенствованным синтаксисом.

Во-вторых, когда создается объект производного класса, все его члены должны получить значения, в том числе, естественно, и те, что перейдут от родителя, через конструктор или путем доступа напрямую (см. `cr.Speed=88;`).

Запрет на наследование

В C# существует специальное ключевое слово. Если им пометить класс, то от этого класса нельзя будет наследовать. Это слово — `sealed` (запечатанный). Если вы объявили `sealed class car {члены класса}`, то при попытке компиляции конструкции `class My_car : car {члены класса}` компилятор выдаст ошибку. Например, многие системные классы запечатаны, т. е. не позволяют себя расширять за счет наследования. Конструкция

```
My_string : string {}
```

выдаст ошибку компиляции, т. к. класс `string` запечатан: пользоваться его членами — пожалуйста, но свои не добавляйте.

Конструкторы и наследование

В иерархии классов допускается, чтобы у базовых и производных классов были собственные конструкторы. В связи с этим возникает следующий резонный вопрос: какой конструктор отвечает за построение объекта производного класса: конструктор базового класса, конструктор производного класса или же оба? Оказывается, что конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса — производную часть этого объекта. И в этом есть своя логика, поскольку базовому классу неизвестны и недоступны любые элементы производного класса, а значит, их конструирование должно происходить отдельно.

Если конструктор определен только в производном классе (так называемый специализированный конструктор, т. е. с параметрами), то все происходит очень просто: конструируется объект производного класса,

а базовая часть объекта автоматически конструируется его конструктором, используемым по умолчанию (в базовом классе конструктора нет, как мы предположили, следовательно, берется конструктор по умолчанию). Когда конструкторы определяются как в базовом, так и в производном классе, процесс построения объекта несколько усложняется, поскольку должны выполняться конструкторы обоих классов. А в базовом классе может быть много конструкторов. Какой надо выполнять в данном конкретном случае? Тут приходится использовать ключевое слово `base`, которое находит нужный конструктор базового класса и выполняет его. Существует форма объявления конструктора производного класса, с помощью которого может быть вызван конструктор, определенный в его базовом классе:

```
конструктор производного_класса(список_параметров) : base
(список_параметров)
{
    // тело конструктора
}
```

где `base (список_параметров)` — это параметры, необходимые некоему конструктору в базовом классе для его запуска. Какой конструктор именно должен запускаться, компилятор определяет по количеству и типу параметров, заданных в методе (а это именно метод) `base`. После этого инициализированные поля станут известны в производном объекте.

Пример программы приведен в листинге 8.11. Результат работы представлен на рис. 8.8.

Листинг 8.11

```
/* Created by SharpDevelop.
 * User: user
 * Date: 02.12.2012
 * Time: 16:42
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app28_base
{
    class MyClass
    {
        public int x, y, z;    // Поля
```



```
// Конструктор базового класса
public MyClass(int x, int y, int z)
{
    /* this применен, т. к. имена полей и имена
       параметров в конструкторе совпадают.
       this.x означает, что x относится к полю x
    */
    this.x = x;
    this.y = y;
    this.z = z;
}

class inherit_class : MyClass
{
    int point; // поле

    // Конструктор производного класса с применением base:
    // в базовый класс конструктору передаются
    // аргументы x, y, z, т. е. первый параметр
    // конструктора участвует в вычислении поля
    // наследника, остальные три — в вычислении
    // полей родителя
    public inherit_class(int point, int x, int y, int z)
        : base(x, y, z)
    {
        this.point = point; // Инициализирует поле
                           // наследника
    }

    // Метод класса-наследника
    public void Pointer(inherit_class new_point)
    {
        // Установка полей базового класса в наследнике:
        new_point.x += new_point.point;
        new_point.y += new_point.point;
        new_point.z += new_point.point;
        Console.WriteLine("Новые координаты объекта " +
                           "в производном классе: {0} {1} {2}",
                           new_point.x, new_point.y, new_point.z);
    }
}
```

```
class Program
{
    static void Main()
    {
        Console.WriteLine("Работа с ключевым " +
                           "словом base\n");

        inherit_class obj = new inherit_class(5, 2, 3, 4);
        Console.WriteLine("Координаты объекта в базовом " +
                           "классе: {0} {1} {2}", obj.x, obj.y, obj.z);
        obj.Pointer(obj);
        Console.ReadLine();
    }
}
```

