Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

C#

Объектно-ориентированный язык программирования

Пособие к практическим занятиям - №7

Проф. Забудский Е.И.

Москва 2005

Тема 7. Объектно-ориентированный подход к разработке программного обеспечения.

Наследование Часть II:

Абстрактные функции. Полиморфизм. Интерфейсы

Наследование — механизм, облегчающий повторное использование кода

Основные этапы компьютерного моделирования реальных и концептуальных систем – с. 50
(вместо заключения)

Три практических занятия
(6 часов)

В cs-программах (**листинги 7.1 – 7.11**) рассматриваются аспекты одной из важных концепций объектно-ориентированного программирования - **полиморфизма**

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. Абстрактные функции. Полиморфизм. Интерфейсы

Наследование не только помогает рационально выстроить иерархию классов и повторно использовать их код (Практ. зан. №6), но и позволяет реализовать такую концепцию, как **полиморфизм**.

Применение полиморфизма к иерархии классов затрагивает один базовый класс (класс-предок) и несколько производных классов (классов-потомков).

Иногда полиморфизм применяется к нескольким классам, не имеющим **базового**, то есть общего предка. Для этого используется понятие интерфейса (оно помогает освободиться от иерархической структуры классов).

Перед тем как рассматривать полиморфизм, введем понятие абстрактной функции, важное для концепции полиморфизма применительно к иерархиям классов.

2. Абстрактные функции: абстрактные методы, свойства, индексаторы и классы

Класс **Car** (листинг 6.2, практ. зан. №6) содержал определение метода **MoveForward** (приведенное ниже), который увеличивал значение переменной **odometer** на единицу.

```
22:    public virtual void MoveForward()  // метод ( virtual – это объявление обеспечивает возможность
23:{                                       // переопределения методов в производных классах)
24:         Console.Write("Перемещение вперед... ");
25:         odometer += 1;
26:         Console.WriteLine("Показание одометра: {0}", odometer);
27:    }
28: }
```

Метод **MoveForward** может делать что угодно в классе **Car**, где он определен. Его реализация в этом классе рассчитана на последующее переопределение в классах **FamilyCar**, **SportsCar** и **RacingCar**, где известно его действие (**FamilyCar** движется медленно, **SportsCar**— со средней скоростью, а **RacingCar** — очень быстро).

Строго говоря, метод **MoveForward** обобщенному классу **Car** не нужен. В самом деле, зачем нужно измерять дистанцию (см. строку **25**) на которую перемещается не конкретный, а обобщенный автомобиль (**Car**). Ее необходимо измерять для конкретных автомобилей: **FamilyCar**, **SportsCar** и **RacingCar**.

Однако убрать метод **MoveForward** из класса **Car** и реализовать его только в специализированных классах нельзя, так как это идет вразрез с ранее представленной иерархией классов и, более того, мешает использовать важный механизм, называемый полиморфизмом.

В качестве решения C# позволяет создать **абстрактный** метод, используя ключевое слово **abstract**. Такой метод имеет только заголовок, но не содержит реализации. Вместо этого ему требуется, чтобы реализация содержалась в производных классах. В результате, можно отбросить ненужную реализацию **MoveForward** в классе **Car**, одновременно указав, что каждый производный класс должен содержать метод **MoveForward**.

Если **класс** содержит хотя бы один абстрактный метод, он и сам **должен быть объявлен abstract**. Экземпляр абстрактного класса создать нельзя, поскольку он содержит методы, не имеющие реализации.

Листинг 7.1 содержит пример того, как определяется **абстрактный метод** и **класс**. Исходный код практически идентичен листингу 6.2 (Прак. Зан. №6, с. 10), но на это раз метод **MoveForward** объявлен как **abstract** в строке **22**, а класс **Car** — в строке **06**.

2.1. Листинг 7.1 Исходный код AbstractMoveForward.cs

```
01: using System;   // Сравните эту c#-программу с программой OverridingMoveForward.cs
02:                 //       в листинге 6.2 (Практ. зан. №6, с. 10)
03: namespace ConsAppl_AbstractMoveForward
04: {
05: /*************************************************************************/
06:     abstract  class Car    // абстрактный класс не имеет экземпляров
07:     {
08:         private uint odometer = 0;
09:
10:         protected uint Odometer
11:         {
12:             set
13:             {
14:                 odometer = value;
15:             }
16:             get
17:             {
18:                 return odometer;
19:             }
20:         }
21:
22:         public abstract void MoveForward();  // абстрактный метод не содержит тела метода
23:     }                   // и завершается точкой с запятой после пары круглых скобок, то есть – ();
24: /*****************************************************************************/
25:     class RacingCar : Car
26:     {
27:         public override void MoveForward()
28:         {
29:             Console.Write("Быстрое перемещение вперед опасно... ");
30:             Odometer += 30;
31:             Console.WriteLine("Odometer in racing car: {0}", Odometer);
32:         }
33:     }
34: /*****************************************************************************/
35:     class FamilyCar : Car
36:     {
37:         public override void MoveForward()
38:         {
39:             Console.Write("Медленное перемещение вперед не опасно...");
40:             Odometer += 5;
41:             Console.WriteLine("Odometer in family car: {0}", Odometer);
42:         }
43:     }
44: /*************************************************************************/
45:     class CarTester
46:     {
47:         public static void Main()
48:         {
```

```
49:                 RacingCar myRacingCar = new RacingCar();
50:                 FamilyCar myFamilyCar = new FamilyCar();
51:                 myRacingCar.MoveForward();
52:                 myFamilyCar.MoveForward();
53:                 Console.ReadLine();
54:             }
55:     }
56: }
```

Результаты работы программы

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**

Использование ключевого слова **abstract** (в **строке 22**) с одновременной заменой тела метода на точку с запятой в конце строки требуется для того, чтобы сделать метод **MoveForward** **абстрактным**. Поскольку класс **Car** теперь содержит абстрактный метод, он тоже должен быть объявлен абстрактным, как показано в строке **06**.

Способ, которым абстрактный **метод** переопределяется в производном классе, аналогичен тому, который использовался для виртуальных **методов**.

Фактически, абстрактный метод неявным образом также объявлен **virtual**. Следовательно, никакой код в классах **RacingCar** и **FamilyCar** из листинга 6.2 (Прак. Зан. №6, с. 10) не требуется менять — вывод программы также не изменился. Все же, с объявлением **MoveForward** и **Car** **abstract** в исходный код вносится несколько важных изменений. Во-первых, невозможно создать экземпляр класса **Car**. Во-вторых, если бы метод **MoveForward** в классе **RacingCar** был не переопределен, а просто унаследован, класс **RacingCar** содержал бы абстрактный метод. Тогда и сам класс также следовало бы определить как абстрактный. Иногда несколько уровней в цепочке производных классов состоят из классов абстрактных. Это корректный прием, часто применяемый при разработке иерархии классов.

// ПРИМЕЧАНИЕ

Невозможность создать экземпляр абстрактного класса не мешает объявить соответствующую переменную. Например, абстрактный класс **Car** можно использовать для объявления

Car myCar;

Далее показано, что эта возможность необходима для использования полиморфизма (см. с. 8, сл.).

Неабстрактные классы, экземпляры которых можно породить, называются конкретными.

get / **set**-аксессоры свойств и индексаторов также можно объявить как **abstract**. Эффект будет почти тот же, что и для абстрактных методов. Синтаксис объявления: **1)** абстрактных методов, **2)** свойств, **3)** индексаторов и **4)** классов показан в синтаксическом блоке 7.1.

2.2. Синтаксический блок 7.1. Абстрактные метод, свойство, индексатор и класс

1) Абстрактный_**метод**: :-
[<Спецификаторы_метода>] **abstract** <Тип__возвращаемого_значения>
<Идентификатор__метода> ([<Список_формальных_параметров>]);

2) Абстрактное_**свойство**: : -
[<Спецификвторы_свопства>] **abstract** — <Тип__возвращаемого_значения>
<Идентификатор_свойства>
{

[**get;**]
[**set;**]
}

3) Абстрактный_ **индексатор** : : =
[<Спецификаторы_индексатора>] **abstract** [<Тип_возвращавмого_значения>] **this**
[<Список_параметров >]
{
[**get;**]
[**set;**]
}

4) Абстрактный_**класс** : : =
<Спецификаторы_класса> **abstract** <Имя_кпасса>
{
<Члены класса>
}

П р и м е ч а н и я

1. Абстрактный метод объявляется при помощи ключевого слова **abstract**. Он не содержит тела метода и завершается точкой с запятой после двух круглых скобок, между которыми заключен список формальных параметров. Например:

```
public abstract void MoveForward();
```

2. Для объявления абстрактного свойства в его заголовок добавляется ключевое слово **abstract**. Блоки **get** и **set** должны быть заменены на точку с запятой:

```
public abstract int MyProperty
{
get ;
set ;
}
```

Теперь **get**- и **set**- аксессоры стали абстрактными.

3. Абстрактные методы и аксессоры нельзя объявлять как **private, static** или **virtual.**

4. Абстрактные методы и аксессоры неявно объявлены как **virtual.**

5. Абстрактные методы и аксессоры могут находиться только внутри абстрактного класса.

6. Абстрактный класс может содержать и неабстрактные методы и аксессоры.

7. Невозможно породить экземпляр абстрактного класса. **NB**

8. Если класс является производным от класса, содержащего абстрактные методы или аксессоры, производный класс также становится абстрактным, если только все абстрактные методы и аксессоры, унаследованные от класса базового, не будут переопределены новыми реализациями.

9. Виртуальный метод (или аксессор) можно переопределить при помощи абстрактного метода (или аксессора) в производном классе. Чтобы сделать неабстрактным любой класс, производный от этого абстрактного, следует переопределить эти абстрактные методы и обеспечить для них новую реализацию.

К переопределению абстрактных методов и аксессоров применимы те же правила, что и к переопределению виртуальных.

У переопределяющего метода должны быть то же имя, тип возвращаемого значения и формальные параметры (число и тип), что и у абстрактного переопределяемого. Он должен также включать слово **override** и быть объявлен с тем же самым спецификатором доступности.

Чтобы сделать аксессор переопределяющим, в его заголовок нужно добавить ключевое слово **override**. Приведенное ниже свойство содержит переопределяющие **set**- и **get**-аксессоры:

```
protected override int Odometer
{
    set
    {
        <Операторы>
    }
    get
    {
        <Операторы>
    }
}
```

3. Полиморфизм

Этимология слова «полиморфизм» **:** от греческих **poly** - **много** и **morphē** – **форма**. То есть в широком смысле термин полиморфизм означает способность принимать несколько форм. Перед тем как рассматривать, что он означает в программировании (**!!!** – см. с. 9), сделаем несколько важных замечаний об иерархии наследования.

3.1. Объекты производных классов - объекты более чем одного типа

Рассмотрим цепочку классов на рис. 6.3 (Прак. Зан. №6, с. 26). Очевидно, что гоночный автомобиль — это автомобиль, автомобиль — это сухопутный транспорт, а сухопутный транспорт — это вид транспорта. Но обратное не справедливо, и автомобиль — это не обязательно гоночный автомобиль. Такая же логика поддерживается в программировании и, в частности, в C#. Объект типа **RacingCar** является объектом: **1)** типа **Car**, **2)** типа **SurfaceVehicle** и **3)** типа **TranspotrationVehicle**. Следовательно, переменная типа **Car** может содержать ссылку на объект типа **RacingCar**:

```
Car myCar;
myCar = new RacingCar();
```

Обратное НЕверно. В переменной типа **RacingCar** не может храниться объект типа **Car**, поскольку **Car** — не обязательно **RacingCar**. Таким образом, приведенный ниже код некорректен:

```
RacingCar  myRacingCar   // переменная myRacingCar имеет тип RacingCar
myRacingCar = new Car ();  // НЕверно, т. к. в перем-й типа RacingCar не может хран-я объект типа Car
```

Эта логика распространяется на всю иерархию: переменная **myCar** типа **Car** может хранить ссылку на объект **FamilyCar** или **SportsCar**, а переменная типа **TranspotrationVehicle** может указывать на объект любого из его производных типов.

