Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

C#

Объектно-ориентированный язык программирования

Пособие к практическим занятиям - №6

Проф. Забудский Е.И.

Москва 2005

Тема 6. Объектно-ориентированный подход к разработке программного обеспечения.

Наследование Часть I: Основные понятия.

Наследование — механизм, облегчающий повторное использование кода

Три практических занятия
(6 часов)

В восьми cs-программах (**листинги 6.1 – 6.8**) рассматриваются аспекты одной из важных концепций объектно-ориентированного программирования - **наследование**

За компонентно-ориентированным программированием – будущее
(см. **листинги 4.5 - 4.10** – Материалы к Практич. занятию № 4)

// **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).

Содержание

1. Наследование — механизм, облегчающий повторное использование кода

Важный аспект **объектно-ориентированного программирования — повторное использование кода** — позволяет не писать вновь и вновь фрагменты кода, уже написанные и отлаженные более опытными разработчиками.

Ранее были рассмотрены примеры повторного использования кода, в частности, компонентов библиотеки классов **.NET**. Один из приемов повторного использования кода, **агрегация**, был рассмотрен в примере **SimpleElevatorSimulation.cs** (см. Материалы к Практич. занятию № 3, **строки 12** и **28**). Напомним, что агрегация позволяет объединить в классе **разные типы**. В том примере класс **Elevator** состоял из трех элементов типа **int** и класса **Person** (ссылочный тип). Далее, (см. Материалы к Практич. занятию № 5), был приведен пример повторного использования класса **Array** из библиотеки классов **.NET** (**строка 058** программы **BankSimulation.cs**) для создания массива банковских счетов.

Наследование — еще один важный механизм ООП, облегчающий повторное использование кода. Наследование позволяет определить новый класс **расширением** существующего. **Производный** класс *наследует* элементы старого класса, При необходимости**,** можно указать различия между старым и новым классом, следующим образом:

1) добавив новые элементы (функции и данные) к производному классу;

2) изменив поведение унаследованных функций за счет их новых реализаций.

Итак, агрегация позволяет создавать новые классы из частей (**Автомобиль**-Car из **Двигатель**-Engine, **Колесо**-Wheel, **КоробкаПередач**-Gear-box**,** **Руль**-SteeringWheel и других классов).

Наследование же дает другую возможность – расширить описание класса **Car**-Автомобиль так, чтобы создать более специализированный класс, например **SportsCar**-СпортивныеМашины, указав разницу между **Car**-Автомобиль и **SportsCar**-СпортивныеМашины (лучшая подвеска, более мощный мотор и т. п.), сохраняя все общие компоненты (четыре колеса, руль, сиденья). Часто говорят, что класс **Car** содержит **Engine**-двигатель, **Wheel**-руль и т. д., а **SportsCar** является **Car**.

Понятие наследования является основой еще одной важной концепции ООП — **полиморфизма** — обеспечивающей доступ к объектам разных типов с помощью **одной переменной**. Это позволяет **одному имени функции** соответствовать **нескольким различным реализациям** на этапе исполнения.

2. Необходимость наследования

Греческий философ Аристотель был первым известным систематиком, который начал распределять по категориям окружающие его объекты.

Работая с наследованием, полезно смотреть на мир глазами систематика. Например, используя этот подход, можно разделить транспортные средства на три категории: наземные, воздушные и водные, как показано на **рис. 6.1**. Затем, двигаясь вниз по иерархии, категории можно сделать более специализированными.

Наиболее общее понятие **на вершине** иерархии описывается **минимальным набором признаков,** характерных для всех элементов иерархии. Например, любая машина, велосипед, сани могут иметь атрибуты:

- **максимальноеКоличествоПассажиров**-maximumNumberOfPassengers
- и **приближеннаяМаксимальнаяСкорость**-approximateMaxSpeed,

а также выполнять действия:

- **Движение Вперед**-MoveForward
- и **Остановка**-Stop.

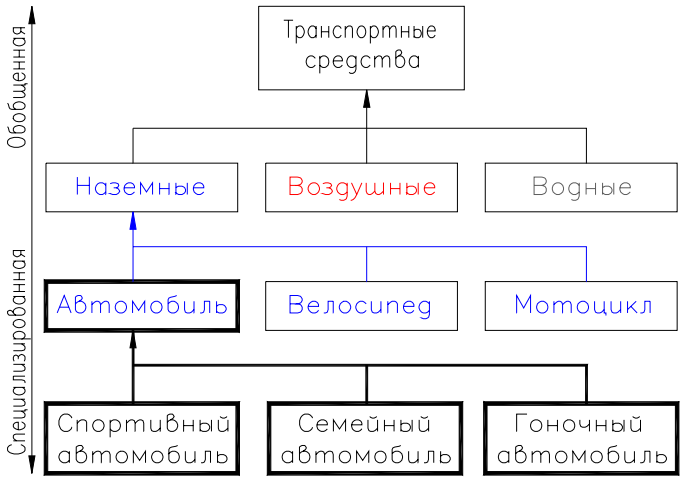


Рис. 6.1. Специализированная/обобщенная иерархия транспортных средств

По мере продвижения по иерархии вниз, к специализированным категориям, **добавляются более специфические атрибуты**. Например, характеристика автомобиля **рекомендуемыйРазмерШины**-recommendedTireSize неприменима ко всем наземным средствам, поскольку они включают в себя сани, которые обычно обходятся без шин.

В общем случае любые две категории, связанные стрелкой, удовлетворяют правилу: категория, из которой исходит стрелка, содержит те же самые атрибуты и действия, что и та, на которую она указывает, плюс дополнительные. Можно сказать, что категория, из которой исходит стрелка, является тем, на что стрелка указывает. Например, велосипед является наземным транспортным средством, а спортивный автомобиль — автомобилем. Обратное не справедливо. Наземное транспортное средство не обязательно велосипед, это может быть и автомобиль, и поезд, и что-нибудь еще.

// ПРИМЕЧАНИЕ

Может показаться, что стрелки на **рис. 6.1** показывают в неправильном направлении. Наземные транспортные средства, например, передают атрибуты и действия автомобилю. Не должно ли направление отражать этот факт? Возможно, но **по соглашению** стрелки повернуты в другую сторону по следующей причине: автомобилю присущи все атрибуты и действия, которые есть у наземного транспорта, но не наоборот.

3. Жизнь без наследования

Предположим, что в программе требуется создать объект с атрибутами и функциональностью **SportsCar**, а также объектов **FamilyCar** и **RacingCar** (все они принадлежат к категории автомобилей на **рис. 6.1**). Без наследования это можно сделать двумя способами:

1 подход. Все три типа объектов можно рассматривать как автомобили и сделать все атрибуты общими. Таким образом, все три типа автомобилей будут экземплярами одного класса **SportsFamilyRacingCar** (не лучшее имя).

2 подход. Можно поступить наоборот - создать три класса: **SportsCar, FamilyCar** и **RacingCar**.

Оба подхода можно реализовать программно. Но более детальный анализ свидетельствует, что у каждого из них есть серьезные недостатки, которые приводят к дилемме, а именно:

- элементы класса **SportsFamilyRacingCar**, общие для всех трех категорий автомобилей, хорошо обрабатываются при **первом** подходе, но вызывают проблемы при **втором**.
- элементы, уникальные для каждой категории автомобилей, вызывают проблемы при первом подходе, но естественным образом обрабатываются при **втором** - классы: **SportsCar, FamilyCar** и **RacingCar**.

Поэтому далее представлен **третий подход**, объединяющий лучшие стороны двух описанных выше и свободный от их недостатков.

4. Подход 3: Наследование. Проблемы разрешаются

При реализации концепции наследования необходимо сделать следующее:

1. Создать класс **Car,** содержащий **общие элементы**, которые характеризуют все экземпляры этого класса.
2. Создать три специализированных класса: **SportsCar, FamilyCar** и **RacingCar**, содержащие только характеристики, уникальные для каждого типа автомобилей.
3. Реализовать такую идею: помимо уникальных элементов каждый из специализированных классов должен содержать элементы класса **Car**.

Другими словами, требуется создать три **новых** класса, которые расширяют существующий класс **Car**. При этом **три** производных **класса наследуют свойства класса Car и в них добавляются уникальные свойства.**

Механизм наследования обеспечивает реализацию перечисленных пунктов 1, 2 и 3.

Наследование используется для создания объектов, имеющих как общие, так и особые свойства.

Перед тем как изучить концепцию наследования в C#, необходимо представить связанную с ней терминологию.

Когда класс **SportsCar** наследует элементы **Car**, говорят, что класс **SportsCar** — **производный** от **Car**, или **SportsCar** — **прямой подкласс** класса **Car**. Класс **Car** называется **базовым** (а также **надклассом** или **родительским** классом) класса **SportsCar**, поскольку он служит **основой** для **SportsCar.**

5. Основы наследования

Синтаксис для получения производного класса из базового показан в **синтаксическом блоке 6.1.** В заголовке производного класса указаны все необязательные спецификаторы, за которыми следует ключевое слово **class** и имя класса. Важно при этом двоеточие (**:**), за которым следует имя базового класса. Его смысл эквивалентен "является производным от" или "наследует от".

СИНТАКСИЧЕСКИЙ БЛОК 6.1. Определение класса с необязательным наследованием

Определение_класса_с_необязательным_наследованием::=

[<**Спецификатор_доступности**>] **class** <Имя_**производного**_класса> [**:** <Имя_**базового**_класса>]

{

<Элементы_**производного**_класса>

}

П р и м е ч а н и е

Двоеточие, за которым следует имя базового класса (**:** <Имя_**базового**_класса>) называется спецификацией базового класса.