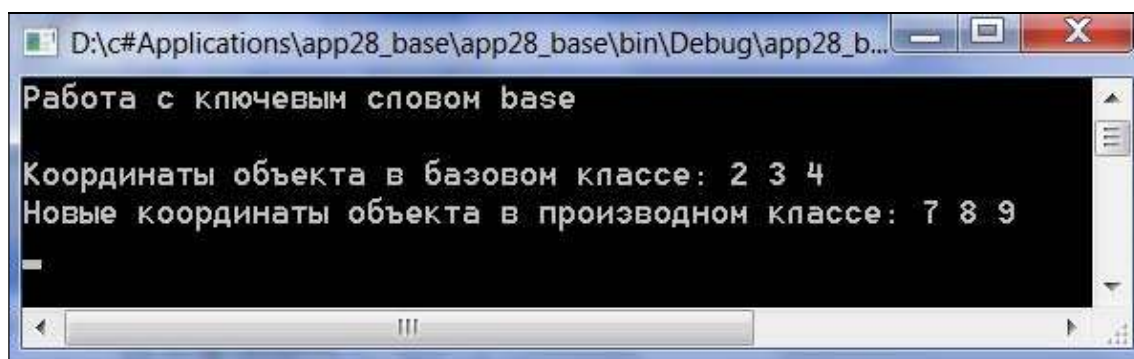


Рис. 8.8. Работа с ключевым словом base

Повторим, что с помощью ключевого слова `base` можно вызвать конструктор любой формы, определенной в базовом классе, причем выполняться будет лишь тот конструктор, параметры которого соответствуют переданным аргументам.

А теперь рассмотрим вкратце основные принципы действия ключевого слова `base`. Когда в производном классе указывается ключевое слово `base`, вызывается конструктор из его непосредственного базового класса. Следовательно, ключевое слово `base` всегда обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим классом. Аргументы передаются базовому конструктору в качестве аргументов метода `base()`. Если же ключевое слово отсутствует, то автоматически вызывается конструктор, используемый в базовом классе по умолчанию.

Добавление к классу запечатанного класса

Мы видели, что запечатанный класс не наследуется. А хотелось бы в некоторых ситуациях все-таки как-то использовать его члены в своем создаваемом классе. Как это сделать? Повторяю, наследование не проходит. Для решения этой проблемы применяется прием включения в создаваемый класс объекта, получаемого из запечатанного класса. Программа, реализующая сказанное, приведена в листинге 8.12, а результат ее работы показан на рис. 8.9.

Листинг 8.12

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 02.12.2012  
 * Time: 19:03  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace app29_include_sealed  
{  
    sealed public class A  
    {  
        int a; // Поля  
        int b;  
  
        // Свойства  
        public int A_a  
        {  
            get { return a; }  
            set { a=value; }  
        }  
        public int A_b  
        {  
            get { return b; }  
            set { b=value; }  
        }  
        // Конструктор  
        public A(int a,int b)  
        {this.a=a; this.b=b;}  
    }  
}
```

```
// Метод
public int M_A()
{
    return (a + b);
}

class B
{
    public int c;
    public A ab = new A(15,20);
    public B(int cc)
    {
        c=cc;
    }
}

class Program
{
    public static void Main()
    {
        B ba = new B(5);
        int sum=ba.c + ba.ab.M_A();

        Console.WriteLine("Использование запечатанного " +
            "класса\нкак объекта создаваемого");

        Console.WriteLine("Поле с класса B = {0}", ba.c);
        Console.WriteLine("Поля запечатанного класса A " +
            "равны {0}, {1}",ba.ab.A_a, ba.ab.A_b);
        Console.WriteLine("Результат суммирования поля " +
            "из B и работы метода из A: {0}", sum);