Когда объект производного типа (**RacingCar**) присваивается переменной типа-предка (**Car**), как показано ниже:

```
Car myCar = new  RacingCar();
```

можно вызывать только те методы (свойства или индексаторы), имена которых определены для типа-предка через его переменную (**myCar**). Например, если класс **Car** содержит метод **MoveForward**, и его производный класс **RacingCar** кроме наследования этого метода также определяет свой собственный метод **StartOnBoardCamera**, то через **myCar** можно сделать только вызов:

```
myCar.MoveForward();
```

но не:

```
myCar.StartOnBoardCamera(); // НЕверно
```

, поскольку **StartOnBoardCamera()** не определен в **Car** (хотя и определен в классе **RacingCar**).

3.2. Динамическое связывание виртуальных методов и асессоров (get, set)

Продолжим рассмотрение примера с **Car** и **RacingCar** из предыдущего раздела:

```
Car myCar;
myCar = new RacingCar();
```

Предположим, что метод **MoveForward** из myCar вызывается так:

```
myCar.MoveForward();
```

Оба класса, **Car** и **RacingCar**, содержат реализацию метода **MoveForward**, поэтому среде исполнения необходимо сделать важный выбор:

вызвать ли реализацию **MoveForward** из класса **Car**, или из **RacingCar**.

Если **MoveForward** объявлен как **virtual** в классе **Car** и **override** в производном классе **RacingCar**, то будет вызван метод из класса **RacingCar**.

Таким образом, когда программа с объявлением

```
Car myCar;
```

будет компилироваться, компилятор не будет знать, объект какого типа хранится в переменной myCar: объект **Car**, **RacingCar**, **SportsCar** или **FamilyCar**, когда он встретит строку:

```
myCar.MoveForward();  .
```

Почему же можно исполнить:

1. реализацию **MoveForward** для класса **RacingCar**, если переменная myCar содержит объект **RacingCar**,
2. реализацию **MoveForward** для **SportsCar**, если она содержит объект **SportsCar**,
3. или реализацию для класса **Car**, если она содержит объект класса **Car**?

Для этого в C# используется механизм **динамического связывания**, или позднего связывания (или виртуальной диспетчеризации; все это одно и то же), названный так потому, что даже если известно имя вызываемого метода, в момент компиляции строки

```
myCar.MoveForward();
```

действительная реализация, о которой говорит это имя, **определяется динамически** (динамическое связывание) и **после компиляции** (позднее связывание) — **при исполнении программы**.

Рис. 7.1 дает представление о механизме динамического связывания. Например, если myCar присвоен объект **SportsCar**:

```
myCar = new SportsCar();
```

тогда строка

```
myCar.MoveForward();
```

приведет к исполнению версии метода **MoveForward** для класса **SportsCar**.

Значение слова полиморфизм — "возможность принимать множество форм".

В программировании же полиморфизм означает:

возможность метода [z.B. **MoveForward**()], реализованного для объекта указанного класса [**Car**, **RacingCar**, **SportsCar** или **FamilyCar**],

вызываться автоматически, используя одну переменную [переменная myCar типа **Car**]

для доступа к объектам разных классов [**Car**, **RacingCar**, **SportsCar** и **FamilyCar**] посредством механизма динамического связывания.

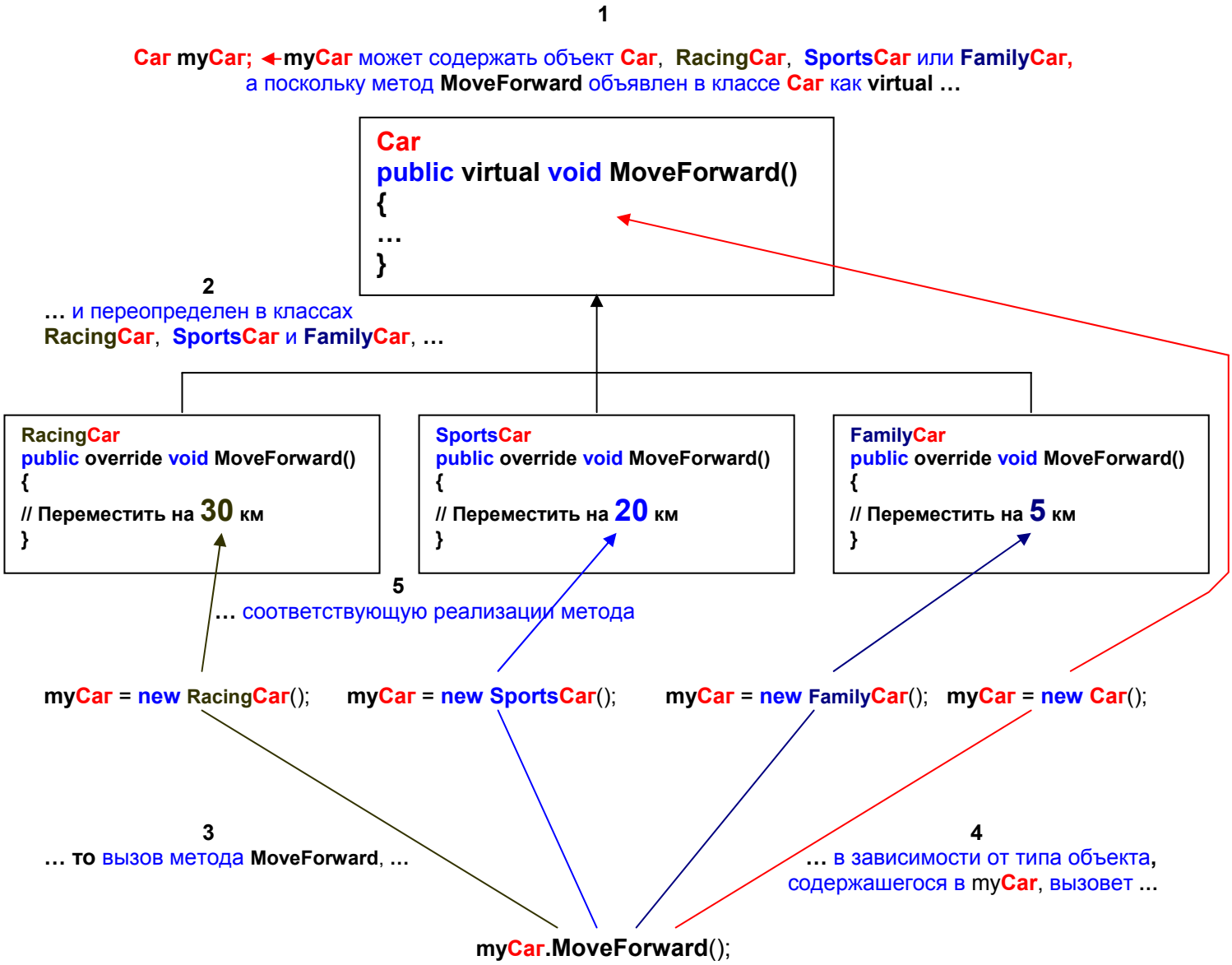


Рис. 7.1. Динамическое связывание (то есть программная реализация полиморфизма)

Таким образом, динамическое связывание и полиморфизм — разные названия одного и того же механизма. Динамическое связывание относится к самому процессу в вычислительной технике, а полиморфизм — концепция, используемая в абстрактных рассуждениях о классах и объектах.

// ПРИМЕЧАНИЕ

Чтобы считаться объектно-ориентированным, язык программирования должен по крайней мере поддерживать понятие классов и объектов вместе с инкапсуляцией, наследованием и полиморфизмом.

// ПРИМЕЧАНИЕ

Иногда перегрузку методов такте называет полиморфизмом, поскольку разные сигнатуры метода с одним именем позволяют запускать различные реализации. В большинстве случаев все же термин полиморфизм используется применительно к динамическому связыванию.

3.3. Пример использования полиморфизма в разработке простой программы черчения

Архитектурные чертежи можно рассматривать как наборы геометрических фигур — окружностей, квадратов, треугольников и т. п. Предположим, нужно написать архитектурную программу, чтобы архитектор мог создавать чертежи прямо на экране компьютера. Одна из основных ее задач — хранить чертеж и выводить его на экран по требованию. Большая часть этого примера посвящена поиску элегантного способа решения этой задачи и разработке упрощенной версии программы.

Вначале постараемся определить классы, которые понадобятся в программе. Для этого перефразируем предложения, выделяя существительные: Чертеж состоит из **набора фигур** (**Shape**). Фигура может быть: **1)** окружностью (**Circle**), **2)** прямоугольником (**Rectangle**), **3)** треугольником (**Triangle**), **4)** и т. д. Соответственно, нужно реализовать класс **Shape** и несколько подклассов. В данном случае достаточно ограничиться тремя подклассами:**Circle, Rectangle** и **Triangle**, как показано на рис. 7.2.

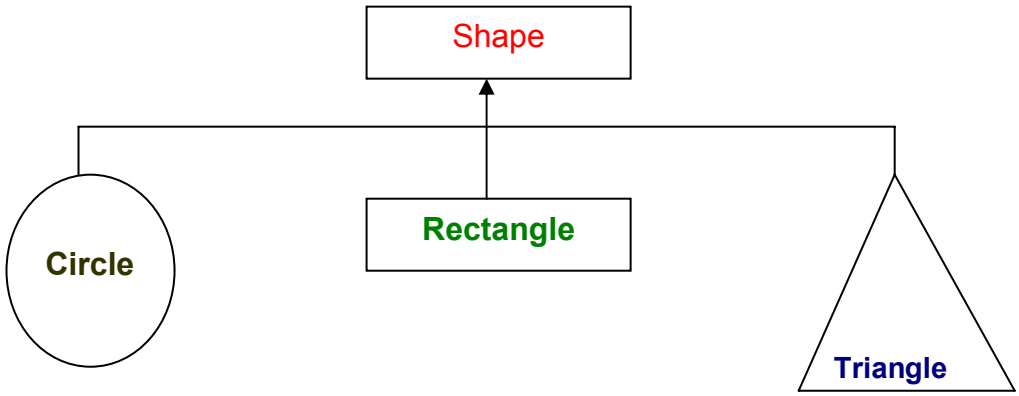


Рис. 7.2 Подклассы Circle, Rectangle и Triangle класса Shape

Помимо класса **Shape** и трех его подклассов потребуется реализовать и набор (коллекцию) фигур. Здесь можно воспользоваться уже знакомым массивом **ArrayList**.

// ПРИМЕЧАНИЕ

После объявления длина массива фиксированная. Это не подходит в данном случае, поскольку чертеж, как правило, состоит из неизвестного числа фигур. **ArrayList**, который может расти и уменьшаться динамически, представляет собой более удобную коллекцию. Однако выбор подходящей коллекции не влияет на демонстрацию свойств полиморфизма.

По мере того как архитектор добавляет фигуры к чертежу, в программе создаются соответствующие объекты **Shape** (в данном случае **Circle**, **Rectangle** и **Triangle**). Свойства каждого объекта **Shape** (положение на чертеже, размер, цвет и т. д.) сохраняются внутри него. Затем объект добавляется в массив, как показано на рис. 7.3.