Пример, в котором **SportsCar** произведен от **Car**:

```
class Car
{
<Элементы_класса_Car>
}
class SportsCar : Car
{
< Элементы _класса_SportsCar>
}
```

Пример с машинами развит в **листинге 6.1**. Кроме синтаксиса образования класса **RacingCar** (строки **29**—**49**) от **Car** (строки **06**—**27**) он показывает, почему наследование является хорошим решением проблем, возникающих при первом и втором подходах.

5.1. Листинг 6.1. Исходный код SimpleRacingCar.cs

```
01: using System;
02:
03:namespace Cons_Appl_SimpleRacingCar
04: {
05: /*************************************************************************/
06:     class Car  // базовый класс Car  - Автомобиль
07:     {                   // этот класс содержит элементы общие для всех автомобилей, а именно:
08:         private string brandName;              // 1. Переменная экз-ра brandName – имя марки
09:
10:         public string BrandName                // 2. Свойство
11:         {
12:             get
13:             {
14:                 return brandName;
15:             }
16:             set
17:             {
18:                 brandName = value;
19:             }
20:         }
21:
22:         public double CalculateFuelConsumptionPerKilometer()       // 3. Метод
23:         {
24:             Console.WriteLine("Теперь расчет потребления топлива");
25:             return 0.15;
26:         }
27:     }
28: /*************************************************************************/
29:     class RacingCar : Car                  // производный класс RacingCar - ГоночныйАвтомобиль
```

```
30:     {           // В этом классе описаны элементы характерные только для этого класса, а именно:
31:             private string highPressureFuelPumpSystem;          // 1. Переменная экз-ра
32:
33:             public string HighPressureFuelPumpSystem            // 2. Свойство
34:             {
35:                     get
36:                     {
37:                             return highPressureFuelPumpSystem;
38:                     }
39:                     set
40:                     {
41:                             highPressureFuelPumpSystem = value;
42:                     }
43:             }
44:
45:             public void StartOnBoardCamera()                    // 3. Метод
46:             {
47:                     Console.WriteLine("Теперь filming и передача изобр-ния");
48:             }
49:     }
50: /*****************************************************************************/
51:     class CarTester
52:     {
53:             public static void Main()
54:             {                                                   //  Созданы объекты:
55:                     Car myGeneralCar = new Car();               // 1) класса Car,
56:                     RacingCar yourRacingCar = new RacingCar(); // 2) класса RacingCar
57:                                     // вывод всех элементов класса Car (строки 58—61)
58:                     myGeneralCar.BrandName = "Volvo";
59:                     Console.WriteLine("Имя автом-ля общего класса: " + myGeneralCar.BrandName);
60:                     Console.WriteLine("Потреб-ние топлива автом-лем об-го класса: " +
61:                             myGeneralCar.CalculateFuelConsumptionPerKilometer());
62:             // использование свойства BrandName  и метода CalculateFuelConsumptionPerKilometer,
                                // описанных в классе Car, для объекта класса RacingCar (строки 63—66)
63:                     yourRacingCar.BrandName = "Ferrari";
64:                     Console.WriteLine("\n Имя гоноч-го автом-ля: " + yourRacingCar.BrandName);
65:                     Console.WriteLine("Потреб-ние топлива гон-ым автом-м: " +
66:                             yourRacingCar.CalculateFuelConsumptionPerKilometer());
67:                     yourRacingCar.HighPressureFuelPumpSystem = "MaxPressureLtd";
68:                     Console.WriteLine("Система топл-го насоса выс-го дав-ния: " +
69:                             yourRacingCar.HighPressureFuelPumpSystem);
70:                     yourRacingCar.StartOnBoardCamera();
71:                     Console.ReadLine();
72:             }
73:     }
74: }
```

Результаты работы программы

General car name - Имя автом-ля общего класса: **Volvo**
Now calculating the fuel consumption - Теперь расчет потребления топлива
Fuel consumption of normal car - Потреб-ние топлива автом-лем об-го класса: **0,15**

Racing car name - Имя гоночного автом-ля : **Ferrari**
Now calculating the fuel consumption - Теперь расчет потребления топлива
Fuel consumption of racing car - Потреб-ние топлива гон-ым автом-м: **0,15**
High pressure fuel pump system - Система топл-го насоса выс-го дав-ния: **MaxPressureLtd**
Now filming and transmitting pictures - Теперь **filming** и передача изобр-ния

Класс **Car** содержит элементы, общие для всех автомобилей. В данном случае это переменная **brandName** (строка **8**), свойство **BrandName** (строки **10**—**20**) и метод **CalculateFuelConsumptionPerKilometer** (строки **22**—**26**).

В определении класса **RacingCar** описаны только элементы, характерные для этого класса. Здесь это переменная **highPressureFuelPumpSystem** (строка **31**) с информацией о топливной системе и связанное с ней свойство **HighPressureFuelPumpSystem** (строки **33**— **43**). Наконец, в строках **45**—**48** добавлен метод **StartOnBoardCamera**.

В класс **RacingCar** должны входить и элементы класса **Car**. Вместо размещения их внутри **RacingCar** компилятору C# указывают, чтобы он включил их из класса **Car** (**двоеточие** и **имя класса Car** в строке **29**). Таким образом, класс **RacingCar** содержит как все элементы класса **Car**, так и элементы класса, описанные в его собственном блоке. Для проверки наследования применяется метод **Main** класса **ClassTester**. Он выводит все элементы класса **Car** (строки **58**—**61**) с помощью переменной **myGeneralCar**, объявленной в строке **55**. А переменная **yourRacingCar** класса **RacingCar** позволяет убедиться (в строках **63**—**66**), что можно использовать свойство **BrandName** (следовательно, и переменную **brandName**) и метод **CalculateFuelConsumptionPerKilometer**, которые описаны в классе **Car.**

6. Переопределение функций

В использованной иерархии метод **MoveForward** должен быть частью класса **Car**. Это вызывает проблему при порождении класса **RacingCar** (и двух других подклассов) от класса **Car**: **MoveForward** — один из методов, которые наследует **RacingCar**. К сожалению, реализация **MoveForward** в **Car** отличается от реализации этого метода в **RacingCar**. Вместо добавления одного километра к значению одометра, как в классе **Car** (предположим, что метод **MoveForward** именно так реализован в классе **Car**), требуется добавлять по тридцать **километров** в **RacingCar** (**пять** в **FamilyCar** и **двадцать** в **SportsCar**). **Для решения этой задачи требуется возможность переопределить реализацию метода MoveForward** в **Car для каждого подкласса**, как показано на **рис. 6.2.**

Рис. 6.2. Переопределение метода MoveForward класса Car

В **листинге 6.2** показана реализация переопределения функций (методов) в C#.

6.1. Листинг 6.2. Исходный код OverridingMoveForward.cs

```
01: using System;
02:
03: namespace ConsApplOverridingMoveForward
 04: {
05: /*************************************************************************/
06:     class Car
07:     {
08:         private uint odometer = 0;
09:
10:         protected uint Odometer   // свойство  ( protected – разрешает доступ к св-ву из
11:         {                                                    // производных классов)
12:             set
13:             {
14:                 odometer = value;
15:             }
16:             get
17:             {
18:                 return odometer;
19:             }
20:         }
21:
22:         public virtual void MoveForward()  // метод  ( virtual – это объяв-е обесп-т возм-ть
23:         {                                   // переопределения методов в произ-х классах)
24:             Console.Write("Перемещение вперед... ");
25:             odometer += 1;                                  // обращение к переменной
26:             Console.WriteLine("Показание одометра: {0}", odometer);
27:         }
28:     }
```

```
29: /*************************************************************************************/
30:     class RacingCar : Car
31:     {
32:             public override void MoveForward()       // метод  (override – м-д переопределяется)
33:             {
34:                     Console.Write("Быстрое перемещение вперед опасно... ");
35:                     Odometer += 30;                          // обращение к свойству
36:                     Console.WriteLine("Показ-е одометра в гоноч-м автом-ле: {0}", Odometer);
37:             }
38:     }
39: /*************************************************************************************/
40:     class FamilyCar : Car
41:     {
42:             public override void MoveForward()    // метод  (override – в произ-м классе переоп-й
43:             {                                     // метод должен иметь то же  имя, что и в базовом классе )
44:                     Console.Write("Медленное перемещение вперед не опасно...");
45:                     Odometer += 5;                           // обращение к свойству
46:                     Console.WriteLine("Показ-е одометра в семейном автом-ле: {0}", Odometer);
47:             }
48:     }
49: /*************************************************************************************/
50:     class CarTester
51:     {
52:             public static void Main()
53:             {
54:                     Car myCar = new Car();                                  // объект класса Car
54a:                    RacingCar myRacingCar = new RacingCar();  // объект класса RacingCar
55:                     FamilyCar myFamilyCar = new FamilyCar();    // объект класса FamilyCar
56:                     myCar.MoveForward();
56a:                    myRacingCar.MoveForward();
57:                     myFamilyCar.MoveForward();
58:                     Console.ReadLine();
59:             }
60:     }
61: }
```

Результаты работы программы

Moving forward - **Перемещение вперед...**

Odometer reading - **Показание одометра: 1**

Moving dangerously fast forward - **Быстрое перемещение вперед опасно...**

Odometer in racing car - **Показ-е одометра в гоноч-м автом-ле: 30**

Moving slowly but safely forward - **Медленное перемещение вперед не опасно...**

Odometer in family car - **Показ-е одометра в семейном автом-ле: 5**

Листинг 6.2 содержит класс **Car** (строки **06**—**28**) с методом **MoveForward**, который увеличивает значение переменной экземпляра **odometer** (объявленной в строке **8**) на **1** (в строке **25**).

Классы **RacingCar** (в строках **30—38**) и **FamilyCar** (строки **40—48**) содержат переопределения (строки **32—37** и **42—47**) метода **MoveForward**.

Чтобы обеспечить возможность определения метода **MoveForward** (определенного в строках **22—27**) класса **Car** в производных классах, он должен быть объявлен как **virtual** (строка **22**).