        // TODO: Implement Functionality Here

        Console.Write("Press any key to continue... ");
        Console.Read();
    }
}
```

Итак, имеется запечатанный класс A с двумя полями, доступ к которым осуществляется только через два соответствующих свойства — A.a и

A_b. В классе A есть метод `M_A()`, суммирующий значения закрытых полей класса A. Мы строим класс B с одним полем `c` и объектом `ab`, получаемым из запечатанного класса A. Вывод данных на экран показывает, что в классе B используется функциональность класса A (в частности, метод `M_A()`), хотя наследования не было, т. к. оно невозможно из-за запечатанности A. Заметим, что если объект одного класса вложен в объект другого класса, то доступ к элементам вложенного объекта идет через точку от имени объекта (это общее правило), а само имя объекта, которое определяет членство объекта в другом классе, идет через точку от имени объекта основного класса. В нашем случае, чтобы добраться до свойства `A.a` из запечатанного класса, надо было записать `ba.ab.A.a`.

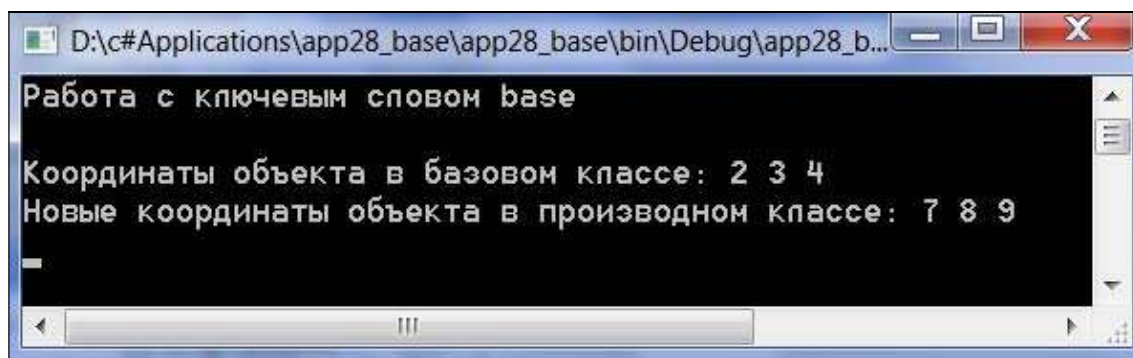


Рис. 8.9. Результат использования запечатанного класса в создаваемом классе

Вложенность классов

Рассмотренный выше пример является примером вложенности типов: в C# допускается вложенность типов. (В данном случае классов как типов. Про остальные подобные типы мы пока что ничего не знаем.) При этом вложенный тип считается обычным членом охватывающего или внешнего класса. Таким вложенным типом можно манипулировать, как и любым другим членом класса. Синтаксис таков:

```
public class OuterClass
{
    public class InnerClass1 {}
    private class InnerClass2 {}
}
```

Если внутренний класс станет использоваться вне класса, в котором он определен, его, естественно, надо будет указывать с именем включающего его класса (мы это видели в приведенном ранее примере на уровне объектов). Например, можем записать фрагмент основной программы:

```
static void Main()  
{  
    OuterClass.InnerClass1 obj1 =  
        new OuterClass.InnerClass1();  
}
```

Как выйти правильно на элементы такого объекта (даже если вложенность более глубокая, что тоже допускается), вам сообщит подсказчик среды исполнения (в нашем случае `SharpDevelop`): как только вы наберете точку после имени созданного объекта (в нашем случае после `obj1`), подсказчик откроет окно с перечнем элементов, из которых предстоит выбрать нужный. Если вы его в этом списке найдете, щелкните на нем, и он приклеится к набранному имени объекта. Поставьте точку после найденного имени. Откроется новый список. И т. д. Если вы не найдете в списке нужного вам имени (имени подобъекта, имени члена подобъекта и т. д.), то участок вашей программы, в котором вы находитесь, не видит нужного вам элемента. Причина может быть в том, что нужный вам элемент не имеет атрибута `public`. Или где-то вкралась синтаксическая ошибка, которая мешает увидеть элемент. Тут много вариантов. Главное — следить, чтобы в списке обязательно находилось имя нужного вам элемента класса. Если это не так, стоит разобраться, почему.

Полиморфизм

Слово это означает "многоформие". Когда в классе-потомке после наследования его от базового класса появляются методы базового класса, бывает, что возникает необходимость эти базовые методы переделать под потребность производного класса. Классический пример: классы "Многоугольники" и "Окружности". Оба унаследованы от базового класса `Object`, в котором определен метод `Draw()` — рисовать фигуру. Но в первом случае надо рисовать многоугольник, а во втором — окружность. А в обоих классах находится метод с одним и тем же именем `Draw()`. Принцип полиморфизма обеспечивает возможность разной "начинки" метода `Draw()`, оставляя нетронутым имя метода. То есть фактически старый метод базового класса в обоих его потомках имеет возможность получить новое содержание (ясно, что все это может делать компилятор). Скажете, мол, а при чем тут форма? Здесь не надо путать русские философские понятия содержания и формы, а вспомнить, что слово "формировать" по-гречески будет звучать как "морфи". С этой точки зрения старый метод `Draw()` получит в своих потомках две разные "формы". Отсюда полиморфизм: много форм.

Процесс переделки старого метода в новые носит название *переопределения метода* (method overriding). Для возможности переопределения надо дать какую-то информацию компилятору на сей счет. А информация эта такая: если вы в базовом классе хотите разрешить переопределять некоторый метод в подклассах, то этот метод надо снабдить атрибутом `virtual`. Следовательно, т. к. мы знаем, что метод `Draw()` рисует разные фигуры, можем предположить, что в классе-родителе он помечен атрибутом `virtual`. Такие методы называют *виртуальными*. Действительно, когда в базовом классе вы помечаете некий метод атрибутом `virtual`, вы тем самым даете возможность в будущем этот метод переопределить в некоем классе-потомке со своей функциональностью. То есть в момент присвоения вами атрибута `virtual` методу, других методов пока не существует, иными словами, они как бы существуют, но не по-настоящему, т. е. виртуально. Кроме того, метод, помеченный ключевым словом `virtual`, дает возможность применять этот метод по умолчанию, которая распространяется на всех потомков базового класса. Если дочерний класс решит, он переопределит такой метод, но он не обязан это делать. Может просто вызвать этот метод из базового класса как метод по умолчанию.

Из базового класса вы дали сигнал компилятору, что такой-то метод (пусть это будет `Draw()`) можно переопределять в будущем классе-потомке. А как потомок даст сигнал компилятору, что он переопределяет виртуальный метод? Потомок должен присвоить этому методу атрибут `override`. Все. Цепочка замкнулась. Значит в классе "Многоугольники" или в классе "Окружности" метод `Draw()` имеет атрибут `override`.

Если вас устраивает функциональность виртуального метода базового класса, и вы хотите еще добавить свои какие-то операторы, то чтобы не писать все тело старого метода, есть возможность воспользоваться ключевым словом `base`. Тогда, например, обращение к методу `Draw()` внутри его переопределения будет выглядеть как `base.Draw()`; , т. е. он нарисует фигуру заложенными в него операторами в классе-родителе, а потом дальше в переопределяемом теле вы можете записать какие-то свои операторы, которые, например, нарисованную окружность разобьют на три сектора.

Работа по переопределению метода показана в программе листинга 8.13. Результат работы представлен на рис. 8.10.

Листинг 8.13

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 04.12.2012
```

```
* Time: 12:30
*
* To change this template use Tools | Options | Coding |
* Edit Standard Headers. */
using System;

namespace app30_overriding
{
    public class A
    {
        int a; // Поля
        int b;

        // Свойства
        public int A_a
        {
            get {return a;}
            set {a=value;}
        }

        public int A_b
        {
            get {return b;}
            set {b=value;}
        }

        // Конструкторы
        public A() {}
        public A(int a,int b)
        { this.a=a; this.b=b; }

        // Виртуальный метод
        public virtual int M_A()
        { return (a + b); }
    }

    class B : A
    { int a;
      int b;
      int c;
```



```
// Свойство
public int C_c
{
    get {return c;}
    set {c=value;}
}
// Конструктор (указывается, какой конструктор
// базового класса будет вызван)
public B(int aa, int bb, int cc):base(aa, bb)
{
    a=aa; b=bb; c=cc;
}
// Переопределение метода базового класса: он должен
// поле "c" возводить в квадрат и добавлять результат
// к результату старого метода (из класса A)
public override int M_A()
{
    return (base.M_A() + c*c);
}
}
class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Результат работы " +
                           "переопределенного метода");
        B b_b = new B(5,6,7);
        Console.WriteLine("1-е поле в классе-родителе – {0}\n2-е
поле в классе-родителе – {1}", b_b.A_a, b_b.A_b);
        Console.WriteLine("Значение поля производного класса –
{0}\nРезультат переопределенного метода – {1}",b_b.C_c,b_b.M_A());

        Console.Write("Press any key to continue... ");
        Console.ReadKey(true);
    }
}
}
```

Все пояснения — по тексту программы.

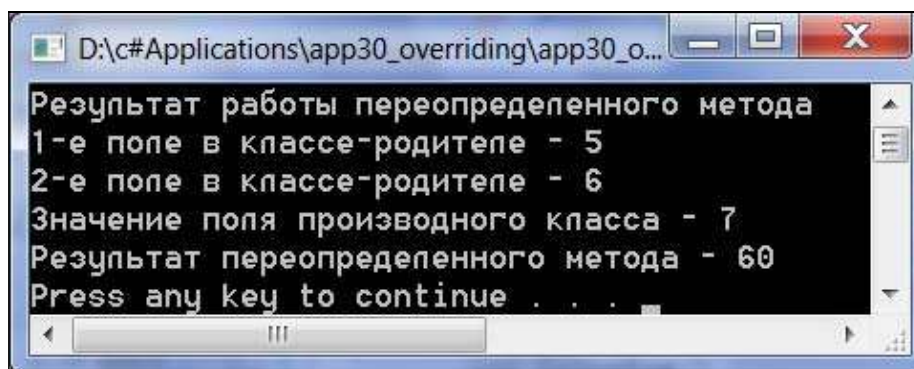


Рис. 8.10. Результат работы переопределенного в классе-потомке виртуального метода