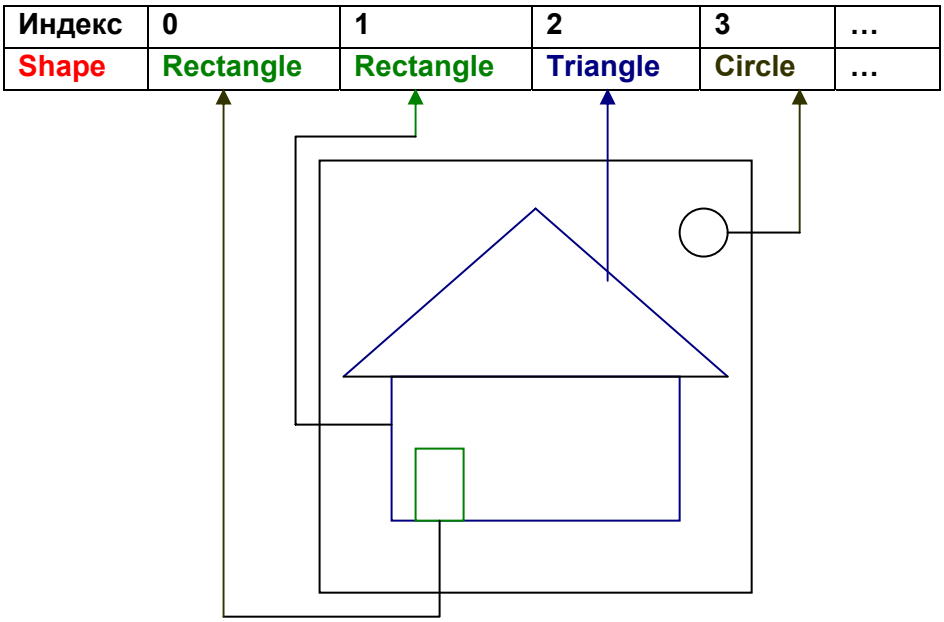


Рис. 7.3 Массив форм, представляющих чертеж

Так же разные объекты могут быть размешены в одном массиве, если все его элементы должны иметь один тип? Поскольку **Circle**, **Rectangle** и **Triangle**— потомки **Shape**, любой из объектов этих трех типов может быть помещен в переменную типа **Shape**, Следовательно, если объявить массив типа **Shape**, каждый элемент сможет хранить объект **Circle**, **Rectangle** или **Triangle**.

Кроме создания массива фигур требуется возможность рисовать набор объектов на экране. Чтобы нарисовать объект, можно было бы написать метод в отдельном классе **DrawingMachine** и назвать его **MakeDrawing**. Этот метод перебирал бы объекты в массиве, просматривал значения их переменных экземпляра и определял позицию, размер и форму объекта. Такой подход был бы неудобен, поскольку параметры и алгоритмы рисования объектов различны. Например, окружности нужен радиус и координаты центра, прямоугольнику — высота и ширина, а более замысловатым формам может понадобиться и большее число параметров. Здесь и стоит обратиться к полиморфизму.

Полиморфизм позволяет реализовать следующую методику: вместо того, чтобы разбираться извне, как следует рисовать каждый объект, стоит позволить объекту самому поддерживать свой алгоритм рисования. Каждый объект будет **инкапсулированным** пакетом, содержащим **данные** и **действия**, необходимые для его отображения. Назовем метод, выводящий объект на экран, **DrawYourself**. Он нужен объекту любой формы, поэтому его необходимо объявить в классе **Shape**. Однако точно так же, как была неизвестна реализация **MoveForward** для класса **Car**, здесь неизвестна реализации **DrawYourself** для **Shape**. Поэтому **DrawYourself** нужно объявить **abstract** в классе **Shape**, а затем включить его реализацию в каждом из трех подклассов.

Напомним, что абстрактный метод одновременно **виртуален**, и если переопределить **DrawYourself** в каждом классе, производном от **Shape**, сработает механизм **динамического связывания**. Это означает, что для рисования объектов достаточно просто вызывать метод **DrawYourself** каждого из них. Механизм динамического связывания определит, какого типа объект **DrawYourself**.

Листинг 7.2 содержит простую реализацию класса **Shape** (строки **06**—**09**) и трех его подклассов. Следует отметить, что **DrawYourself** объявлен **abstract** в классе **Shape** и переопределен в каждом из трех подклассов. Подклассы просты и не содержат информации о положении или размерах. Они просто выводят фигуру на экран.

3.4. Листинг 7.2. Исходный код ThreeShapes.cs

```
01: using System;
02:
03: namespace ConsAppl_ThreeShapes
04: {
05: /*************************************************************************/
06:     public abstract class Shape   // абстрактный класс
07:     {
08:         public abstract void DrawYourself();  // абстрактный метод: без тела и завершается  - ();
09:     }
10: /*************************************************************************/
11:     public class Triangle : Shape
12:     {
13:         public override void DrawYourself()
14:         {
15:             Console.WriteLine("       *     ");
16:             Console.WriteLine("      * *    ");
17:             Console.WriteLine("     *   *   ");
18:             Console.WriteLine("    *     *  ");
19:             Console.WriteLine("   *___ *  ");
20:         }
21:     }
22: /*************************************************************************/
23:     public class Circle : Shape
24:     {
25:         public override void DrawYourself()
```

```
26:             {
27:                     Console.WriteLine("       ***     ");
28:                     Console.WriteLine("    *       *  ");
29:                     Console.WriteLine("   *         * ");
30:                     Console.WriteLine("   *         * ");
31:                     Console.WriteLine("    *       *  ");
32:                     Console.WriteLine("       ***     ");
33:             }
34:     }
35: /*************************************************************************/
36:     public class Rectangle : Shape
37:     {
38:             public override void DrawYourself()
39:             {
40:                     Console.WriteLine(" ---------------------------- ");
41:                     Console.WriteLine("|____________________|");
42:             }
43:     }
44: /*************************************************************************/
45:     class ShapeTester
46:     {
47:             public static void Main()
48:             {
49:                     Console.WriteLine();
50:                     Console.WriteLine("Окружность");
51:                     Shape myShape;
52:                     myShape = new Circle();
53:                     myShape.DrawYourself();       // обращение к строке 25
54:                     Console.WriteLine();
55:                     Console.WriteLine("Треугольник");
56:                     myShape = new Triangle();
57:                     myShape.DrawYourself();       // обращение к строке 13
58:                     Console.WriteLine();
59:                     Console.WriteLine("Прямоугольник");
60:                     myShape = new Rectangle();
61:                     myShape.DrawYourself();       // обращение к строке 38
62:             Console.ReadLine();
63:             }
64:     }
65: }
```

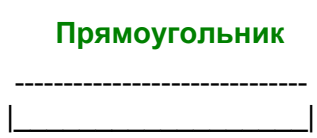
Результаты работы программы

Окружность

```
   ***
  *   *
 *     *
 *     *
  *   *
   ***
```

Треугольник

```
    *
   * *
  *   *
 *     *
*_______*
```



Объявленная в строке **51** переменная **myShape** принадлежит типу **Shape**. Поэтому **myShape** может ссылаться на объекты классов **Circle**, **Triangle** и **Rectangle**, производных от **Shape**.

Несмотря на то что строки **53**, **57** и **60** листинга 7.2 содержат один и тот же вызов **DrawYourself** из переменной **myShape**, благодаря механизму динамического связывания в каждой из них исполняется свои реализация метода **DrawYourself**. В строке **53** исполняется версия **DrawYourself** для **Circle** (поскольку **myShape** в этот момент содержит объект типа **Circle**). Вывод примера подтверждает это, поскольку первая изображенная фигура — окружность. Та же логика применима к операторам в строках **57** и **60**.

Простой тест листинге 7.2 знакомит с огромными возможностями полиморфизма. При вызове **myShape.DrawYourself()** программисту не приходится задумываться, какой объект хранится и **myShape**, и о других деталях. Полиморфизм позволяет рисовать чертеж на высоком уровне абстракции.

Окончательный вариант чертежной программы приведен в листинге 7.3.

3.5. Листинг 7.3. Исходный код TestDrawingEngine.cs

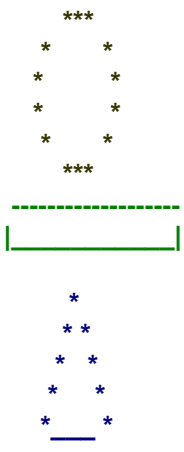
```
01: using System;
02:
03: namespace ConsAppl_TestDrawingEngine
04: {
05: /*********************************************************************/
06:     public abstract class Shape                 // абстрактный класс Shape
07:     {
08:         public abstract void DrawYourself();// абстрактный метод: без тела и завершается - ();
09:     }
10: /*********************************************************************/
11:     public class Triangle : Shape               // производный класс Triangle
12:     {
13:         public override void DrawYourself()
14:         {
15:             Console.WriteLine("       *     ");
16:             Console.WriteLine("      * *    ");
17:             Console.WriteLine("     *   *   ");
18:             Console.WriteLine("    *     *  ");
19:             Console.WriteLine("   *___ * ");
20:         }
21:     }
22: /*********************************************************************/
23:     public class Circle : Shape                 // производный класс Circle
24:     {
25:         public override void DrawYourself()
26:         {
27:             Console.WriteLine("       ***     ");
28:             Console.WriteLine("    *        *  ");
29:             Console.WriteLine("   *          * ");
30:             Console.WriteLine("   *          * ");
31:             Console.WriteLine("    *        *  ");
32:             Console.WriteLine("       ***     ");
33:         }
34:     }
```

```
35: /*************************************************************************/
36:     public class Rectangle : Shape                  // производный класс Rectangle
37:     {
38:             public override void DrawYourself()
39:             {
40:                     Console.WriteLine(" ---------------------------- ");
41:                     Console.WriteLine("|_____________________|");
42:             }
43:     }  // между строками 43 и 44 можно поместить еще один производный класс, z.B. – Rhombus (см. комментарий в стр. 71)
44: /*************************************************************************/
45:     public class DrawingEngine
46:     {
47:             public Shape [ ] CreateDrawing()
48:             {
49:                     int numberOfShapes = 0;         // заданное число отрисованных фигур
50:                     int shapeCounter = 0;           // текущее число отрисованных фигур
51:                     string choice;
52:                     Shape [ ] drawing;
53:
54:                     Console.Write("Сколько форм Вы хотите иметь в  чертеже? ");
55:                     numberOfShapes = Convert.ToInt32(Console.ReadLine());
56:                     drawing = new Shape[numberOfShapes];
57:                     do
58:                     {
59:                             Console.Write("Следующая выбранная форма: C)ircle R)ectangle T)riangle: ");
60:                             choice = Console.ReadLine().ToUpper();
61:                             switch(choice)
62:                             {
63:                                     case "C":
64:                                             drawing[shapeCounter] = new Circle();
65:                                             break;
66:                                     case "R":
67:                                             drawing[shapeCounter] = new Rectangle();
68:                                             break;
69:                                     case "T":
70:                                             drawing[shapeCounter] = new Triangle();
71:                                             break; // между строками 71 и 72 можно поместить: case … Rhombus …
72:                                     default:
73:                                             Console.WriteLine("Invalid choice");
74:                                             shapeCounter--;
75:                                             break;
76:                             }
77:                             shapeCounter++;
78:                     } while (shapeCounter < numberOfShapes);
79:                     return drawing;
80:             }
81:
82:             public void DrawDrawing(Shape [ ] drawing)
83:             {               // Этот метод демонстрирует преимущества использования полиморфизма
84:                     for(int i = 0; i < drawing.Length; i++)
85:                     {
86:                             drawing [ i ].DrawYourself();
87:                     }
88:             }
```

```
89:     }
90: /*****************************************************************/
91:     class TestDrawingEngine
92:     {
93:         public static void Main()
94:         {
95:             Shape [ ] myDrawing;
96:             DrawingEngine myCAD = new DrawingEngine();
97:             string choice;
98:
99:             do
100:            {
101:                Console.WriteLine("Пожалуйста, приготовьтесь для создания чертежа ");
102:                myDrawing = myCAD.CreateDrawing();
103:                do
104:                {
105:                    Console.WriteLine("Вот ваш красивый чертеж\n");
106:                    myCAD.DrawDrawing(myDrawing);
107:                    Console.Write("\nВы хотите видеть это снова? Y)es N)o ");
108:                    choice = Console.ReadLine().ToUpper();
109:                } while(choice != "N");
110:                Console.Write("Вы хотите создать другой чертеж? Y)es N)o ");
111:                choice = Console.ReadLine().ToUpper();
112:            } while(choice != "N");
113:        }
114:    }
115: }
```