Если производный класс переопределяет метод (с ключевым словом **override**) базового класса, он должен содержать метод с тем же именем. Для этого метод **MoveForward** включен в классы **RacingCar** и **FamilyCar**. Число и тип параметров в переопределенном методе должно совпадать с методом базового класса. Об этом свидетельствуют заголовки в строках **32** и **42** (число параметров равно нулю).

Когда создается производный класс, метод с тем же именем и параметрами, что и в базовом классе, может быть определен случайно. Это значит, что планировалось создать новый метод, а не переопределить уже существующий. Программисту, читающему исходный код, важно понимать**:** **1)** создается ли новый метод или **2)** переопределяется метод базового класса.

При переопределении существующего метода компилятор выдает предупреждение. Чтобы избавиться от него, необходимо четко указать намерения программиста**:**

1. Если метод переопределяется, следует использовать ключевое слово **override**, как в строках **32** и **42**.

2. Если имя и параметры совпали случайно, следует применить ключевое слово **new**.

//ПРИМЕЧАНИЕ

Убрав ключевое слово **override** в строке **42**, можно увидеть следующее предупреждение компилятора:
OverridingMoveForward.cs(41,17): warning CS0114: 'FamilyCar.MoveForward()'
hides inherited member 'Car.MoveForward()'. To make the current method override that implementation, add the **override** keyword. Otherwise add the **new** keyword.

Ключевое слово **protected** в строке **10** — это спецификатор доступности, который имеет почти тот же смысл, что и спецификатор **private**, с одним отличием: он разрешает доступ из производных классов. Свойство **Odometer** объявлено как **protected**, поскольку переопределяющие методы **MoveForward** в производных классах требуют доступа к нему (строки **35**, **36**, **45** и **46**). В то же время, свойство **Odometer** недоступно из других классов.

Как и ожидалось, вывод примера, генерируемый методом **MoveForward** в строках **56a** и **57**, подтверждает, что вызываются два разных метода, принадлежащих объектам разных классов.

6.2. Синтаксис переопределения метода базового класса в производном

Синтаксический блок 6.2 описывает синтаксис переопределения метода базового класса в производном. Для корректного переопределения необходимо:

1. Разместить ключевое слово **virtual** (строка **22**) в заголовке метода базового класса (первая часть блока 6.2).

2. Разместить ключевое слово **override** (строка **32 и 42**) в заголовке переопределяющего метода производного класса (вторая часть блока 6.2).

Еще несколько важных моментов упомянуты в примечании к блоку 6.2.

СИНТАКСИЧЕСКИЙ БЛОК 6.2. Идентификатор базового класса

```
class <Идентификатор_базового_класса>
{
        …
        <Спецификатор_доступности> virtual <Тип_возвращаемого_значения>
         <Идентификатор_метода> ([<Список_формальных_параметров>])
        {
            <Операторы>
        }
        …
}
class <Идентификатор_производного_класса>
{
        …
        <Спецификатор_доступности> override <Тип_возвращаемого_значения>
        <Идентификатор_метода> ([<Список_формальных_параметров>])
        {
            <Операторы>
        }
}
```

где

<Спецификатор_доступности>::=
public
protected
internal
protected internal

П р и м е ч а н и я

1. Виртуальный метод не может быть объявлен как **private**.

2. Если метод в производном классе переопределяет метод базового класса, он должен иметь то же имя и параметры (число и последовательность типов должны совпадать), что и метод базового класса.

3. Переопределяющий метод должен иметь тот же спецификатор доступности и тип возвращаемого значения, что и переопределяемый метод базового класса.

4. Кроме нестатических методов, как **virtual** могут быть объявлены только нестатические свойства и нестатические индексаторы. Никакие другие функции-члены не могут быть виртуальными, а, значит, не могут быть переопределены.

//ВИРТУАЛЬНЫЕ ФУНКЦИИ

Чтобы не допустить переопределения функции, достаточно опустить ключевое слово **virtual**.

//ТЕРМИНАЛЬНЫЕ КЛАССЫ

Чтобы предотвратить использование класса в качестве базового, его необходимо объявить терминальным (с помощью ключевого слова **sealed**). К примеру, следующий класс не позволяет создавать производные классы:

```
sealed class MyMath
{
…
}
```

Поэтому следующее объявление класса будет некорректным:

```
class MoreMath : MyMath
{
...
}
```

Три причины, по которым следует объявлять класс как **sealed**:

1. Класс содержит только статические элементы — например класс **Math** в **.NET Framework.**

2. Устройство класса делает его непригодным для использования в качестве базового. Например, если внутренняя структура разработана настолько тонко, что производные классы могут привести к ошибкам. К этой категории принадлежит класс **String** в **.NET Framework.**

3. Невиртуальные функции и терминальные классы обеспечивают более высокую скорость исполнения, поскольку компилятор больше "знает" об их использовании.

7. Спецификаторы доступности и наследование

Спецификаторы **private** и **protected** особенно важны в связи с наследованием. Рассмотрим их подробнее. Также познакомимся с новыми спецификаторами доступности: **protected**, **internal** и **internal protected**.

7.1. Спецификатор доступности protected

Как отмечалось при анализе листинга 6.2, элемент класса, снабженный спецификатором **protected** (строка **10**), доступен только из своего собственного класса (как элемент с спецификатором **private**) и его подклассов. Объявления данных-членов как **protected** следует избегать, поскольку это нарушает принципы инкапсуляции, рассмотренные ранее (см. Материалы к Практич. занятию № 3, стр. 4-10). Об этой проблеме рассказано подробнее в следующем разделе.

7.2. Доступ к закрытым членам базового класса

Если свойство **Odometer** в классе **Car** (строка **10**) не содержит спецификатор **protected,** а содержит спецификатор **private** -

```
10:        private uint Odometer                    // свойство ( private) ,
```

то доступ к переменной **odometer** возможен только из класса **Car**, как в методе **MoveForward** (строки **25** и **26**). Однако элемент класса со спецификатором **private** недоступен извне того класса, в котором он объявлен. Таким образом, переменная **odometer** недоступна из производных классов, таких как **RacingCar**, несмотря на то, что она является одним из унаследованных элементов.

Однако все переменные экземпляра необходимо объявлять как **private**. Правильный способ доступа к закрытой переменной экземпляра базового класса из производного класса — **свойство** или **метод**, определенный в базовом классе и объявленный как **protected** (например, свойство **Odometer** в листинге 6.2, строка **10**), **internal** или **public**.

Почему же все переменные экземпляра следует объявлять **private**, а не **protected**, чтобы они не были доступны из подклассов? В конце концов, класс **RacingCar** — это **Car**, и ему тоже требуется доступ к переменной **odometer**. Почему же не сделать переменную экземпляра доступной из подклассов посредством ключевого слова **protected**? Проблема в следующем: если переменная экземпляра базового класса объявлена как **protected**, для доступа к ней достаточно создать производный класс. Такая простота доступа нарушает принцип инкапсуляции.

7.3. Спецификатор доступности internal protected

Вернемся к спецификатору **internal**, представленному в листинге 4.7 (см. Материалы к Практич. занятию №4, **стр**. **33, 34**). Он обеспечивает доступ ко всем классам внутри сборки. Спецификаторы **internal** и **protected** можно использовать вместе, как показано ниже:

```
internal protected int myNumber;
```

Теперь к **myNumber** возможен доступ из:

1) класса, где объявлена переменная;
2) производного класса;
3) класса, входящего в ту же сборку.

Таким образом, спецификатор **internal protected** обеспечивает **internal**- или **protected**-доступ.

//ПРИМЕЧАНИЕ

Спецификатор **internal protected** не обеспечивает **internal**-доступ и **protected**-доступ одновременно (то есть доступ из собственного класса и производного класса, принадлежащего той же сборке). Добиться этого можно, объявив базовый класс как **internal**, а элемент класса — как **protected**.

7.4. Обзор спецификаторов доступности C#

Спецификатор **internal protected** завершает знакомство со спецификаторами доступности в C#. Они собраны в табл. 6.1.

Таблица 6.1. Модификаторы доступа в C#

Спецификатор	Назначение
private	Доступ к элементу возможен только из типа, в котором он объявлен. Элементы класса и структуры имеют этот спецификатор по умолчанию.
public	Доступ к элементу возможен из любой части программы.
internal	Доступ к элементу возможен только из сборки, в которой он объявлен.
protected	Элемент класса доступен из класса, в котором он объявлен, и его производных классов.
internal protected	Элемент класса, обладающий этим спецификатором, доступен только из того класса, где он объявлен, из производного класса или из сборки, в которой он определен

8. Конструкторы производного класса

Независимо от спецификаторов доступности **конструкторы базового класса** никогда не наследуются производным. Несмотря на это, их можно вызывать из конструкторов **производного класса**, и в некоторых случаях компилятор неявно обеспечивает необходимый вызов конструктора базового класса.

//ПРИМЕЧАНИЕ

Подобно **кон**структорам, **де**структоры также не наследуются производным классом.

Возможна ситуация, когда конструктор **B** вызывается из конструктора **A** одного и того же класса. При этом операторы конструктора **B** исполняются до (то есть раньше) операторов конструктора **A**. Синтаксически такие функциональные возможности достигаются добавлением инициализатора конструктора

```
: this (<Список_аргументов>)                    // : this – это инициализатор конструктора
```

к заголовку, например:

```
class Dog
{
    ...
    public Dog  (string initialName)   : this()
    {                                          ↑
    ...                                        └── Инициализатор вызывает ...
    }
    ...
    public Dog()
    {
    ...   ←──────────────────────  ... этот конструктор до исполнения операторов
                                       собственного конструктора
    }
    ...
}
```

Аналогично можно вызвать конструктор, находящийся в **базовом** классе, из конструктора производного класса, используя инициализатор, показанный ниже:

```
: base (<Список_аргументов>)                    // : base – это инициализатор конструктора
```

Как и ранее, компилятор будет искать конструктор со списком формальных параметров, совпадающим с **<**Список_аргументов**>** по числу и типам аргументов. Но на этот раз поиск проводится в базовом, а не производном классе.