Абстрактные классы

Допустим, вы разработали некий класс и не хотите, чтобы этим классом впоследствии пользовались непосредственно, т. е. создавали из него экземпляры (объекты), с которыми бы работали. Одна из причин — довольно общий характер его членов, который можно уточнять только на этапе разных подклассов, полученных путем наследования данного класса. Чтобы добиться запрета создания экземпляров, надо сообщить компилятору об этом. Это происходит путем придания классу атрибута `abstract`. Попытка создать экземпляр такого абстрактного класса приведет к ошибке компиляции. Но наследовать этот класс можно и нужно (иначе, для чего же он создавался?).

Вспомним, что при наследовании методы базового класса, помеченные ключевым словом `virtual`, могут переопределяться (но не обязательно) в дочернем классе. Если они переопределяются, то используется их новая функциональность, если нет — то функциональность базового класса. То есть метод в таких случаях может использоваться по умолчанию. В абстрактных классах виртуальные методы задаются лишь в том смысле, чтобы можно было просто к ним обратиться и в них что-то выполнялось. Например, рассмотренный нами ранее метод рисования `Draw()` в абстрактном классе может просто в своем теле выдать какое-то сообщение. Например, "Здравствуйте! Я — метод рисования из абстрактного класса". И все. Но наследующий класс может и не переопределять такой метод: не обязан. Просто разработчик что-то забыл и получает в момент отладки подобное "Здравствуйте" и не поймет в чем дело. То есть могут возникнуть неприятности еще на этапе разработки. Чтобы избежать подобных ситуаций, есть способ заставить разработчика переопределять метод класса-родителя в потомке. Для этого такой метод в родителе помечают ключевым словом `abstract` (как и абстрактный класс). Тут уже компилятор не пропустит эту ситуацию и напомним разработчику

выдачей соответствующей ошибки компиляции, что такой-то метод следует переопределить в классе-потомке. Методы с ключевым словом `abstract` могут определяться только в абстрактных классах. Вот пример:

```
abstract class Shape
{
    ...
    public abstract void Draw()
    {
        ...
    }
}
```

Соккрытие членов класса

Допустим, вы пользуетесь неким классом "втемную": получили его от некоего поставщика. Поэтому может наступить ситуация, когда вы в своем производном классе дали имя какой-то переменной, которое случайно совпало с именем, объявленным в используемом родительском (которым вы пользуетесь в данный момент) классе. В этом случае компилятор выдаст сообщение, что он прячет такую переменную в родительском классе (она вам становится недоступна при наследовании) и предложит либо дать своей совпавшей по имени переменной атрибут `override` (вполне естественно, как мы видели при переопределении методов) или добавить к ее объявлению ключевое слово `new` (видимо, даст ей свое другое имя). То есть, если вам нечего переопределять, добавьте `new`. Тогда ваша переменная может выглядеть так, например:

```
public new int indecs;
```

Здесь предполагается, что переменная с именем `indecs` была объявлена в классе, члены которого вы пытаетесь наследовать. Так заявляют некоторые авторы — специалисты по C#. Но жизнь, повторю, на месте не стоит, и компиляторы совершенствуются. Если взять программу листинга 8.13 и убрать в ней в определении метода `M_A()` ключевое слово `override`, а затем скомпилировать, компилятор действительно выдаст ошибку о том, что он прячет `M_A()`. Но ошибка выдается предупреждающая, а не фатальная. Это значит, что скомпилированную программу можно запускать на выполнение. Если это проделать, то окажется, что результат получим точно такой же, как показано на рис. 8.10. Похоже, что компилятор сам исправляет ситуацию. Что касается одинаковости имен переменных-неметодов, то тут никакого криминала компилятор не выявляет.