Результаты работы программы

```
Пожалуйста, приготовьтесь для создания чертежа
Сколько форм Вы хотите иметь в  чертеже? 3
Следующая выбранная форма: C)ircle R)ectangle T)riangle: c
Следующая выбранная форма: C)ircle R)ectangle T)riangle: r
Следующая выбранная форма: C)ircle R)ectangle T)riangle: t
Вот ваш красивый чертеж
```



```
Вы хотите видеть это снова? Y)es N)o n
Вы хотите создать другой чертеж? Y)es N)o y
Пожалуйста, приготовьтесь для создания чертежа
Сколько форм Вы хотите иметь в  чертеже? 1
Следующая выбранная форма: C)ircle R)ectangle T)riangle: t
```

Вот ваш красивый чертеж

Назначение метода **CreateDrawing** (строки **47—80**) из листинга 7.3 — позволить пользователю создать любую последовательность объектов **Shape** и сохранить этот набор объектов в массиве **Shape**.

В строке **52** объявлен массив **drawing** типа **Shape**. Его элементам присваиваются объекты **Circle** (строка **64**), **Rectangle** (строка **67**) или **Triangle** (строка **70**). После заполнения массива ссылка на него возвращается в точку вызова в строке **79**.

Метод **DrawDrawing** (строки **82—88**) получает в аргументе массив **Shape** и выводит его содержимое на экран. Этот метод демонстрирует преимущества использования полиморфизма. Чтобы вывести каждую форму из массива **brawing**, методу **DrawDrawing** достаточно пройти по всем элементам, вызывая **DrawYourself** для каждого из них.

Студентам рекомендуется – распространить эту программу на любые другие классы **Shape**. Например, добавить класс **Octagon**, для этого достаточно сделать его производным от **Shape** и включить в него реализацию метода **DrawYourself**. Затем нужно добавить объект **Octagon** к чертежу, причем метод **DrawDrawing** остается при этом неизменным: он не зависит от того, с какой фигурой ему приходится иметь дело, главное, чтобы она была производной от класса **Shape** (т. е. имела свой метод **DrawYourself**).

В строке **102** метод **CreateDrawing** используется для создания нового массива фигур и присваивания его переменной **myDrawing**. Затем этот массив передается как аргумент методу **DrawDrawing** (строки **82…88**) в строке **106** для вывода фигур на экран.

Полиморфизм позволяет при разработке программы сосредоточиться на главном и поручить детали среде исполнения. Даже не зная конкретные типы группы объектов, можно давать команды этим объектам, и они будут исполнены надлежащим образом. Полиморфизм способствует расширяемости программы, поскольку полиморфные вызовы методов в определенной степени независимы от типов.

Наследование и полиморфизм коренным образом изменили написание программ.

Ниже приведено несколько примеров, когда стоит использовать полиморфизм.

Компьютерные игры (1-й пример – это тема курсовой работы - **NB**) содержат объекты различных типов, которые движутся, действуют и отображаются на экране: космические корабли, пришельцы, машины, монстры и т. д. Можно создать общий базовый класс для элементов игры, скажем, **GameElement**, который, наряду с другими методами, будет содержать абстрактный метод **DrawYourself**. Несмотря на то что набор содержит совершенно разные игровые элементы, каждый из которых рисуется на экране по-своему, для их вывода достаточно просто пройти по списку, вызывая метод **DrawYourself**.

База данных сотрудников (2-й пример - это тема курсовой работы - **NB**) включает данные о разных служащих: программистах, секретарях, маркетологах, уборщицах, директорах и т. д. Несмотря на различия, у них есть и общие атрибуты — имя, адрес, дата рождения, и методы — начисление зарплаты (**CalculateSalary**) и т. д. Можно создать базовый класс **Employee**, производными от которого будут остальные классы. Снабдив **Employee** общими атрибутами и методами (**CalculateSalary**), при начислении зарплаты достаточно пройти по всему списку, вызывая метод **CalculateSalary**. В итоге, не обязательно знать, кем именно является конкретный служащий, поскольку подходящий метод вызывается посредством динамического связывания.

4. Потеря и восстановление информации о типе

Если присвоить объект производного класса (**Rectangle**) переменной базового класса (**Shape**), как ниже:

```
Shape myShape = new Rectangle();
```

часть информации об объекте **Rectangle** будет утрачена. Теперь нельзя четко определить, относится ли **myShape** к типу **Rectangle**, **Circle** или **Triangle**.

Даже если известно, что **myShape** содержит **Rectangle**, попытка обратиться к свойству **Height** (предположим, что **Rectangle** содержит свойство **Height**) с помощью оператора:

```
myShape.Height = 20.4; // НЕверно
```

приведет к ошибке, поскольку в момент компиляции неизвестно, указывает ли **myShape** на объект типа **Rectangle**, **Triangle** или **Circle**. Если таковым окажется **Circle**, код будет некорректен, так как **Circle** не имеет свойства **Height** (а содержит свойство **Radius**). Таким образом, даже если переменная содержит объект подкласса со своими функциями, можно вызывать только функции, определенные в классе **Shape**.

Пока вызываются только те элементы класса **Rectangle**, которые определены и в **Shape**, ограничение в доступе **не** проявляется. В программе черчения, например, реальное содержимое элементов массива **drawing** типа **Shape** не требовалось, поскольку вызывался только метод **DrawYourself** (посредством динамического связывания).

Дело обстоит иначе, когда нужно вызвать функцию-член, специфичную для класса **Rectangle**. Например, пусть требуется подсчитать общую высоту всех прямоугольников в чертеже (предполагается, что **Rectangle** имеет свойство **Height**). Первая попытка

```
double totalHeight;
for(int i = 0; i < drawing.Length; i++)
{
totalHeight += drawing[ i ].Height;    //НЕверно
}
```

не дает результата, поскольку элементы **drawing** принадлежат типу **Shape** и не поддерживают обращения к **Height**. Если бы можно было четко определить, что в элементе хранится объект типа **Rectangle** и затем преобразовать переменную к этому типу, можно было бы и обратиться к свойству **Height**.

К счастью, C# следит за тем, на какой объект (и какого типа) указывает переменная в каждый момент времени. (Иначе не реализовать механизм динамического связывания.) Для доступа к этой информации в программе применяются операции **is** и **as**, представленные в следующих разделах.

4.1. Операция is

Операция **is** позволяет проверить, является ли переменная (например, **myShape**) указателем на объект определенного типа (например, **Rectangle**). Результат ее — логическое значение **true** или **false**. Чтобы проверить, содержит ли **myShape** объект **Rectangle**, используется логическое выражение:

```
(myShape is Rectangle)
```

которое возвращает **true**, если **myShape** указывает на **Rectangle**, и **false** — в противнрм случае.

Если в **myShape** хранится **Rectangle**, эту переменную можно преобразовать к типу **Rectangle** и обратиться к свойству **Height**. Для этого применяется операция приведения типа, описанная в следующем разделе.

4.1.1. Приведение типов объектов

Оператор присваивания

```
Car myCar = new FamilyCar();
```

показывает, что объект класса **FamilyCar** можно присвоить переменной **myCar** типа **Car**. Это возможно, поскольку класс **FamilyCar** — производный от класса **Car**, т. е. все элементы класса, доступные в **Car**, доступны и в **FamilyCar**. **Car** в иерархии выше, чем **FamilyCar**, и такое присваивание требует восходящего *приведения типа*, как показано на рис. 7.4

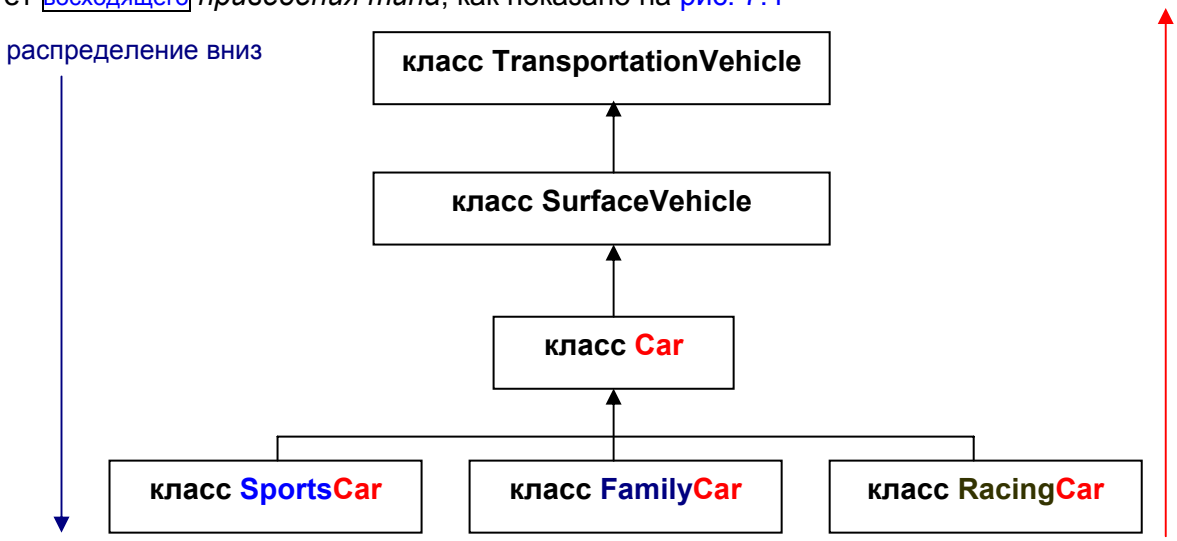


Рис. 7.4 Восходящее и нисходящее приведение типов

Движение в противоположном направлении, с приведением переменной к типу, расположенному в иерархии ниже, называется нисходящим *приведением типа*. Нисходящее приведение создает некоторые трудности. Например, нельзя использовать оператор вида

```
FamilyCar myFamilyCar = myCar; // НЕверно
```

поскольку **myCar** может также содержать **SportsCar** или **RacingCar**, а эти классы могут и не иметь элементов, к которым можно обращаться через **FamilyCar**. Тем не менее, если известно, что **myCar** фактически указывает на объект **FamilyCar**, можно воспользоваться явным приведением типа:

```
FamilyCar myFamilyCar = (FamilyCar) myCar;
```

// **ПРИМЕЧАНИЕ**

Если объект класса **A** приводится к классу **B**, причем **A** — потомок **B**, это называется **восходящим** приведением. При этом не требуется явной операции приведения типа.

Если же **A** — предок **B**, это называется **нисходящим** приведением. Нисходящее приведение всегда требует явной операции (<Тип>).