Листинг 6.3 служит иллюстрацией изложенного принципа, а также показывает, зачем может понадобиться вызывать конструктор базового класса из конструктора производного класса. Оба конструктора класса **RacingCar** используют инициализатор (строки **37** и **42**) для вызова конструктора из класса **Car**.

8.1. Листинг 6.3. Исходный код DerivedClassConstructors.cs

```
01: using System;
02:
03: namespace ConsAppl_DerivedClassConstructors
04:{
05: /*************************************************************/
06:     class Car
07:     {
08:             private string brandName;
09:                                     // Первый конст-р для иниц-ции пер-ной brandName с форм-м пар-м
10:             public Car(string initialBrandName)
11:             {
12:                     brandName = initialBrandName;
13:             }
14:                                 // Второй конст-р для иниц-ции пер-ной brandName без форм-го пар-ра
15:             public Car()
```

```
16:             {
17:                     brandName = "unknown";
18:             }
19:
20:             public string BrandName  // Свойство для проверки  значений пер-ной brandName
21:             {
22:                     get
23:                     {
24:                             return brandName;
25:                     }
26:                     set
27:                     {
28:                             brandName = value;
29:                     }
30:             }
31:     }
32: /*****************************************************************/
33:     class RacingCar : Car    // Класс RacingCar имеет две пер-ные экз-ра:1)  brandName, унаследованную
                                 // от класса Car, и 2) onBoardCameraName, объявленную в нем самом.
34:     {
35:             private string onBoardCameraName;
36:                                                     // Первый кон-р класса RacingCar:
37:             public RacingCar() : base() // 1) присв-т знач-е "unknown" пер-ной onBoardCameraName,
38:             {                           // 2) присв-т знач-е "unknown" пер-ной brandName (стр. 15-18)
39:                     onBoardCameraName = "unknown";
40:             }
41:                     // Второй кон-р класса RacingCar;  вызывается конструктор класса Car (стр. 10-13)
42:     public RacingCar(string initialBrandName, string initialCameraName) : base(initialBrandName)
43:             {
44:                     onBoardCameraName = initialCameraName;
45:             }
46:
47:             public string OnBoardCameraName
48:             {                                   // Св-во для пров-ки  знач-ий пер-ной onBoardCameraName
49:                     get
50:                     {
51:                             return onBoardCameraName;
52:                     }
53:                     set
54:                     {
55:                             onBoardCameraName = value;
56:                     }
57:             }
58:     }
59: /*****************************************************************/
60:     class CarTester
61:     {
62:             public static void Main()
```

```
63:         {
64:             Car myNoNameCar = new Car();
65:             Car myCar = new Car("Volvo");
66:             RacingCar myNoNameRacingCar = new RacingCar();
67:             RacingCar yourRacingCar = new RacingCar("Ferrari", "Sony");
68:             Console.WriteLine("The name of myNoNameCar: " +
69:                 myNoNameCar.BrandName);
70:             Console.WriteLine("The name of myCar: " + myCar.BrandName);
71:             Console.WriteLine("The name of myNoNameRacingCar: " +
72:                 myNoNameRacingCar.BrandName);
73:             Console.WriteLine("The camera name of myNoNameRacingCar: " +
74:                 myNoNameRacingCar.OnBoardCameraName);
75:             Console.WriteLine("The name of yourRacingCar: " +
76:                 yourRacingCar.BrandName);
77:             Console.WriteLine("The camera name of yourRacingCar: " +
78:                 yourRacingCar.OnBoardCameraName);
79:             Console.ReadLine();
80:         }
81:     }
82: }
```

Результаты работы программы

```
The name of myNoNameCar - Имя myNoNameCar: unknown
The name of myCar - Имя myCar: Volvo

The name of myNoNameCarRacingCar - Имя myNoNameCarRacingCar: unknown
The camera name of myNoNameCarRacingCar - Имя камеры myNoNameCarRacingCar: unknown

The name of yourRacingCar - Имя yourRacingCar: Ferrari
The camera name of yourRacingCar - Имя камеры yourRacingCar: Sony
```

Класс **Car** (строки **06—31**) содержит одну переменную экземпляра **brandName.** Для ее инициализации объявлено два конструктора. Первый (строки **10—13**) имеет один формальный параметр типа **string**, другой (строки **15—18**) не имеет аргументов. Свойства в строках **20—30** созданы специально для проверки значений **brandName.**

Класс **RacingCar** (строки **33—58**) является производным от **Car.** Он содержит две переменные экземпляра: **brandName,** унаследованную от класса **Car,** и **onBoardCameraName,** объявленную в нем самом. Обе переменные должны быть инициализированы. Переменную **onBoardCameraName** можно инициализировать при помощи конструктора обычным образом, однако с **brandName** ситуация осложняется, поскольку она объявлена как **private**. Более того, правильный код инициализации уже содержится в классе **Car,** поэтому его нужно использовать вновь. Для этого предназначен инициализатор конструктора, описанный ранее.

Первый конструктор класса **RacingCar** (строки **37—40**) **не имеет аргументов** и просто присваивает строку **"unknown"** переменной **onBoardCameraName.** А поскольку переменной **brandName** не присваивается никакого значения, вызывается конструктор по умолчанию (**без аргументов**) класса **Car** (строки **15—18**). Он присваивает переменной **brandName** значение **"unknown".**

Второй конст-р класса **RacingCar** (строки **42—45**) **имеет два аргумента**. Первый из них предназначен для **brandName**, второй — для **onBoardCameraName.** На этот раз вызывается

конструктор класса **Car**, **имеющий один аргумент**, поскольку известно значение (**initialBrandName**), которое должно быть присвоено **brandName** (конструктор базового класса **Car** присваивает значение **initialBrandName** переменной **brandName** (строка **12**)).

Метод **Main** класса **CarTester** проверяет конструкторы классов **Car** и **RacingCar**. Он использует свойства **BrandName** (строки **20—30**) и **OnBoardCameraName** (строки **47—57**) для доступа к переменным экземпляра **brandName** и **onBoardCameraName**.

Любой конструктор в производном классе, который не включает один из двух инициализаторов **:this**(<Список_аргументов>) или **:base**(<Список_аргументов>), получает инициализатор по умолчанию, присваиваемый компилятором. Первый конструктор класса **RacingCar** из листинга 6.3:

```
37:         public RacingCar() : base()
38:         {
39:                 onBoardCameraName = "unknown";
40:         }
```

идентичен следующему, в котором инициализатор отсутствует:

```
37:         public RacingCar()
38:         {
39:                 onBoardCameraName = "unknown";
40:         }
```

Компилятор добавляет инициализатор, даже если конструктор по умолчанию не был определен в базовом классе. В этом случае при компиляции выдается сообщение об ошибке.

//ПРИМЕЧАНИЕ

Конструктор по умолчанию — конструктор без формальных параметров.

Если в классе определен свой конструктор, компилятор не добавляет конструктор по умолчанию. Реализация конструктора возлагается на программиста.

//ПРИМЕЧАНИЕ

Конструктор может содержать только один инициализатор.

9. Индексаторы тоже наследуются и могут быть переопределены

Несмотря на то, что индексаторы, в отличие от методов и свойств, не имеют имен и вызываются по имени объекта, они тоже наследуются и могут быть переопределены. В листинге 6.4 показано, как индексаторы можно переопределить, а также вводится новое применение для ключевого слова **base** (не то, что используется в инициализаторе конструктора), позволяющее вызывать индексаторы базового класса.

9.1. Листинг 6.4. Исходный код SeasonalAdjustment.cs

```
01: using System;
02:
03: namespace ConsAppl_SeasonalAdjustment
04: {
05: /*********************************************************/
06:   class ProductionList //Кл. ProductionList пред-чен для пред-ния кварт-ных пок-лей пром. произ-ва
07:     {
08:             private uint [ ] production = new uint [ 4 ];
```

```
09:                     // Индексатор (строки 10—21). обеспечивает удобный доступ к массиву production
10:             public virtual uint this [uint index]
11:             {
12:                     get
13:                     {
14:                             return production[ index ];
15:                     }
16:
17:                     set
18:                     {
19:                             production[ index ] = value;
20:                     }
21:             }
22:     }
23: /*****************************************************/
24:     class ProductionSeasonAdjust : ProductionList
25:     { // Этот кл-с также пред-т пок-ли произ-ва, но возв-т значения, скорректированные с учетом сезона.
26:             public override uint this [uint index]
27:             {
28:                     get
29:                     {  // вызыв-ся индек-р баз-го кл-са (строки 10-21) и «index» передается как параметр.
30:                             if(index < 2)                   // кооррекция показателя производства
31:                                     return base[ index ] + 50;
32:                             else
33:                                     return base[ index ] - 30;
34:                     }    // base[ index ] – это конструкция базового доступа к индексатору (строки 10-21)
35:
36:                     set
37:                     {
38:                             base[ index ] = value;
39:                     }
40;             }
41;
42:             public uint TotalSeasonalAdjusted .
43:             {// это св-во сум-т знач-я индекс-ра в самом кл-се (так как они уже испр-ны с учетом сезона)
44:                     get
45:                     {
46:                             uint tempTotal = 0;
47:
48:                             for(uint i = 0; i < 4; i++)
49:                             {
50:                                     tempTotal += this[ i ];
51:                             }
52:                             return tempTotal;
53:                     }
54:             }
55:     }
56: /*****************************************************/
```

```
57:     class Tester
58:     {
59:             public static void Main()
60:             {
61:                     ProductionSeasonAdjust prodAdjust = new ProductionSeasonAdjust();
62:
63:                     prodAdjust[ 0 ] = 100;
64:                     prodAdjust[ 1 ] = 300;
65:                     prodAdjust[ 2 ] = 200;
66:                     prodAdjust[ 3 ] = 500;
67:                     Console.WriteLine("Production in first quarter season adjusted: " +
68:                             prodAdjust[ 0 ]);
69:                     Console.WriteLine("Total production seasonally adjusted: " +
70:                             prodAdjust.TotalSeasonalAdjusted);
71:                     Console.ReadLine();
72:             }
73:     }
74: }
```

Результаты работы программы

Production in first quarter season adjusted - Скоррек-ные показ-ли произ-ва в **1**-ом квартале сезона: **150**
Total production seasonally adjusted - Итоговые скорректированные показатели производства: **1140**

Класс **ProductionList** (строки **06**—**22**) предназначен для представления квартальных показателей промышленного производства. Класс содержит массив **production** (объявлен в строке **8**), удобный доступ к которому обеспечивает индексатор из строк **10**—**21**. Следует отметить, что он объявлен в строке **10** как **virtual**.