Приведение классов к базовому и производному

Тернарный условный оператор

Изучив принцип наследования, мы можем строить иерархию классов (расположение классов в порядке от высшего (родителя) к низшему (потомку)). Заметим, что все классы происходят от одного прародителя — абстрактного класса `Object`, в котором сосредоточены наиболее общие элементы, требующиеся для создания классов. Если при создании класса компилятор не находит родителя, он автоматически создает класс из класса `Object`. Ключевым словом типа этого класса является `object`. Если у нас есть некая иерархия классов, то язык C# позволяет тип класса-потомка связывать с типом класса-родителя (это так называемое *неявное приведение классов*) и, наоборот, тип класса-родителя связывать с типом класса-потомка (*явное приведение классов*). В листинге 8.14 представлен пример программы, в которой показаны оба вида преобразования. Результат работы программы приведен на рис. 8.11.

Листинг 8.14

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 05.12.2012  
 * Time: 14:03  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace app32_class_casting  
{  
    public class Base  
    {  
        public virtual void WhoAmI()  
        {  
            Console.WriteLine("I'm Base");  
        }  
    }  
    public class Derived: Base  
    {
```

```
public override void WhoAmI()
{
    Console.WriteLine("I'm Derived");
}
}
public class Test
{
    public static void Main()
    {
        Derived d = new Derived();
        Base b = d;    // Неявное преобразование ссылки
                       // в базовый тип
        b.WhoAmI();    // Проверка, что ссылка продолжает
                       // указывать на тот же объект
        Derived d1 = (Derived) b; // Восстановление ссылки
                               // на объект производного класса (явное
                               // преобразование)
        object o = d;    // Неявное преобразование ссылки
                        // на объект в тип object
        Derived d2 = (Derived) o; // Восстановление ссылки
                               // на объект производного класса (явное
                               // преобразование)
        d.WhoAmI(); // Проверка: восстановилась ли ссылка
                  // на объект
        Console.Read();
    }
}
```

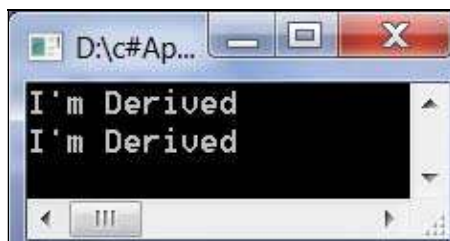


Рис. 8.11. Варианты неявного и явного приведения классов

Операторы *as* и *is*

Поскольку `object` является всеобщим базовым типом, любую ссылку на класс можно преобразовать в ссылку на `object`, а ссылку на `object` можно попытаться преобразовать в ссылку на любой класс. Слово "по-

пытаться" сказано не случайно, потому что явное преобразование (из потомка в родителя) проходит не всегда, т. к. потомок — это расширение родителя. Приведение "вниз" всегда выполняется явно. Существуют специальные операторы, которые определяют совместимость типов. Это операторы с ключевыми словами `as` и `is`. Они работают на совместимость не только таких сложных типов, как классы, но и для базовых типов, таких как, например, `int`. Как работает оператор `is`, видно из примера, приведенного в листинге 8.15.

Листинг 8.15

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 05.12.2012  
 * Time: 15:04  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace app33_as_and_is  
{  
    public class test  
    {  
        static void Main()  
        {  
            String derived_Obj = "Dummy"; // Производный объект  
                                           // класса String (из статических  
                                           // классов экземпляры не создаются)  
            Object base_Obj1 = new Object (); // Базовый объект  
            Object base_Obj2 = derived_Obj;   // Явное  
                                           // преобразование: запоминание  
                                           // ссылки в базовом объекте  
            Console.WriteLine ("base_Obj2 {0} String",  
                               base_Obj2 is String ? "является" : "не является");  
            Console.WriteLine ("base_Obj1 {0} String",  
                               base_Obj1 is String ? "является" : "не является");  
            Console.WriteLine ("derived_Obj {0} Object",  
                               derived_Obj is Object ? "является" : "не является");  
            int j = 123;  
            object b = j;
```

```
object obj = new Object ();
Console.WriteLine ("b {0} int",
    b is int ? "является" : "не является");
Console.WriteLine ("obj {0} int",
    obj is int ? "является" : "не является");
Console.WriteLine("b {0} System.ValueType",
    b is ValueType ? "является" : "не является");
float f=12.3f;
Console.WriteLine ("f {0} int",
    f is int ? "является" : "не является");
Console.Read();
}
}
}
```

Результат работы программы представлен на рис. 8.12.