Используя операцию **is** (позволяющую узнать, указывает ли переменная на объект определенного типа) и операцию нисходящего приведения типа, можно решить проблему с **Rectangle** и вычислить сумму высот объектов этого типа в массиве **drawing** (см. код на с. 18, низ). Программа приведена в листинге 7.4

4.1.2. Листинг 7.4. Исходный код DownCastingRectangles.cs

```
01: using System;
02;
03: namespace ConsAppl_DownCastingRectangles
04: {
05:
06:     abstract public class Shape  // абстрактный класс Shape
07:     {
08:         public abstract void DrawYourself();   // абстрактный метод DrawYourself()
09:     }
10:
11:     public class Rectangle : Shape          // производный класс Rectangle
12:     {
13:         private double height;
14:
15:         public Rectangle(double initialHeight)
16:         {
17:             height = initialHeight;
18:         }
19:
20:         public override void DrawYourself()    // переопределяющий метод DrawYourself()
21:         {
22:             Console.WriteLine("Сделайте прямоугольником");
23:         }
24:
25:         public double Height
26:         {
27:             get
28:             {
29:                 return height;
30:             }
31:             set
32:             {
33:                 height = value;
34:             }
35:         }
36:     }
37:
38:     class Circle : Shape              // производный класс Circle
39:     {
40:         public override void DrawYourself()    // переопределяющий метод DrawYourself()
41:         {
42:             Console.WriteLine("Сделайте окружностью");
43:         }
44:     }
45:
46:     class Tester
47:     {
48:         private static Shape[ ] drawing;
49:
50:         public static void Main()
51:         {
52:             Rectangle myRectangle;
```

```
53:                 double totalHeight = 0;
54:                 drawing = new Shape[ 3 ];
55:
56:                 drawing[ 0 ] = new Rectangle(10.6);
57:                 drawing[ 1 ] = new Circle();
58:                 drawing[ 2 ] = new Rectangle(30.8);
59:
60:                 for(int i = 0; i < drawing.Length; i++)
61:                 {
62:                     if(drawing[ i ] is Rectangle) // Ссыл-ся ли элемент мас-ва с инд-ом i на объект Rectangle?
63:                     {
64:                         myRectangle = (Rectangle)drawing[ i ]; // нисходящее приведение типа
65:                         totalHeight += myRectangle.Height;
66:                     }
67:                 }
68:                 Console.WriteLine("Общая высота прямоугольников: {0}", totalHeight);
69:             Console.ReadLine();
70:             }
71:     }
72: }
```

Результаты работы программы

Общая высота прямоугольников: **41,4** (см. строки **56** и **58**)

Для сокращения исходного кода подкласс **Triangle** исключен, а каждый из переопределяющих методов **DrawYourself** содержит только вызов метода **WriteLine**. Естественно, это не влияет на то, как работают операция **is** и операция приведения в методе **Main** класса **Tester** (что и демонстрируется в этой программе).

В массив (строки **56**—**58**) записываются три объекта, два из которых принадлежат типу **Rectangle**. Счетчик **i** в цикле **for** (строки **60**—**67**) итерирует массив **drawing**. В строке **62** применяется операция **is**; смысл ее эквивалентен вопросу: "Ссылается ли элемент массива с индексом **i** на объект **Rectangle**?" При положительном ответе исполняются строки **64** и **65**. Поэтому в строке **64** возможно нисходящее приведение типа, которое позволяет обратиться к свойству **Height** и добавить его значение к **totalHeight** в строке **65**.

***//*СОВЕТ**

В общем случае выполнять восходящее приведение следует тогда, когда информация о конкретном типе уже не нужна программе (ведь она при этом теряется). В большинстве программ ни операция **is**, ни операция нисходящего приведения типа не требуются. Если подобные конструкции часто используются в программе, это свидетельствует о серьезных упущениях при ее разработке.

Тип элемента **drawing** проверяется не только операцией **is**, но и операцией приведения типа в строке **64**, хотя в этом и нет необходимости (проверка производится автоматически при каждом приведении). Это сопровождается излишними затратами вычислительных ресурсов, избежать которых позволяет операция **as**, рассматриваемая в следующем разделе.

4.2. Операция as

Операция **as** предназначена специально для нисходящего приведения. Она объединяет: **1)** операцию **is** проверки типа, **2)** оператор **if** и **3)** операцию приведения в одном действии, как показано в представленном ниже фрагменте (он может заменить следующие строки **60**—**67** листинга 7.4):

```
for(int i = 0; i < drawing.Length; i++)
{
        myRectangle = drawing[ i ] as Rectangle;
```

```
            if(myRectangle != null)   // строка 4
            {
            totalHeight += myRectangle.Height;
            }
}
```

Операция **as** между **drawing**[**i**] и **Rectangle** указывает, что элемент **drawing**[**i**] нужно привести к типу **Rectangle**. Если это возможно (когда **drawing**[**i**] содержит объект класса **Rectangle**), возвращается ссылка на объект **Rectangle**, которая присваивается переменной **myRectangle.** Если же **drawing**[**i**] не содержит объект **Rectangle**, то **myRectangle** присваивается значение null, возвращаемое операцией **as**. Таким образом, до обращения к свойству **Height** в my**Rectangle** достаточно проверить, что переменная не равна null (строка **4**).

5. Основной базовый класс: System.Object

Библиотека классов **.NET Framework** содержит класс **System.Object**, по отношению к которому все остальные классы являются производными. Это имеет два важных последствия. Во-первых, при создании класса, когда не указан базовый класс, C# автоматически использует в таком качестве **System.Object**. Во-вторых, любая цепочка производных классов всегда будет содержать верхний класс (например, **TransportVehicle** на рис. 6.3, см. Материалы к Практ. занят №6, с. 26), у которого нет базового. А значит, такой класс будет автоматически наследовать **System.Object**.

System.Object заслуживает подробного рассмотрения, поскольку любой класс наследует его элементы. Краткое представление о методах **System.Object** дает табл. 7.1. В первом столбце показано, как каждый из методов объявлен внутри **System.Object**.

5.1. Таблица 7.1. Методы System.Object

Объявление метода	Описание
public virtual string ToString()	Возвращает представление текущего объекта в виде строки, содержащей его имя пространства имен и имя класса. Это виртуальный метод, и его можно переопределить, т. е. возвратить любую строку по выбору программиста. **ToString()** часто переопределяют, чтобы получить информацию о состоянии объекта.
public virtual bool Equals(**object** obj)	Возвращает **true**, если текущий объект является тем же экземпляром, что и **obj**, иначе — **false**. Таким образом, **Equals** не сравнивает состояния двух объектов (сравнение по значению), а проверяет равенство ссылок. Зачастую **Equals** переопределяют в производном классе, чтобы получить сравнение по значению
public Type GetType()	Класс **Type**, возвращаемый методом **GetType**, обеспечивает доступ к метаданным типа текущего объекта (см. Материалы к Практ. занят №4 с. 17-20).
public virtual int GetHashTable()	Возвращает целочисленное представление объекта (**int**), называемое хэш-ключом. Он используется для быстрого доступа к объекту через специальные коллекции — хэш-таблицы.

protected object MemberwiseClone()	Возвращает ограниченную копию текущего объекта, т. е., если объект ссылается на другие объекты, копируются только ссылки, но не сами объекты.
public static bool Equals(**object** objA, **object** objB	Сравнивает объекты **objA** и **objB** по значению.
public static bool ReferenceEquals(**object** objA, **object** objB)	Возвращает **true**, если **objA** и **objB** ссылаются на один и тот же объект.
protected virtual void Finalize()	Этот метод называют также деструктором. Если переопределить его, как описано в [1, глава 13], то переопределяющий метод будет вызван во время "сборки мусора". На практике **Finalize** применяется редко.

Для использования функциональных возможностей методов из табл. 7.1, унаследованных любыми пользовательскими классами, достаточно вызывать их, как функцию-член (см. листинг 7.5).

5.2. Листинг 7.5. Исходный код SystemObjectTest.cs

```
01: using System;
02:
03: namespace ConsAppl_SystemObjectTest
04: {
05:
06:     namespace Animals
07:     {
08:         class Dog
09:         {
10:             private string name;
11:
12:             public Dog()
13:             {
14:                 name = "unknown";
15:             }
16:
17:             public Dog(string initialName)
18:             {
19:                 name = initialName;
20:             }
21:
22:             public string Name
23:             {
24:                 get
25:                 {
26:                 return name;
27:                 }
28:             }
29:         }
30:     }
31: /*************************************************************************/
32:     class ObjectTester
33:     {
34:         public static void Main()
```

```
35:         {                       // см. также Материалы к Практ. занят №4,  листинг 4.4, с. 19
36:             Type dogType;
37:             Animals.Dog myDog = new Animals.Dog("Fido");
38:             Animals.Dog yourDog = new Animals.Dog("Pluto");
39:             Animals.Dog sameDog = myDog;  // присвоение ссылки
40:
41:             Console.WriteLine("ToString(): " + myDog.ToString());
42:             dogType = myDog.GetType();
43:             Console.WriteLine("Type: " + dogType.ToString());
44:
45:             if(myDog.Equals(yourDog))  // метод Equals() сравнивает объекты
46:                 Console.WriteLine("myDog is referencing the same object as yourDog");
47:             else
48:                 Console.WriteLine("myDog и yourDog ссылаются на разные объекты ");
49:
50:             if(myDog.Equals(sameDog))
51:                 Console.WriteLine("myDog ссылается на тот же объект, что и sameDog ");
52:             else
53:                 Console.WriteLine("myDog and sameDog are referencing different objects");
54:
55:             if(object.ReferenceEquals(myDog, yourDog))
56:                 Console.WriteLine("myDog and yourDog are referencing the same object");
57:             else
58:                 Console.WriteLine("myDog и yourDog ссылаются на разные объекты ");
59:             Console.ReadLine();
60:         }
61:     }
62: }
```

Результаты работы программы

```
ToString(): ConsAppl_SystemObjectTest.Animals.Dog          |      (строка 41)
Type: ConsAppl_SystemObjectTest.Animals.Dog                |      (строка 43)
        myDog и yourDog ссылаются на разные объекты        |  (стр. 48) Equals():
        myDog ссылается на тот же объект, что и sameDog    |  (стр. 51) Equals():
                myDog и yourDog ссылаются на разные объекты  |  (стр. 58) object.ReferenceEquals():
```

В соответствии с таблицей 7.1, метод **ToString** возвращает строку вида:

<Имя_пространства_имен>.<Имя_класса>

Класс **Dog** содержится в пространстве имен **Animals**, поэтому вызов метода **ToString** в строке **41** должен давать:

Animals.Dog

что и подтверждается первой строкой вывода программы.

//ПРИМЕЧАНИЕ

Метод ToString() часто переопределяют в производном классе так, чтобы он возвращал строку, отражающую состояние объекта.

Класс **Type**, возвращаемый **myDog.GetType()**, позволяет обратиться к метаданным о классе **Dog**. В этом несложном примере их применение ограничено запросом строкового представления типа (строка **43**), идентичного возвращаемому значению метода **ToString()**.

В строке **45** проверяется, указывают ли **myDog** и **yourDog** на один и тот же объект.

Оператор в строке **39** присваивает переменной **sameDog** ссылку, хранящуюся в **myDog**, после чего **myDog** и **sameDog** указывают на один и тот же объект. Сравнение производится методом **Equals** в строке **50**.