Требуется создать другой класс, который также представляет показатели производства, но возвращает значения, скорректированные с учетом сезона. Для простоты учет сезона в данном случае означает добавление **50** к значению из первого или второго квартала (индекс 0 или 1 в массиве **production**) или вычитание **30** — в ином случае. Кроме того необходимо включить свойство **TotalSeasonAdjusted**, которое возвращает сумму элементов массива **production** с учетом сезона.

Для всего этого создается новый класс **ProductionSeasonAdjust** (строки **24**—**55**), производный от **ProductionList**. Индексатор в строках **26**—**40** перекрывает индексатор из **ProductionList**. Если индекс меньше двух, возвращается элемент массива **production** плюс 50, если больше или равен 2 — элемент массива **production** минус 30. Здесь возникает проблема: несмотря на то что массив **production** унаследован в **ProductionSeasonAdjust**, в классе **ProductionList** он был объявлен как **private**, поэтому к нему нет прямого доступа из **ProductionSeasonAdjust**. Проблему можно решить, если вызвать индексатор базового класса, поскольку он имеет доступ к **production**. Именно для этого и предназначена конструкция базового доступа.

Конструкция базового доступа состоит из ключевого слова **base** и следующего за ним индекса, заключенного в квадратные скобки. Она должна быть размещена в элементе-функции производного класса. Конструкция базового доступа вызывает индексатор базового класса, который имеет индекс того же типа, что и индекс в конструкции. В данном примере блок (в строках **31** и **33**)

```
base[ index ]
```

вызывает индексатор базового класса в строках **10—21**, передавая **index** как параметр. В результате **base**[index] становится равным **production**[index] (в строке **14**). Таким образом, по правилам сезонного учета, к **base**[index] добавляется **50**, если index меньше **2**, и вычитается 30 в других случаях. За это отвечает фрагмент кода в строках **30—33**.

Свойство **TotalSesonalAdjusted** суммирует значения индексатора в самом классе (поскольку они уже исправлены с учетом сезона), выбирая их так:

this[**i**]

Обратите внимание на сходство с конструкцией базового доступа **base**[index]. В то время как **this**[**i**] вызывает индексатор из того же класса, **base**[index] вызывает индексатор базового класса.

В строке **63** метод **Main** присваивает значение **100** элементу **production** [0]. Затем оно выводится на экран в строках **67** и **68**, но с учетом сезонных поправок. Свойство **TotalSeasonalAdjusted** вычисляется как

(100 + 50) + (300 + 50) + (200 - 30) + (500 - 30) = **1140**

Как показано далее, конструкция базового доступа позволяет обращаться не только к индексаторам, но и к любым элементам базового класса.

10. Вызов переопределенной функции базового класса

Листинг 6.4 демонстрирует возможность вызова индексатора базового класса из производного посредством конструкции базового доступа. Подобным образом можно вызвать метод или свойство. Листинг 6.5 служит примером того, как метод базового класса можно вызвать из производного при помощи конструкции базового доступа, а также того, для чего такая конструкция может пригодиться.

10.1. Листинг 6.5. Исходный код CallingOverridenMethod.cs

```
01: using System;
02: // Иллюс-ция – м-д баз-го кл-са можно вызвать из произ-го при помощи конст-ции базового доступа
03: namespace ConsAppl_CallingOverridenMethod
04: {
05: /*********************************************************/
06:     class Car // базовый класс
07:     {
08:         private string brandName;
09:         private uint odometer;
10:
11:         public Car(string initialBrandName, uint initialOdometer)   // Первый конст-р класса Car
12:         {
13:             brandName = initialBrandName;
14:             odometer = initialOdometer;
15:         }
16:
17:         public Car()                                                 // Второй конст-р класса Car
18:         {
19:             brandName = "unknown";
20:             odometer = 0;
```

```
21:             }
22:
23:             public virtual void PrintAllInfo()
24:             {
25:                     Console.WriteLine("Brand name: " + brandName);
26:                     Console.WriteLine("Odometer: " + odometer);
27:             }
28:     }
29: /*****************************************************/
30:     class RacingCar : Car // производный класс содержит три перем-е экз-ра (строки 08, 09, 32)
31:     {
32:             private string onBoardCameraName;
33:                                                     // Первый конст-р класса RacingCar
34:             public RacingCar(string initialBrandName, uint initialOdometer,
35:                     string initialCameraName) : base(initialBrandName, initialOdometer)
36:             {
37:                     onBoardCameraName = initialCameraName;
38:             }
39:
40:             public RacingCar() : base()                     // Второй конст-р класса RacingCar
41:             {
42:                     onBoardCameraName = "unknown";
43:             }
44:
45:             public override void PrintAllInfo()
46:             {
47:                     base.PrintAllInfo();
48:                     Console.WriteLine("On board camera name: " + onBoardCameraName);
49:             }
50:     }
51: /*****************************************************/
52:     class CarTester
53:     {
54:             public static void Main()
55:             {
56:                     RacingCar yourRacingCar = new RacingCar("Lotus", 2000, "Nikon");
57:                     yourRacingCar.PrintAllInfo();
58:                     Console.ReadLine();
59:             }
60:     }
61: }
```

Результаты работы программы

Brand name - Название Марки: **Lotus**
Odometer - одометр: **2000**
On board camera name - Названия марки камеры: **Nikon**

Класс **RacingCar** (строки **30**—**50**) содержит три переменные экземпляра — две унас-

ледованные от класса **Car** (объявленные в строках **08** и **09**) и одну собственную — **onBoardCameraName**, объявленную в строке **32**.

Метод **PrintAllInfo** выводит на экран значения всех переменных экземпляра объекта. В классе **Car** это **brandName** и **odometer**, в **RacingCar** — **onBoardCameraName**, **brandName** и **odometer**. Поскольку требования к **PrintAllInfo** в **RacingCar** отличаются от **PrintAllInfo** в **Car**, этот метод нужно переопределить в производном классе. Теперь имеются две причины использовать конструкцию базового доступа для вызова **PrintAllInfo** класса **Car** из **PrintAllInfo** класса **RacingCar**. Во-первых, необходим доступ к переменным **brandName** и **odometer** из **PrintAllInfo** в классе **RacingCar**, но напрямую он невозможен, поскольку они обе объявлены как **private**. Во-вторых, **PrintAllInfo** из **Car** обеспечивает часть функциональных возможностей, требуемых от **PrintAllInfo** из **RacingCar** (он выводит значения двух из трех переменных). Итак, если каким-то образом вызвать метод **PrintAllInfo** класса **Car** из **PrintAllInfo** класса **RacingCar**, останется лишь вывести значение **onBoardCameraName**. Это действие выполняется в строках **45**—**49**. Оператор в строке **47** вызывает **PrintAllInfo** для базового класса **Car** при помощи ключевого слова **base** и следующего за ним имени **PrintAllInfo**. В метод **PrintAllInfo** добавлена строка **48** для вывода **onBoardCameraName.**

11. Синтаксис вызова метода, свойства или индексатора базового класса из производного посредством конструкции базового доступа

Синтаксис вызова: **1)** метода, **2)** свойства или **3)** индексатора базового класса из производного посредством конструкции базового доступа приведен в синтаксическом блоке 6.3.

СИНТАКСИЧЕСКИЙ БЛОК 6.3. Конструкция базового доступа

Базовый_доступ(метод)::=

base.<Имя_метода_базового_класса> **(**<Список_аргументов>**)**;

Базовый_доступ(свойство)::=

base. <Имя_свойства_базового_класса>;

Базовый_доступ(индексатор)::=

base [<Выражение1> *[*, <Выражение2>, <Выражение3>….*]*]

Примечания

1. <Список_аргументов> для Базовый_доступ(метод) должен соответствовать по числу и типам формальных параметров методу, вызываемому в базовом классе.

2. Список выражений для Базовый_доступ(индексатор) должен соответствовать по числу и типам индексатору базового класса.

3. Любая из трех конструкций базового доступа может находиться только внутри блока конструктора, метода экземпляра или **get**/**set**-аксессора свойства или индексатора.

12. Повторное использование библиотеки классов .NET Framework путем наследования

Многие классы в библиотеке **.NET Framework** разработаны специально для того, чтобы **создавать от них производные классы.** Класс **System.Windows.Forms.Form** (далее просто **Form**) является основным классом при написании графических **Windows**-приложений в **.NET**. Он используется как базовый, а все графические приложения содержат производные от него классы.