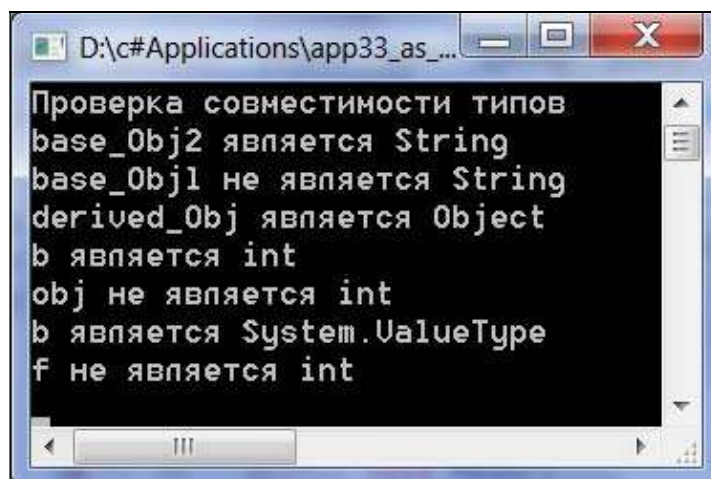


Рис. 8.12. Проверка работы оператора `is`

В программе создается экземпляр класса `String`. Это системный статический класс. Из таких классов экземпляры не создаются, поэтому переменной этого типа `derived_Obj` идет присвоение значения напрямую. Будем считать, что это — производный класс, т. к. все классы, в том числе и `String`, происходят от общего родителя — класса `Object`. Его и станем считать родителем. Его экземпляры — `base_Obj1` и `base_Obj2`. В последнем запоминается ссылка на производный класс `derived_Obj`. А далее следуют проверки с помощью оператора `is` на совместимость типов. Проверка осуществляется внутри оператора `Console.WriteLine()`, в котором в качестве выводимого на экран выражения служит выражение вида `base_Obj2 is String ? "является" : "не является"`. Такой

оператор мы еще не встречали. Его обозначение — вопросительный знак (?). Он называется *тернарным* (из трех частей) *условным оператором* в отличие от бинарного (из двух частей) условного оператора `if...else`. Общая форма такого оператора:

```
Выражение1 ? Выражение2 : Выражение3;
```

Здесь `Выражение1` должно относиться к типу `bool` (здесь и задается условие), а `Выражение2` и `Выражение3` — к одному и тому же типу. Обратите внимание на применение двоеточия и его местоположение в операторе ?. Результат определяется следующим образом. Сначала вычисляется `Выражение1`. Если оно истинно (`True`), то вычисляется `Выражение2`, и полученный результат — результат всего оператора. Если же `Выражение1` оказывается ложным (значение — `False`), то вычисляется `Выражение3`, и его значение становится общим результатом.

У нас в программе записано

```
base_Obj2 is String ? "является" : "не является"
```

То есть `Выражение1` — это `base_Obj2 is String` (проверка: является ли `base_Obj2` типом `String`). Это выражение можно было бы в программе записать в виде

```
bool bl = base_Obj2 is String;
```

Тогда выражение

```
base_Obj2 is String ? "является" : "не является"
```

было бы таким:

```
bl ? "является" : "не является"
```

В итоге оператор

```
Console.WriteLine("base_Obj2 {0} String", bl ? "является" :  
"не является");
```

работает так: выводится `base_Obj2` (то, что не относится к формату, выводится один к одному), затем обрабатывается формат, т. е. выбирается для вывода из списка аргументов первый (он у нас единственный). Это как раз тернарный условный оператор-выражение. Он вычисляется и выводится. После этого за форматом `{0}` снова идут неформатные символы (`String`). Они выводятся на экран без изменения.

Пример работы с оператором `is` показан в программе, приведенной в листинге 8.16. Результат ее работы представлен на рис. 8.13.

Листинг 8.16

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 05.12.2012  
 * Time: 16:02  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace App33_as_and_is2  
{  
    public class BaseType {}  
    public class DerivedType : BaseType {}  
    public class Program  
    {  
        static void Main()  
        {  
            Console.WriteLine ("Проверка оператора as");  
            DerivedType derived_Obj = new DerivedType ();  
            // объект производного класса  
            BaseType base_Obj1 = new BaseType (); // Объект  
                                                    // базового класса  
            BaseType base_Obj2 = derived_Obj; // Неявное  
                                              // преобразование в базовый класс  
  
            // Проверка, является ли базовый тип после  
            // преобразования в него производного производным  
            // типом. Если не является, то результатом as будет  
            // ссылка null  
            DerivedType derived_Obj2 = base_Obj2 as DerivedType;  
            if(derived_Obj2 != null)  
            { Console.WriteLine("Производный тип " +  
                                "преобразовался в базовый успешно");  
            }  
            else  
            { Console.WriteLine("Преобразование производного "+  
                                "типа в базовый не удалось");  
            }  
        }  
    }  
}
```