C# содержит ключевое слово **object** — псевдоним **System.Object**. Как известно, при вызове статического метода нужно использовать имя класса. Это иллюстрируется строкой **55**, где вызов **object.ReferenceEquals** проверяет, указывают ли **myDog** и **yourDog** на один и тот же объект. Эта проверка аналогична производимой в строке **45**.

Как видно из табл. **7.1**, большинство методов класса **System.Object** объявлено как **virtual** и, таким образом, может быть переопределено. Листинг **7.6** показывает, как можно переопределить метод **ToString** (для получения информации о состоянии объекта) и **Equals** (для сравнения двух объектов по значению, а не по ссылке, как в листинге **7.5**).

//ПРИМЕЧАНИЕ

Любой класс, который переопределяет метод **Equals** (например, **Dog** в листинге **7.6**), должен переопределять и метод **GetHashCode**, чтобы тот генерировал уникальное числовое значение, идентифицирующее объекты в хэш-таблице. Для сокращения кода метод **GetHashCode** не приведен. Поэтому при попытке компилировать листинг **7.6** будет выдано предупреждение.

5.3. Листинг 7.6. Исходный код ObjectOverrideTest.cs

```
01: using System;
02:
03: namespace ConsAppl_ObjectOverrideTest
04: {
05:
06:     namespace Animals
07:     {
08:         class Dog
09:         {
10:             private string name;
11:
12:             public Dog()
13:             {
14:                 name = "unknown";
15:             }
16:
17:             public Dog(string initialName)
18:             {
19:                 name = initialName;
20:             }
21:
22:             public string Name
23:             {
24:                 get
25:                 {
26:                     return name;
27:                 }
28:             }
29:
30:             public override string ToString()  // переопределение метода ToString()
31:             {                                  // для вывода содержимого переменной экземпляра name
32:                 return "Dog name: " + name;
33:             }
```

```
34:
35:                     public override bool Equals(object obj) // переопределение метода Equals() для сравнения
36:                     {       // перем-ной name текущего экз-ра Dog с перем-ной объекта Dog, передан-го в аргум-те
37:                             Dog tempDog = (Dog) obj; // нисходящее приведение типа
38:
39:                             if(tempDog.Name == name)
40:                             {
41:                                     return true;
42:                             }
43:                             else
44:                             {
45:                                     return false;
46:                             }
47:                     }
48:             }
49:     }
50: /*************************************************************************/
51:     class ObjectTester
52:     {
53:             public static void Main()
54:             {                                                       // обе собаки имеют имя Fido
55:                     Animals.Dog myDog = new Animals.Dog("Fido");
56:                     Animals.Dog yourDog = new Animals.Dog("Fido");
57:
58:                     Console.WriteLine(myDog);
59:
60:                     if(myDog.Equals(yourDog))  // результат сравнения равен true
61:                             Console.WriteLine("myDog имеет то же имя, что и yourDog");
62:                     else
63:                             Console.WriteLine("myDog и yourDog имеют разные имена");
64:                             Console.ReadLine();
65:             }
66:     }
67: }
```

Результаты работы программы

```
Dog name: Fido
myDog имеет то же имя, что и yourDog
```

В строках **30—33** метод **ToString** переопределяется так, чтобы выводить содержимое переменной экземпляра **name**.

Метод **Equals** переопределяется в строках **35—47**, чтобы сравнивать переменную **name** текущего экземпляра **Dog** с переменной объекта **Dog**, переданного в аргументе.

Напомним, что переменная класса **TransportVehicle** в иерархии на на рис. 6.3, (см. Материалы к Практ. занят №6, стр. 26) могла быть использована для хранения объекта любого типа из данной иерархии, так как **TransportVehicle** находится на ее вершине. Аналогично, поскольку **System.Object** находится наверху в любой иерархии, переменная типа **System.Object** может содержать любой объект какого угодно типа: **Car, Shape, Account, Elevator, Bacterium** и т. д. В итоге, формальный параметр **obj** типа **object**, объявленный в строке **35**, может хранить ссылку на объект любого типа. Тем не менее, как любая переменная типа **Shape** ограничена доступом только к элементам класса **Shape**, даже если она и указывает на подкласс **Rectangle**, так и **obj** позволяет использовать только элементы **System.Object** из табл. 7.1, даже если он хранит объект

Dog. Для обращения к свойству **Name** объекта **Dog** через **obj** нужно выполнить нисходящее приведение типа, как показано в строке **37**, где **Dog** присваивается переменной **tempDog**. Следует обратить внимание, что для краткости здесь не производится проверка (с помощью операции **is** или **as**), является ли **obj** объектом типа **Dog**. Поэтому, если методу будет передан объект другого типа, попытка его приведения вызовет исключение. В строке **39** **tempDog** позволяет получить доступ к свойству **Name** объекта **Dog** и сравнить имя объекта **Dog** в аргументе с именем текущего.
Метод **WriteLine** в строке **58** автоматически вызывает **ToString** переменной **myDog** и выводит полученную строку на экран. Программисты, написавшие метод **WriteLine**, не знали, объект какого типа будет передаваться этому методу, но им было известно, что любой из объектов наследует **ToString**, и они без опасений позволили **WriteLine** вызвать метод **ToString** для объекта в аргументе.

В соответствии со строками **55** и **56** объекты **myDog** и **yourDog** имеют имя **Fido**, поэтому логическое выражение в строке **60** равно true.

//ПРИМЕЧАНИЕ
Любой класс, который переопределяет метод **Equals** (например. **Dog** в листинге 7.6), должен переопределять и метод **GetHashCode**, чтобы тот генерировал уникальное числовое значение, идентифицирующее объекты в хэш-таблице.

6. Сокрытие метода

Даже если элемент-функция (метод, свойство или индексатор) не объявлены как **virtual** в базовом классе, в классе производном все равно можно создать функцию с той же сигнатурой и типом возвращаемого значения. Однако для нее не будет активизирован механизм динамического связывания. Соответственно, если для переменной *базового* класса, которая указывает на объект *производного*, вызвать функцию, не объявленную как **virtual**, будет запущена версия, реализованная в *базовом*, а не в *производном* классе (как в случае виртуальных функций). Вместо переопределения функции базового класса, не объявленной как **virtual**, новая функция скрывает ее. Листинг 7.7 иллюстрирует разницу между: **1)** переопределением и 2) сокрытием метода базового класса.

6.1. Листинг 7.7 Исходный код MethodHidingTest.cs

```
01: using System;
02:
03: namespace ConsAppl_MethodHidingTest
04: {
05:
06:     class Car
07:     {
08:         public virtual void MoveForward() // виртуальный метод, он будет переопределен в произв. классе
09:         {
10:             Console.WriteLine("Обобщенный автомобиль переместился вперед на 1 км ");
11:         }
12:
13:         public void Reverse()
14:         {
15:             Console.WriteLine("Реверс обобщенного автомобиля на 50 м");    // НЕвиртуальный метод
16:         }
17:     }
18:/*****************************************************************************/
19:     class FamilyCar : Car
```

```
20:     {
21:             public override void MoveForward() // этот метод переопределяет м-д баз-го класса
22:             {
23:                     Console.WriteLine("Семейный автомобиль переместился вперед на 5 км ");
24:             }
25:
26:             public new void Reverse()   // этот метод скрыт
27:             {
28:                     Console.WriteLine("Реверс семейного автомобиля на 200 м ");
29:             }
30:     }
31: /*****************************************************************************/
32:     class Tester
33:     {
34:             public static void Main()
35:             {
36:                     Car myCar;
37:
38:                     myCar = new FamilyCar();
39:                     myCar.MoveForward();
40:                     myCar.Reverse();
41:             Console.ReadLine();
42:             }
43:     }
44: }
```

Результаты работы программы

```
Семейный автомобиль переместился вперед на 5 км
Реверс обобщенного автомобиля на 50 м
```

Чтобы показать разницу между виртуальным и невиртуальным методом, класс **Car** содержит как тот так и другой: виртуальный метод **MoveForward** и невиртуальный **Reverse**. Класс **FamilyCar** — производный от **Car**. Он переопределяет метод **MoveForward** в строках **21**—**24** и скрывает метод **Reverse** в строках **26**—**29**.

Если требуется скрыть функцию, следует использовать ключевое слово **new**, как в строке **26** (в таком контексте значение **new** отлично от используемого при создании новых объектов). Подробней о его назначении — в следующем разделе.

Далее в программе объявлена переменная базового класса **Car** (в строке **36**), а затем ей присвоена ссылка на объект производного класса **FamilyCar** (в строке **38**). Как показывает пример вывода, динамическое связывание используется для виртуального метода **MoveForward** (строка **39**), так как активизируется его реализация из производного класса **FamilyCar**. Динамическое связывание для невиртуального метода **Reverse** не применяется (в строке **40**) — исполняется реализация **Reverse** из класса **myCar**. Независимо оттого, на объект какого типа указывает **myCar** в вызове:

```
myCar.Reverse()
```

всегда будет задействована реализация **Reverse** из класса **Car**. Это приводит к серьезной проблеме: если **myCar** указывает на объект производного класса, очевидно, требуется вызвать его реализацию **Reverse**. Для ее решения метод **Reverse** (подобно **MoveForward**) должен быть объявлен как **virtual**.

Большинство функций, объявленных в базовом классе, подобны методам **MoveForward** и **Reverse** в том отношении, что их следует объявлять как **virtual**, чтобы избежать проблем такого

рода (как, например, когда метод **Reverse** не был объявлен как **virtual** в листинге 7.7). Но, если виртуальные функции используются чаще, чем невиртуальные, почему функция (без ключевого слова **virtual**) не становится виртуальной по умолчанию? Одна из главных причин этого связана с управлением версиями, представленным в следующем разделе.

7. Управление версиями с помощью ключевых слов new и override

Сегодня довольно много программного обеспечения создается с использованием библиотек классов, зачастую написанных другими программистами. Как и самостоятельные программы, библиотеки классов обновляются по мере устранения ошибок и добавления новых функций.

Проблемы с обновлением библиотек классов и их решения обычно называют управлением версиями. В этом разделе рассматривается один из недостатков управления версиями, связанный с наследованием и виртуозно устраненный в C#. Это помогает понять, почему C# располагает к использованию ключевого слова **new** (как в строке **26** листинга 7.7) при сокрытии элемента-функции в производном классе и ключевого слова **override** при ее перекрытии. Далее показано, почему *по умолчанию* применяются НЕвиртуальные методы (и почему при объявлении виртуальной функции необходимо *явно* указывать ключевое слово **virtual**).

Часто классы библиотеки используются как базовые — для создания производных. Этот прием демонстрировался, для вывода простого окна графического интерфейса например, расширением класса **System.Windows.FormsWinForms.Form** (см. Материалы к Практ. занят №6 стр. **24**).

Предположим, что нужно создать класс **SpaceShuttle**, расширяющий класс **Rocket** (последний содержится в библиотеке классов компании **BlipSoft**). Обычно классы и библиотеки классов содержатся внутри **dll**-сборок (см. Материалы к Практ. занят №4 стр. **21,сл.**). Их код обычно невозможно просмотреть, но в учебных целях код класса **Rocket** добавлен в листинг 7.8.