Листинг 6.6 показывает, насколько легко вывести окно, подобное представленному после листинга, при помощи наследования от класса **Form**. Несмотря на то, что полученное окно выглядит пустым, оно имеет все возможности стандартного окна (изменение размера, перетаскивание края, свертка, развертка, закрытие, перемещение по экрану и т. д.). Важно, что все это делает

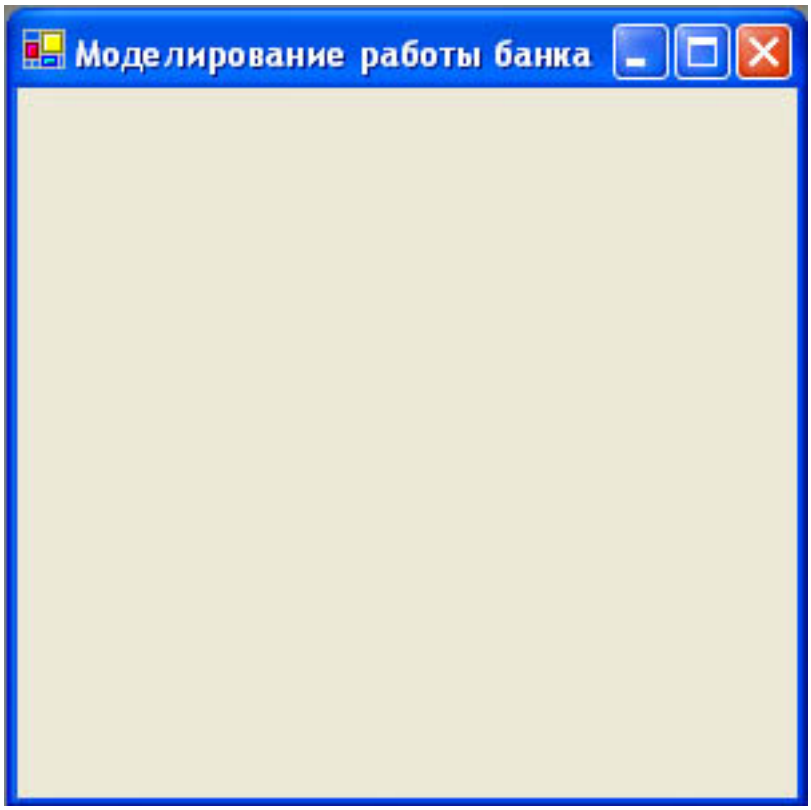
программа длиной всего в 19 строк.

//ПРИМЕЧАНИЕ

1. Класс **Form** находится в **dll**-библиотеке **System.Windows.Forms.dll,** которую автоматически подключает компилятор. Явно указывать ее не нужно (при реализации Windows.Application).
2. При реализации Console.Application нужно сделать ссылку: Project → Добавить ссылку → System.Windows.Forms.dll <OK>

12.1. Листинг 6.6. Исходный код MyFirstGUI.cs

```
01: using System.Windows.Forms;
02:
03: namespace ConsAppl_MyFirstGUI
04: {
05:
06:     public class frmMain : Form
07:     {
08:         public frmMain()
09:         {                                   // "Text" – унаследованное св-во класса Form
10:             this.Text = "Моделирование работы банка";
11:         }
12:
13:         public static void Main()
14:         {
15:             frmMain myFirstForm = new frmMain();
16:             Application.Run(myFirstForm);
17:         }
18:     }
19: }
```



Класс **Form** очень сложен. Он содержит множество функций и данных-членов, которые позволяют создавать и использовать полноценные окна. Класс **frmMain**, будучи производным от **Form**, наследует все эти возможности. Поэтому можно, например, написать конструктор (строки **08—11**), который присваивает текст "**Моделирование работы банка**" одному из унаследованных свойств — **Text**,— управляющему текстом, выводимым в заголовке окна.

В строке **15** создается экземпляр класса **frmMain** по имени **myFirstForm**, который затем передается методу **Run** как аргумент. **Run** — это статический метод класса **Application** в пространстве имен **System.Windows.Forms**. Передача **myFirstForm** методу **Run** приводит к появлению окна на экране и делает возможным его взаимодействие с такими событиями, как щелчки мышью по кнопкам и т. д.

13. Уровни производных классов

Производный класс может служить базовым для другого класса. Другими словами, число уровней наследования не ограничено. Например, можно создать обобщенный класс **TransportationVehicle** и образовать от него класс **SurfaceVehicle**. Затем, используя его как базовый, создать класс **Car**, который послужит базовым для классов **SportsCar**, **FamilyCar** и **RacingCar**, как показано на **рис. 6.3.**

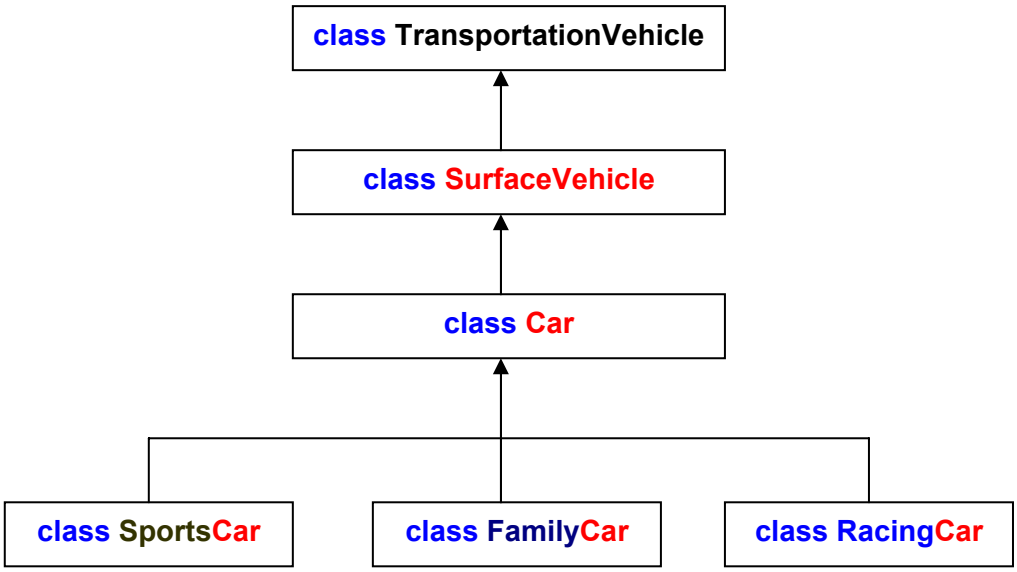


Рис. 6.3. Уровни наследования.

Листинг 6.7 показывает, как часть **рис. 6.3** может быть реализована при разработке трех классов **SurfaceVehicle**, **Car** и **FamilyCar**. Элементы классов при этом наследуются на нескольких уровнях.

13.1. Листинг 6.7. Исходный код ThreeInheritanceLevels.cs

```
001: using System;
002:
003: namespace ConsAppl_ThreeInheritanceLevels
004: {
005: /*********************************************************/
006:    class SurfaceVehicle
007:    {
008:          private float weight;
```

```
009:
010:        public SurfaceVehicle() // Первый конструктор класса SurfaceVehicle
011:        {
012:            weight = 0;
013:        }
014:
015:        public SurfaceVehicle(float initialWeight) // Второй конструктор класса SurfaceVehicle
016:        {
017:            Console.WriteLine("Now initializing weight");
018:            weight = initialWeight;
019:        }
020:
021:        public float Weight  // Свойство Weight
022:        {
023:            get
024:            {
025:                return weight;
026:            }
027:            set
028:            {
029:                weight = value;
030:            }
031:        }
032:    }
033: /*****************************************************/
034:    class Car : SurfaceVehicle
035:    {
036:        private uint odometer;
037:
038:        public Car() : base ()  // Первый конструктор класса Car

039:        {
040:            odometer = 0;
041:        }
042:
043:        public Car(uint initialOdometer, float initialWeight) : base(initialWeight)
044:        {                                                    // Второй конструктор класса Car
045:            Console.WriteLine("Now initializing odometer");
046:            odometer = initialOdometer;
047:        }
048:
049:        public uint Odometer // Свойство Odometer
050:        {
051:            get
052:            {
053:                return odometer;
054:            }
055:            set
```

```
056:                    {
057:                            odometer = value;
058:                    }
059:            }
060:    }
061: /*****************************************************************/
062:    class FamilyCar : Car
063:    {
064:            private byte numberOfBabySeats;
065:
066:            public FamilyCar() : base()   // Первый конструктор класса FamilyCar
067:            {
068:                    numberOfBabySeats = 0;
069:            }
070:                                          // Второй конструктор класса FamilyCar
071:            public FamilyCar(byte initialNumberOfBabySeats, uint initialOdometer,
072:                    float initialWeight) : base (initialOdometer, initialWeight)
073:            {
074:                    Console.WriteLine("Now initializing numberOfBabySeats");
075:                    numberOfBabySeats = initialNumberOfBabySeats;
076:            }
077:
078:            public byte NumberOfBabySeats  // Свойство NumberOfBabySeats
079:            {
080:                    get
081:                    {
082:                            return numberOfBabySeats;
083:                    }
084:                    set
085:                    {
086:                            numberOfBabySeats = value;
087:                    }
088:            }
089:    }
090: /*****************************************************************/
091:    class Tester
092:    {
093:            public static void Main()
094:            {
095:                    FamilyCar myCar = new FamilyCar(1,10000,1500);
096:
097:                    Console.WriteLine("\nWeight: {0}\nOdometer: {1}\nBaby Seats: {2}\n",
098:                            myCar.Weight, myCar.Odometer, myCar.NumberOfBabySeats);
099:                    myCar.Weight = 1800;
100:                    myCar.Odometer = 4000;
101:                    myCar.NumberOfBabySeats = 2;
102:                    Console.WriteLine("Weight: {0}\nOdometer: {1}\nBaby Seats: {2}",
103:                            myCar.Weight, myCar.Odometer, myCar.NumberOfBabySeats);
```

```
104:                    Console.ReadLine();
105:              }
106:    }
107: }
```

Результаты работы программы

```
Non initializing weight - Не инициализирована перем-я weight
Non initializing odometer - Не инициализирована перем-я odometer
Non initializing numberOfBabySeats - Не инициализирована перем-я numberOfBabySeats

Weight - Груз: 1500
Odometer - Одометр: 10000
Baby Seats - Детские Места 1

Weight - Груз: 1800
Odometer - Одометр: 4000
Baby Seats - Детские Места 2
```

В каждом из классов листинга 6.7 объявлена одна переменная экземпляра и связанное с ней свойство. Класс **Car** — производный от **SurfaceVehicle**,— наследует переменную **weight** и свойство **Weight**. Класс **FamilyCar** — производный от **Car**, и он наследует не только переменную **odometer** и свойство **Odometer**, объявленные в **Car**, но и два элемента, которые **Car** унаследовал от **SurfaceVehicle**. Это видно в методе **Main** класса **Tester**, где myCar (строки **99**—**103**) может использовать три свойства: **Weight** (и, как следствие, переменную экземпляра **weight**), **Odometer** и **NumberOfBabySeats**.