```
// Проверка: является ли базовый тип производным
derived_Obj2 = base_Obj1 as DerivedType;
if(derived_Obj2 != null)
{ Console.WriteLine("Базовый тип преобразовался "+
    "в производный успешно");
}
else
{ Console.WriteLine("Преобразование базового типа "+
    "в производный не удалось");
}

// Проверка: является ли производный тип базовым
BaseType base_Obj3 = derived_Obj as BaseType;
if (base_Obj3 != null)
{ Console.WriteLine("Производный тип совместим " +
    "с базовым");
}
else
{ Console.WriteLine("Производный тип не совместим "+
    "базовым");
}
Console.Read();
}
}
```

Оператор `as` в отличие от `is` при несовместимости типов выдает значение `null`.

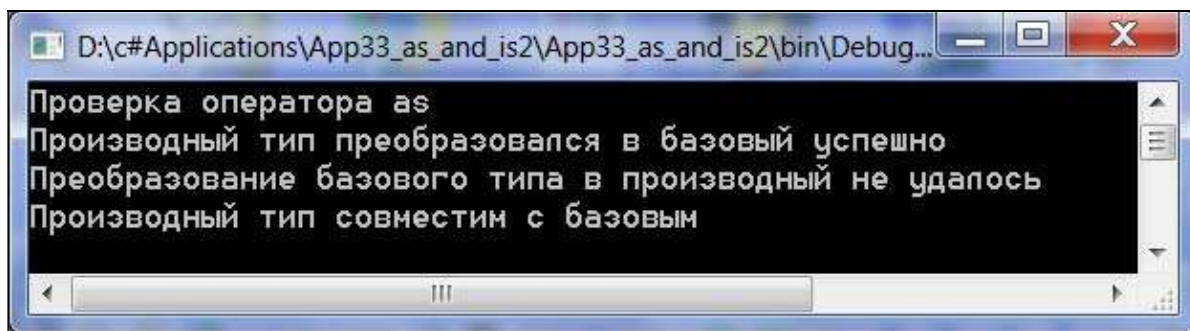


Рис. 8.13. Проверка работы оператора `as`

Структуры

Думается, что такая сущность в языке, как структура, являлась непосредственной предшественницей понятия классов. Действительно, идея объединить в один объект, в одну сущность данные различных типов появилась до классов, а в классе эта идея получила свое настоящее воплощение. Поэтому в таком языке, как C, структуры изучаются довольно подробно и с различными усовершенствованиями от одной версии языка к другой до изучения классов. Однако в C#, став на позицию класса, можно увидеть, что фактически структура — это частный случай класса, правда, с некоторыми особенностями.

Начнем с объявления структуры. Синтаксис объявления таков (покажем на примере):

```
public struct Person
{
    public int x, y;
    public Person(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

Ключевое слово `struct` определяет заданную сущность как структуру. После этого задается имя структуры (`Person`). Далее знакомые нам понятия: поля, конструктор. Но для структур имеют место следующие особенности.

- ❑ В отличие от класса структура не поддерживает наследования.
- ❑ В объявлении структуры поля не могут быть инициализированы до тех пор, пока они будут объявлены как постоянные или статические.
- ❑ Структура не может объявлять используемый по умолчанию конструктор (конструктор без параметров).
- ❑ Структуры копируются при присваивании. При присваивании структуры к новой переменной выполняется копирование всех данных, а любое изменение новой копии не влияет на данные в исходной копии. Вспомните, что у классов данные не перемещаются, а копируются только ссылки на данные. Отсюда видно, что структуры являются типами значений, а классы — ссылочными типами.

- ❑ В отличие от классов структуры можно создавать без использования оператора `new` (т. е. они могут не храниться в куче).
- ❑ Структуры могут объявлять конструкторы, имеющие параметры.

Пример программы, использующей структуру, приведен в листинге 8.17.

Листинг 8.17

```
/* Created by SharpDevelop.  
 * User: user  
 * Date: 07.12.2012  
 * Time: 12:34  
 *  
 * To change this template use Tools | Options | Coding |  
 * Edit Standard Headers. */  
using System;  
  
namespace app35_struct  
{  
    public struct Person  
    {  
        public int x, y;  
        int c; // Тип по умолчанию — private  
  
        public Person(int p1, int p2, int p3)  
        {  
            x = p1;  
            y = p2;  
            c = p3;  
        }  
        public int M(int a)  
        {  
            c=a;  
            return c;  
        }  
    }  
  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            Console.WriteLine("Работа со структурами");  
        }  
    }  
}
```