7.1. Листинг 7.8. Исходный код SpaceShuttle.cs

```
01: using System;
02:
03: namespace ConsAppl_SpaceShuttle
04: {
05:/*************************************************************************/
06:     class Rocket
07:     {
08:         // class Rocket  является частью библиотеки класса,
09:         // разработанного компанией BlipSoft
10:         private int age = 0;
11:
12:         public int Age
13:         {
14:             get
15:             {
16:                 return age;
17:             }
18:
19:             set
20:             {
21:                 age = value;
22:             }
23:         }
24:     }
```

```
25: /*****************************************************************/
26:     class SpaceShuttle : Rocket
27:     {
28:         // класс SpaceShuttle написан Вами
29:         private int distanceTravelled = 0;
30:
31:         public int DistanceTravelled
32:         {
33:             get
34:             {
35:                 return distanceTravelled;
36:             }
37:         }
38:
39:         public void MoveForward(int distanceAdded)
40:         {
41:             distanceTravelled += distanceAdded;
42:         }
43:     }
44: /*********************************************************************/
45:     class Tester
46:     {
47:         public static void Main()
48:         {
49:             SpaceShuttle columbia = new SpaceShuttle();
50:
51:             columbia.MoveForward(30);
52:             Console.WriteLine("Пройденное расстояние: {0}",
53:                 columbia.DistanceTravelled);
54:         Console.ReadLine();
55:         }
56:     }
57: }
```

Результаты работы программы

Distance travelled – Пройденное расстояние: **30**

Класс **Rocket** компании **BlipSoft** не включает переменную экземпляра для сохранения пройденного расстояния, поэтому она добавлена в класс **SpaceShuttle** (строка **29**) вместе с методом **MoveForward** (строки **39—42**), который прибавляет определенное значение к переменной экземпляра **distanceTravelled**. Такой подход работал, пока в новой версии библиотеки **BlipSoft** к базовому классу **Rocket** не был (по чистой случайности) добавлен виртуальный метод с точно таким же именем, типом возвращаемого значения и типами параметров, что и метод **MoveForward** в классе **SpaceShuttle**. Он выглядит так:

```
public virtual void MoveForward(int daysAdded)
{
age += daysAdded;
}
```

Программист, создающий приложение, никак не связан с **BlipSoft**, поэтому он может и не знать, что метод был добавлен. Если компилятор, как в некоторых объектно-ориентированных языках, автоматически "решит", что виртуальный метод базового класса должен быть переопределен методом с идентичной сигнатурой в производном, тогда **MoveForward** класса

SpaceShuttle переопределит **MoveForward** класса **Rocket**. Такой подход ошибочен, поскольку два этих метода выполняют совершенно разные действия: **MoveForward** из **Rocket** добавляет дни к переменной экземпляра **age**, а **MoveForward** из **SpaceShuttle** — расстояние к переменной экземпляра **distanceTravelled**. Компиляторы других языков не сообщают программисту о возможной ошибке, которая может долго оставаться необнаруженной (поскольку она не мешает программе исполняться).

//ПРИМЕЧАНИЕ

Если бы код был написан на C++, **MoveForward** из **SpaceShuttle** автоматически переопределял бы **MoveForward** из **Rocket** без каких-либо предупреждений. Благодаря динамическому связыванию любой вызов **MoveForward** через переменную типа **Rocket**, содержащую объект типа **SpaceShuttle**, приводил бы к вызову метода **MoveForward** из класса **SpaceShuttle**.

Если бы программа была написана на Java, и два метода **MoveForward** имели бы разные типы возвращаемых значений, программа не стала бы компилироваться, поскольку переопределяющий метод должен иметь тот же самый тип возвращаемого значения, что и переопределяемый.

Компилятор C# предотвращает подобные проблемы, выдавая предупреждение:

```
SpaceShuttle.cs(40,17): warning CS0114: 'SpaceShuttle.MoveForward(int)' hides inherited member 'Rocket.MoveForward(int)'. To make the current method override that implementation, add the override keyword. Otherwise add the new keyword.

SpaceShuttle.cs(40,17): предупреждение CS0114: 'SpaceShuttle.MoveForward(int)' скрывает унаследованный член 'Rocket.MoveForward(int)'. Для того, чтобы сделать метод текущим, аннулируйте эту реализацию, добавьте ключевое слово override. В противном случае добавить ключевое слово new.
```

Теперь можно объявить **MoveForward** в **SpaceShuttle** или переопределяющим, или новым методом (скрывающим **MoveForward** из **Rocket**). В данном случае метод следует объявить новым (**new**):

```
39:     public new void MoveForward(int distanceAdded)
40:     {
41:             distanceTravelled += distanceAdded;
42:     }
```

Следует отметить, что если игнорировать предупреждение компилятора и опустить ключевое слово **new** или **override**, программа все равно будет компилироваться, и **MoveForward** из **SpaceShuttle** станет по умолчанию новым методом. Тем не менее, ключевые слова **new** и **override** важны, в первую очередь для тех, кто читает текст программы, поэтому их рекомендуется использовать, устраняя таким образом и предупреждения компилятора.

Предположим, что метод **MoveForward** обновленного класса **Rocket** не подходит для переопределения, и разработчики сознательно не объявили его виртуальным. Тогда **MoveForward** из **SpaceShuttle** используется только для сокрытия **MoveForward** из **Rocket**, поскольку невиртуальный метод не может быть переопределен. Более того, здесь не будет возникать проблем из-за типов возвращаемого значения и других нюансов переопределен. Другими словами, два этих метода будут полностью разделены. Метод **MoveForward** из **Rocket** нельзя будет вызвать, значит, нельзя и спутать с методом из **SpaceShuttle**.

Как правило, больше беспорядка в производные классы вносят виртуальные, а не НЕвиртуальные методы. Поэтому, хотя большинство методов в программах и **virtual**, C# не делает такого объявления по умолчанию, а требует явной инструкции от программиста.

//ПРИМЕЧАНИЕ

Виртуальные функции исполняются немного медленнее, поскольку среде исполнения требуется определить, какую из реализаций вызвать. Исполнение невиртуальных методов определено на стадии компиляции. Это различие в производительности — еще один аргумент в пользу того, чтобы не объявлять функции как **virtual** по умолчанию.

8. Множественное наследование - сторонники и противники

Множественное наследование позволяет классу иметь более одного базового класса. Например, конкретный тип автомобиля (**Car**) может быть джипом (**Jeep**) и семейной машиной (**FamilyCar**) одновременно, как показано на рис. 7.5. Класс **FamilyCarJeep** — производный от **FamilyCar** и **Jeep**.

Несмотря на то что множественное наследование — признанная концепция, реализованная в языках **C++** и **Eiffel**, есть все же несколько проблем, осложняющих его применение.

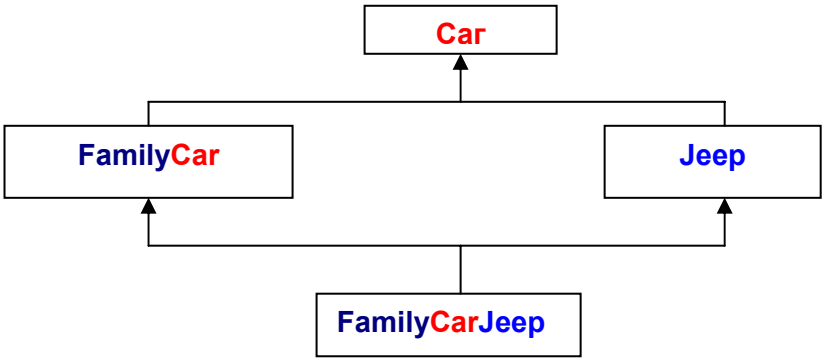


Рис. 7.5. Пример множественного наследования

//ПРИМЕЧАНИЕ

Концепцию множественного наследования активно обсуждают в среде программистов как ее сторонники, так и противники:

1. Одна из часто упоминаемых его противниками проблем предполагает ситуацию когда два базовых класса имеют метод с одной и той же сигнатурой, но разными реализациями. Какую из двух версий должен унаследовать производный класс? Предположим, например, что и **FamilyCar**, и **Jeep** реализуют один и тот же метод **ToString**. Версия этого метода в **FamilyCar** возвращает значения **brandName** и **odometer** (в виде строки), в то время как **Jeep** возвращает **numberOfSeats** и **suspensionSystemName**. Если **FamilyCarJeep** не переопределит **ToString**, какая из двух версий **ToString** должна наследоваться?

2. Сходная проблема возникает, когда два базовых класса содержат два члена данных с одинаковыми именами. Каждый из них может хранить свою информацию в каждом из двух базовых классов, и оба будут важны для производного. Нужно ли наследовать оба элемента, и если да, то как их различить?

В языках программирования, допускающих множественное наследование, предпринимаются попытки разрешить эти проблемы различными более или менее рациональными способами. Разработчики же C# решили устранить проблему, просто **запретив множественное наследование,**
и обеспечили вместо него другую конструкцию языка **— интерфейсы**.

9. Интерфейсы

Вернемся к классу **Shape** из программы черчения (листинги 7.2 и 7.3). Он содержал абстрактный метод **DrawYourself**, вынуждавший каждый класс, производный от **Shape**, реализовать метод **DrawYourself** с той же сигнатурой и типом возвращаемого значения. Такой подход гарантирует возможность вызывать разные реализации **DrawYourself** в **Circle**, **Triangle** и **Rectangle** посредством динамического связывания. Следует отметить, что класс **Shape** не содержит реализаций и поэтому не дает никаких преимуществ от повторного использования кода своим подклассам. Главная причина создания класса **Shape** — динамический вызов **DrawYourself**.

Конструкция языка, называемая **интерфейсом** представляет собой **особый абстрактный класс,** в котором содержатся только абстрактные функции (синтаксис интерфейса повторяет синтаксис абстрактного класса).

Класс может реализовать интерфейс. То есть класс должен реализовать абстрактные функции-члены интерфейса так же, как производный класс должен реализовать абстрактные функции базового. Таким образом, **интерфейс — альтернатива чисто абстрактному базовому классу для реализации функций, которые можно вызывать при помощи динамического связывания.** Например, в программе черчения можно заменить класс **Shape** на эквивалентный

интерфейс **Shape**, также содержащий абстрактный метод **DrawYourself**. Тогда классы **Circle**, **Triangle** и **Rectangle** реализовали бы интерфейс **Shape**. Функциональные возможности программы при этом остались бы неизменными.

Несмотря на то что интерфейсы и абстрактные классы тесно связаны синтаксически и семантически, у них есть одно важное различие: интерфейсы могут содержать только абстрактные методы, в то время как абстрактные классы помимо абстрактных функций могут содержать данные-члены и реализованные неабстрактные функции. По этим причинам реализация нескольких интерфейсов НЕ вызывает проблем, потенциально присущих множественному наследованию (с. 22). В результате класс может иметь только один базовый класс, но реализовывать сколько угодно интерфейсов.

Интерфейс может сам воплощать один или более интерфейсов и потому наследовать их абстрактные функции. Это позволяет интерфейсам формировать иерархию, подобную формируемой классами.

Обычно классы образуют строгие соподчиненные иерархии, представленные ранее, где каждая пара базовый класс — производный класс представляет отношение "является". Эти иерархии обладают рядом достоинств, но несколько ограничивают применение полиморфизма, и вот почему: полиморфизм требует, чтобы группа классов имела общего предка. В этом классе-предке определены заголовки функций (в форме абстрактных классов) и, таким образом, "протокол" для связи с индивидуальными реализациями в производных классах. Но как быть, если у классов, которые нужно сгруппировать в такую структуру, общего предка нет? Что если они разбросаны по программе и принадлежат разным иерархиям? Предположим, например, что создается игра "Космические пришельцы", содержащая две иерархии классов **A** и **B**.