Конструкторы и класса **Car**, и класса **FamilyCar** используют инициализаторы для вызова конструкторов своих базовых классов (строки **38**, **43**, **66** и **72**). Например, при вызове конструктора **FamilyCar** (строки **71**—**76**) сначала вызывается конструктор **Car** (строки **43**—**47**). В свою очередь, конструктор **Car** вызовет конструктор **SurfaceVehicle** (строки **15**—**19**), в результате чего сначала будут исполнены операторы последнего, затем операторы конструктора **Car** и, наконец, операторы конструктора **FamilyCar**. Операторы вывода на экран в строках **17**, **45** и **74** позволяют наблюдать описанную последовательность событий. Первые три строки примера вывода являются результатом создания объекта **FamilyCar** в строке **95**.

Очевидно, насколько эффективно повторное использование кода в этих трех классах. **Weight** и **weight** повторно использованы не только в **Car**, но и в **FamilyCar**. Как правило, при крупных иерархиях классов код очень часто используется повторно.

// КЛАССЫ-ПРЕДКИ И КЛАССЫ-ПОТОМКИ

Иногда производный класс называют дочерним, а базовый — родительским. Распространяя эту логику на иерархическую цепочку, можно назвать производный класс классом-потомком, а базовый — классом-предком.

14. Переопределение и перегрузка используют разные механизмы

Важно различать перегрузку методов и переопределение методов. Как показано в синтаксическом блоке 6.2, переопределяющий метод должен иметь то же самое имя и список формальных параметров, что и виртуальный метод базового класса. А у перегруженного метода то же имя, **но другой список формальных параметров**.

Поэтому если включить в производный класс метод с тем же самым именем, что и метод в базовом классе, но с другим списком формальных параметров, метод базового класса не будет переопределен методом производного. Последний унаследует метод базового класса.

Важно заметить, что этот унаследованный метод базового класса не обрабатывается как метод, объявленный в производном классе. Рассмотрим ситуацию, когда и базовый класс, и производный от него содержат методы с одним и тем же именем (**MyMethod**), но разными формальными параметрами. Если метод **MyMethod** вызван из объекта производного класса, компилятор сначала попробует сопоставить аргументы вызова определениям метода **MyMethod** в производном классе, игнорируя на этом шаге **MyMethod**, унаследованный от базового класса. Если потребуется, он выполнит неявные преобразования типов. Если соответствие не будет найдено, будут рассмотрены методы, унаследованные от базового класса. Если в цепочке производных классов существуют методы **MyMethod** с разными формальными параметрами, компилятор будет подниматься по иерархии от класса, в котором произошел вызов. На каждом уровне он попытается обнаружить совпадение аргументов и формальных параметров.

Листинг 6.8 иллюстрирует описанный процесс. В нем содержится класс **Animal** с двумя методами **Move**. Класс **Dog**, производный от **Animal**, также содержит два метода **Move**. Количество и типы формальных параметров методов **Move** в классе **Animal** и классе **Dog** различаются. Методы не переопределяются, а наследуются (поэтому при объявлении метода **Move** в классе **Dog** не используется ключевое слово **override**).

14.1. Листинг 6.8 Исходный код OverridingOverloading.cs

```
01: using System;
02:
03: namespace ConsAppl_OverridingOverloading
04: {
05: /*************************************************************/
06:     public class Animal
07:     {
08:         public virtual void Move(short distance) // Первый метод Move класса Animal
09:         {
10:             Console.WriteLine("Animal.Move(short). Distance: {0}", distance);
11:         }
12:                                                  // Второй метод Move класса Animal
13:         public virtual void Move(double distance, string direction)
14:         {
15:             Console.WriteLine("Animal.Move(double, string). Distance: {0} Direction: {1}",
16:                 distance, direction);
17:         }
18:     }
19: /*************************************************************/
20:     public class Dog : Animal
21:     {
22:         public void Move(int distance)  // Первый метод Move класса Dog
23:         {
24:             Console.WriteLine("Dog.Move(int). Distance: {0}", distance);
25:         }
26:
27:         public void Move(byte distance) // Второй метод Move класса Dog
28:         {
29:             Console.WriteLine("Dog.Move(byte). Distance: {0}", distance);
```

```
30:             }
31:     }
32: /*****************************************************************/
33:     class Tester
34:     {
35:             public static void Main()
36:             {
37:                     Dog fido = new Dog();
38:
39:                     int myInt = 45;
40:                     short myShort = 25;
41:                     double myDouble = 5.6;
42:
43:                     fido.Move(myInt);
44:                     fido.Move(myDouble, "North");
45:                     fido.Move(myShort);
46:                     Console.ReadLine();
47:             }
48:     }
49: }
```

Результаты работы программы

```
Dog.Move(int).Distance: 45
Animal.Move(double, string).Distance: 5,6 Direction: North
Dog.Move(int).Distance: 25
```

Неудивительно, что при вызове метода **Move**(int) (строка 40) запускается метод класса **Dog**.

Компилятор не может найти метода **Move** в классе **Dog**, список параметров которого совпадал бы с комбинацией (**double, string**) (в строке **44**). Поэтому он ищет соответствия среди методов **Move**, унаследованных от класса **Animal** (где и находит метод **Move**(double..., string...)).

В строке **45** обнаруживается, что компилятор использует неявное преобразование типов аргументов до того, как перейти к методам, унаследованным от базового класса. Несмотря на то что единственный аргумент типа **short** соответствует формальному параметру в унаследованном методе **Move**(short...) (строки **8—11**), компилятор, применяя неявное преобразование, находит соответствие с методом **Move**(int...) (строки **22—25**).

Резюме

На этом занятии изучалась концепция наследования и ее реализация в C#.

При агрегации реализуется отношение "содержит", а при наследовании — "является".

Иерархия классов формируется по тем же принципам, что использовались философами-систематиками: от общих категорий к специализированным.

Посредством наследования члены более общих классов могут быть повторно использованы в более специализированных классах. Без наследования возможны два варианта создания объектов, имеющих общие и особенные свойства: **1)** можно создать один класс, охватывающий все отдельные объекты, или **2)** класс для каждого типа объектов. Оба подхода имеют серьезные недостатки. Наследование позволяет взять лучшее из этих подходов, устраняя присущие им проблемы.

Производный класс расширяет базовый класс, добавляя новые элементы и одновременно наследуя элементы базового класса. Последние можно изменять (в производном классе). Функция-член, объявленная **virtual** в базовом классе, может быть изменена в производном классе при помощи переопределения, задаваемого ключевым словом **override**.

Производные классы также называют *классами-потомками*, а базовые — *классами-предками*.

Чтобы явно указать используемый механизм компилятору, применяется: ключевое слово **override** — для переопределения, **new** — для создания нового элемента класса.

Класс не может быть производным от класса со спецификатором **sealed**.

Элемент класса, объявленный как **protected**, доступен только из своего класса и производных от него.

Несмотря на то, что производный класс наследует закрытые элементы своего базового класса, он не имеет прямого доступа к ним.

Спецификатор доступности **internal protected** обеспечивает доступ **internal** или **protected**.

Производный класс не наследует конструкторы базового класса, но они могут быть вызваны из инициализаторов. Компилятор автоматически присоединяет инициализатор (который называет конструктор базового класса по умолчанию) к любому конструктору производного класса, не содержащему явно заданного инициализатора.

Индексатор базового класса вызывается следующим образом:

base[indexer]

Элементы базового класса можно вызвать из производного при помощи ключевого слова **base**. Это называется базовым доступом.

***Наследование необходимо* для повторного использования многих частей библиотеки классов .NET.**

Переопределение *и* перегрузка метода — ***различные механизмы***. **Переопределяющий** метод в производном классе должен иметь ту же сигнатуру, тип возвращаемого значения и спецификатор доступности, что и *переопределяемый* метод в базовом классе. **Перегружающий** метод имеет то же имя, что и перегружаемый, но другой список формальных параметров.

Контрольные вопросы

1. Вы только что начали работу над двумя проектами и определили следующие классы для двух программ:

Программа 1: самолет, реактивные двигатели, крылья, фюзеляж, пассажирские сиденья, кабина.
Программа 2: человек, учащийся, работник, студент, выпускник, секретарь, уборщик, директор.
Какая программа извлечет большую пользу из наследования? Почему? Какой подход более приемлем для другой программы?

2. Рассмотрим иерархию классов.
В какой из классов вы включили бы каждый из перечисленных элементов:
а) переменная экземпляра **brandName** ("марка");
б) метод **Autodial** ("автонабор");
в) метод **Start** ("пуск");
г) метод **Tuning** ("настройка на волну");
д) переменная экземпляра **purchasePrice** ("цена").

3. Предположим, класс **Animal** содержит метод **Move**, объявленный как **public**. Если класс **Dog** является производным от класса **Animal**, можно ли вызвать метод **Move** для экземпляра класса **Dog** таким образом — **myDog.Move()**, если такой метод не определен в классе **Dog**?

4. Заголовок метода **Move** в классе **Animal**:
public void Move()
Можно ли переопределить этот метод в классе **Dog**? Почему?

5. Класс Animal содержит переменную name, объявленную как private. Возможен ли доступ к name из класса Dog? Это преимущество или недостаток?

6. Класс **Animal** содержит функцию, объявленную как **private**. Можно ли вызвать эту функцию из класса **Dog**?

7. Класс **Animal** содержит метод со следующим заголовком:
protected virtual void MoveADistance(int distance)

Ваш коллега написал следующий заголовок метода в производном классе **Dog**, чтобы переопределить метод **MoveADistance**:
public override int MoveADistance(double distance)
В этом заголовке есть несколько ошибок. Найдите и исправьте их.

8. Можете ли вы предотвратить использование класса в качестве базового? Если да, то как?

9. Почему обычно имеет смысл вызывать конструктор базового класса из конструктора класса производного?

10. Класс **Animal** из предыдущих вопросов содержит один конструктор с одним параметром типа int. Ваш коллега реализовал класс **Dog** с конструктором, который не содержит явно указанного инициализатора. Почему компилятор выдает сообщение об ошибке?

11. Класс **Animal** оснащен сложным алгоритмом, который возвращает показатель метаболизма для базовой клетки животного. Заголовок этого метода выглядит так:
public virtual double MetabolicRateCell()
После вычисления **MetabolicRateCell** для **Animal**, легко вычислить и **MetabolicCellRate** для **Dog**, просто добавляя 100 к полученному значению.
Нужно переопределить метод **MetabolicRateCell()** в классе **Dog**, создав его реализацию. Напишите код для метода, в котором используется значение, возвращаемое методом **MetabolicRateCell** базового класса.

12. Класс **Poodle** является производным от базового класса **Dog**. Содержит ли **Poodle** метод **Move**, первоначально определенный в классе **Animal**?

Упражнения по программированию

Каждое из упражнений **2 - 4** построено на коде предыдущего.

1. Создайте четыре класса: **ElectronicDevice, Radio, Computer и MobilePhone**. Пусть **ElectronicDevice** будет базовым классом для остальных трех и включает в свое определение три элемента:

- Закрытую переменную экземпляра **brandName** (типа **string**).
- Открытое свойство **BrandName** для обращения к **brandName**.
- Закрытую переменную экземпляра **isOn** (типа **bool**).

Включите два метода: **SwitchOn** (он выводит строку "**On**" и устанавливает **isOn** в **true**) и **SwitchOff** (выводит "**Off**" и устанавливает **isOn** в **false**). Три подкласса пока оставьте пустыми.

Напишите код, чтобы удостовериться, что даже пока три класса остаются пустыми, можно использовать унаследованные ими элементы.

2. Перекройте **SwitchOn** и **SwitchOff** в каждом подклассе, чтобы их действие отличалось от действия метода в **ElectronicDevice** тем, что они выводили бы имя устройства, которое включено или выключено. Например, при вызове **SwitchOn** для **Radio** на экран выводится:

On
Radio

а **isOn** принимает значение **true**. (Подсказка: используйте базовый доступ.)

3. Снабдите класс **ElectronicDevice** двумя конструкторами:

- Конструктором по умолчанию, устанавливающим значение **brandName** в "**unknown**" и **isOn** в **false**.
- Конструктором с одним аргументом типа **string**, используемым для инициализации **brandName** с одновременным присваиванием **isOn** значения **false**.

Добавьте следующие переменные экземпляра в программе:

- **currentFrequency** типа **double** к **Radio**;
- **internalMemory** типа **int** к **Computer**;
- **lastNumberDialed** типа **uint** к **MobilePhone**.

Добавьте по два конструктора в классы **Radio**, **Computer** и **MobilePhone**:

- Конструктор по умолчанию, который инициализирует не только значения переменных, объявленных в конкретном классе, но и переменные, унаследованные от базового класса (**brandName** и **isOn**).
- Конструктор, имеющий два аргумента — один для присваивания переменной, объявленной в классе (**currentFrequency**, **internalMemory** или **lastNumberDialed**),

и один для инициализации переменной **brandName**. (Подсказка: примените подходящие инициализаторы.)

4. Напишите класс **LaptopComputer**, производный от **Computer**. Объявите переменную экземпляра **maxBatteriLife** типа **uint** и два конструктора:

- ✓ Конструктор по умолчанию, который инициализирует значения **maxBatteriLife** и других элементов класса, унаследованных **LaptopComputer**.
- ✓ Конструктор, который передает начальные значения переменным **maxBatteriLife** и наследуемым элементам класса.

Список литературы

1. Микелсен Клаус. Язык программирования C#. Лекции и упражнения. Учебник: пер. с англ./ Клаус Микелсен –СПб.: ООО «ДиаСофтЮП», 2002. – 656 с.

2. Джо Майо. C#Builder. Быстрый старт. Пер. с англ. – М.: ООО «Бином-Пресс», 2005 г. – 384 с.

3. Основы Microsoft Visual Studio .NET 2003 / Пер. с англ. - М.: Издательско-торговый дом «Русская Редакция», 2003. – 464 с. Брайан Джонсон, Крэйт Скибо, Марк Янг.

4. Герберт Шилдт. Полный справочник по C# . / Пер. с англ./ Издательство: Вильямс, 2004 г. 752 с.

5. Чарльз Петцольд. Программирование в тональности C# / Пер. с англ. Издательство: Русская Редакция, 2004 г. - 512 с.

C# & .NET по шагам (Web-ресурс)

1 | 2 | 3 | 4

Шаг 1 - Разработка приложений в .NET (основы). (24.09.2001 - 2.3 Kb)
Шаг 2 - Как будет распространяться приложение (основы). (24.09.2001 - 3.8 Kb)
Шаг 3 - Нам нужен .Net Framework SDK. (24.09.2001 - 3.8 Kb)
Шаг 4 - Hello Word C#. (25.09.2001 - 2.4 Kb)
Шаг 5 - Hello Word VB. (25.09.2001 - 1.7 Kb)
Шаг 6 - Hello Word VC++. (25.09.2001 - 1.6 Kb)
Шаг 7 - Пространство имен. (26.09.2001 - 2.7 Kb)
Шаг 8 - Net ассемблер и дизассемблер. (26.09.2001 - 3.5 Kb)
Шаг 9 - Просмотр класса в EXE проекте ILDasm.exe. (26.09.2001 - 1.6 Kb)
Шаг 10 - Две основы Net. (27.09.2001 - 2 Kb)
Шаг 11 - Отладка. (27.09.2001 - 33 Kb)
Шаг 12 - ADO.NET (27.09.2001 - 10 Kb)
Шаг 13 - Попробуем OLEDB. (27.09.2001 - 6 Kb)
Шаг 14 - Типы данных - системные и языка программирования. (28.09.2001 - 3 Kb)
Шаг 15 - Windows Form. (28.09.2001 - 7 Kb)
Шаг 16 - Где взять редактор C#. (28.09.2001 - 21 Kb)
Шаг 17 - Избавляемся от консольного окна. (28.09.2001 - 9 Kb)
Шаг 18 - Создаем окно. (28.09.2001 - 6 Kb)
Шаг 19 - Добавляем меню. (28.09.2001 - 6 Kb)
Шаг 20 - Свойства (properties). (28.09.2001 - 3 Kb)
Шаг 21 - Обработка событий на форме. (30.09.2001 - 5 Kb)
Шаг 22 - Изменение размера формы. (30.09.2001 - 2 Kb)
Шаг 23 - Изменение положения формы. (30.09.2001 - 2 Kb)
Шаг 24 - Override. (30.09.2001 - 2 Kb)
Шаг 25 - Встраиваем элемент управления в окно. (30.09.2001 - 5 Kb)
Шаг 26 - Обработка сообщений элемента классом элемента. (30.09.2001 - 6 Kb)
Шаг 27 - Еще один редактор C#. (30.09.2001 - 30 Kb)
Шаг 28 - Создание меню подробнее. (01.10.2001 - 6 Kb)
Шаг 29 - Одномерные Массивы. (01.10.2001 - 3 Kb)
Шаг 30 - foreach. (01.10.2001 - 2 Kb)
Шаг 31 - Интерфейсы. (01.10.2001 - 3 Kb)
Шаг 32 - Коллекции. (01.10.2001 - 6 Kb)
Шаг 33 - Создаем обработчик событий меню. (01.10.2001 - 6 Kb)
Шаг 34 - Сохраняем данные в файл. (01.10.2001 - 7 Kb)
Шаг 35 - Добавляем строку состояния. (02.10.2001 - 5 Kb)
Шаг 36 - Панели на строке состояния. (02.10.2001 - 6 Kb)
Шаг 37 - Икона формы. (02.10.2001 - 9 Kb)
Шаг 38 - Диалог открытия файлов. (02.10.2001 - 14 Kb)
Шаг 39 - Отображаем картинку. (02.10.2001 - 12 Kb)
Шаг 40 - Создаем панель инструментов. (02.10.2001 - 6 Kb)
Шаг 41 - Net Classes первые вывод. (02.10.2001 - 6 Kb)
Шаг 42 - XML документация кода. (02.10.2001 - 6 Kb)
Шаг 43 - XML notepad. (02.10.2001 - 16 Kb)
Шаг 44 - Заголовок формы и пункт меню выход. (03.10.2001 - 4 Kb)
Шаг 45 - Создаем файл с ресурсами строк. (03.10.2001 - 5 Kb)

……..

1 | 2 | 3 | 4