Подобно **Circle**, **Triangle** и **Rectangle** в программе черчения, требуется изображать обьекты трех классов — **Planet**, **Galaxy** и **SpaceShip** — на экране. Каждый класс поддерживает свой собственный способ отображения, метод **DrawYourself**, и теперь их нужно сгруппировать под общим классом-предком, где этот метод будет объявлен как **abstract**. Это вызывает большие трудности, поскольку три класса принадлежат трем различным частям программы и общего предка у них нет. Можно попытаться решить эту проблему, добавив класс-предок на вершине обеих иерархий (тогда все классы будут содержать абстрактный метод **DrawYourself**). Однако при таком подходе, во-первых, нарушается иерархия, а, во-вторых, самый верхний класс (в силу необходимости применять полиморфизм) будет содержать все возможные, абсолютно несвязанные между собой абстрактные методы.

Более разумное решение — создание интерфейса, скажем, **IDrawable** (по соглашению имя интерфейса начинается с прописной буквы **I**), содержащего абстрактный метод **DrawYourself**, реализуемый в трех классах (рис. 7.6). Интерфейс **IDrawable** может выглядеть так (подробнее о синтаксисе — в следующем разделе):

```
public interface IDrawable
{
 void DrawYourself();
}
```

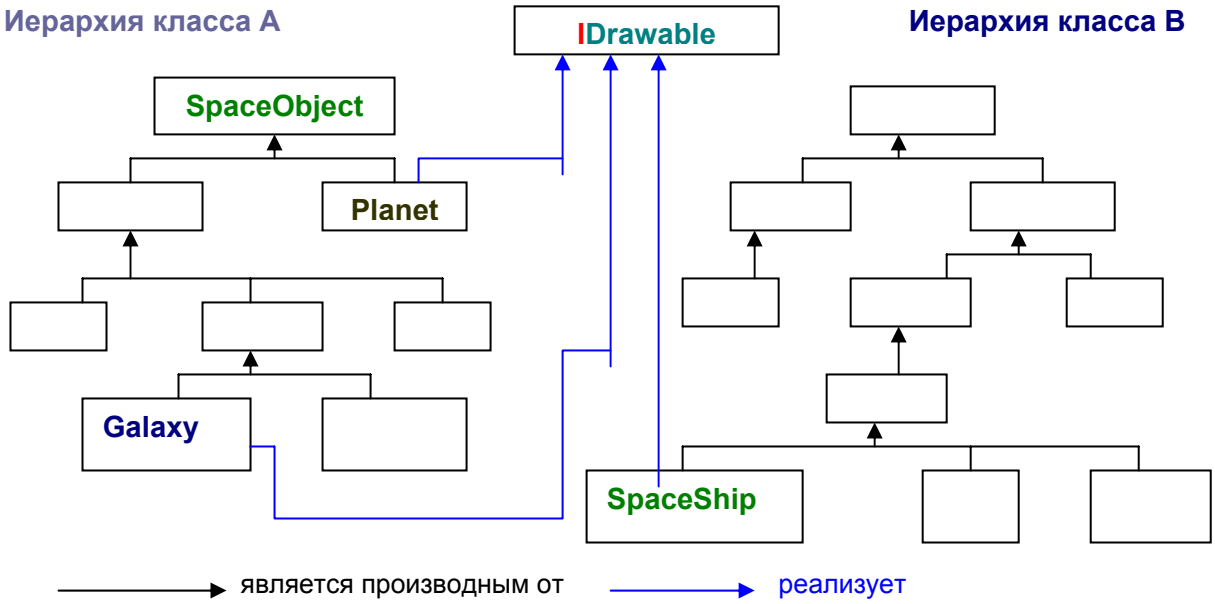


Рис. 7.6. IDrawable позволяет несвязанным между собой классам реализовать один и то же абстрактный метод

//ПРИМЕЧАНИЕ

Заменить интерфейс **IDrawable** классом нельзя, поскольку у **Planet**, **Galaxy** и **SpaceShip** тогда было бы по два базовых класса, что недопустимо.

IDrawable обеспечивает общий интерфейс, посредством которого в трех классах можно создать разные реализации. Аналогично тому как в строках **82—88** листинга 7.3 (с. 16) применялся массив **Shape**, можно разработать метод **DrawSpaceScene**, обрабатывающий массив типа **IDrawable**:

```
public void DrawSpaceScene(IDrawable [ ] drawing)
{
    for(int i = 0; i < drawing.Length; i++)
    {
        drawing [ i ].DrawYourself();
    }
}
```

Это тот же подход, что применялся ранее в программе черчения.

9.1. Определение интерфейса

Синтаксис определения интерфейса показан в синтаксическом блоке 7.2. Оно состоит из необязательного спецификатора доступности и следующих за ним ключевого слова **interface** и идентификатора. В необязательном элементе <Список_базовых_интерфейсов> указаны те из них, что реализованы в этом интерфейсе.

9.2. Синтаксический блок 7.2. Интерфейс

```
Определение_интерфейса::=
[<Спецификатор_доступности>] interface <Идентификатор_интерфейса>
[:<Список_базовых_интерфейсов>]
{
[<Абстрактные_методы>]          // 1)
[<Абстрактные_свойства>]        // 2)
[<Абстрактные_индексаторы>]     // 3)
[<События>]                     // 4)
```

}

где:

1) <Абстрактный_метод>::=
< Тип_возвращавмого_значения> <Идентификатор_метода> ([<Список_параметров>]);

2) <Абстрактное_ свойство:=
<Тип_свойства> <Идентификатор_свойства> { *[* **get;** *] [* **set;** *]*}

3) <Абстрактный_ индексатор >::=
<Тип_индексатора> **this** [<Список_параметров>] { *[* **get;** *] [* **set;** *]*}

4) <Событие>::=
event <Тип_события> <Идентификатор_события>*;*

Примечания:
По соглашению идентификатор интерфейса должен начинаться с прописной буквы **I** (от **Interface**), а поскольку интерфейс разрешает (**enable**) классу определенные действия, заставляя его реализовать свои абстрактные методы, идентификатор интерфейса часто заканчивается суффиксом -**able** (он обозначает способность к действию): **IComparable, ICloneable, IStorable** и т. д. Ни один из элементов интерфейса (абстрактные методы, свойства, индексаторы и события) не может иметь спецификаторов доступности. Все они неявно объявлены как **public**, поскольку должны быть доступны извне класса. Свойства и индексаторы, определенные в интерфейсе, могут иметь **get**- или **set**-аксессор. Аксессоры объявлены **abstract** неявно.

Обратимся к методу **CompareTo** из [1, глава 7, с. 243, сл.]. Он позволяет сравнить две строки лексикографически например:

```
myString.CompareTo(yourString)
```

Если **myString** больше **yourString**, то **CotpareTo** возвращает положительное значение, нуль — если строки идентичны, и отрицательное — если **myString** меньше **yourString**.

Метод **CotpareTo** в [1, глава 7, с. 243, сл.] применялся для сортировки строк в алфавитном порядке. Соответственно, каждый класс, содержащий метод, эквивалентный **CotpareTo**, также может сортировать объекты. Это позволяет создать универсальный метод, который может сортировать не только строки, но любой список объектов класса, содержащего реализацию **CotpareTo**. Для этого класс должен реализовать интерфейс, где объявлен абстрактный метод **CotpareTo**. Таким образом, для универсальной сортировки необходим интерфейс:

```
public interface IComparable
{
int CompareTo(IComparable comp);
}
```

Реализуя его, класс гарантирует, что он будет содержать версию метода **CotpareTo**, объявленную как **public**, с результатом типа **int** и одним формальным параметром типа **IComparable**. Более того, если метод **CotpareTo** реализован корректно и возвращает нуль, отрицательное или положительное значение по той же схеме, что и для класса **string**, значит, объекты можно сравнивать. В следующем разделе показано, как класс реализует такой интерфейс в общем случае, и как конкретно класс **TimeSpan** [1, глава 14, с. 564, сл.] может реализовать интерфейс **IComparable**.

9.3. Реализация интерфейса

Синтаксис для указания на то, что класс реализует интерфейс, представлен в синтаксическом блоке 7.3, похожем на блок наследования от базового класса (см. материалы к Практич. зан. №6, блок 6.1 на с. 6, 7). Двоеточие разделяет идентификатор класса и необязательные идентификатор базового класса и список реализованных интерфейсов.

9.4. Синтаксический блок 7.3. Определение класса

Определение_класса::=
<Спецификатор_доступности> **class** <Идентификатор_класса> [**:** [<Базовый_класс>]
[<Список_интерфейсов>]]
{
<Элементы класса>
}

где

<Список_интерфейсов>::=
<Идентификатор_интерфейса_**1**> [, <Идентификатор_интерфейса_**2**> ...]

П р и м е ч а н и я:

1. У класса может быть лишь один базовый класс.
2. Класс может реализовать неограниченное число интерфейсов. Интерфейсы разделяются запятыми.

Пример:

Класс **SportsCar** произведен от класса **Car** и реализует интерфейсы **ITunable** и **IInsurable**.

public class SportsCar : **Car**, **Itunable**, **IInsurable**
{
<*Элементы_класса*
}

Класс **SportsCar** должен реализовать все абстрактные элементы **Car**, **Itunable**, **IInsurable**, иначе **SportsCar** также будет абстрактным.

Позволив классу **TimeSpan** реализовать интерфейс **IComparable** и написав тело метода **CompareTo** в **TimeSpan**, можно сравнить любые два объекта **TimeSpan**, как в листинге 7.9.

9.5. Листинг 7.9. Исходный код ComparableTimeSpans.cs

```
01: using System;
02:
03: namespace ConsAppl_ComparableTimeSpans
04: {
05: /***************************************************************************/
06:     public interface IComparable   // форм-й парам-р comp имеет тип интерфейс IComparable
07:     {
08:             int CompareTo(IComparable comp);
09:     }
10: /***********************************************************************/
11:     public class TimeSpan : IComparable  // класс TimeSpan реализует интерфейс IComparable
12:     {
13:             private uint totalSeconds;
14:
15:             public TimeSpan()
16:             {
17:                     totalSeconds = 0;
18:             }
19:
20:             public TimeSpan(uint initialSeconds)
21:             {
22:                     totalSeconds = initialSeconds;
23:             }
```

```
24:
25:             public uint Seconds
26:             {
27:                     get
28:                     {
29:                             return totalSeconds;
30:                     }
31:
32:                     set
33:                     {
34:                             totalSeconds = value;
35:                     }
36:             }
37:
38:             public int CompareTo(IComparable comp)
39:             {
40:                     TimeSpan compareTime = (TimeSpan) comp;
41:
42:                     if(totalSeconds > compareTime.Seconds)
43:                             return 1;
44:                     else if(compareTime.Seconds == totalSeconds)
45:                             return 0;
46:                     else
47:                             return -1;
48:             }
49:     }
50: /*************************************************************/
51:     class Tester
52:     {
53:             public static void Main()
54:             {
55:                     TimeSpan myTime = new TimeSpan(3450);
56:                     TimeSpan worldRecord = new TimeSpan(1239);
57:
58:                     if(myTime.CompareTo(worldRecord) < 0)
59:                             Console.WriteLine("Мое время - ниже мирового рекорда");
60:                     else if(myTime.CompareTo(worldRecord) == 0)
61:                             Console.WriteLine("Мое время такое же как и мировой рекорд ");
62:                     else
63:                             Console.WriteLine("Я затратил  больше времени, чем рекордсмен ");
64:                     Console.ReadLine();
65:             }
66:     }
67: }
```

Результаты работы программы

```
Я затратил  больше времени, чем рекордсмен
```

Интерфейс **IComparable** определен в строках **06**…**09**. Его можно использовать для указания типа формального параметра функции **CompareTo** в строке **08**. Таким образом, любой из классов, реализующих **IComparable**, можно передать этому методу как аргумент. Если задать тип формального параметра как **TimeSpan**, интерфейс был бы полезен только для класса **TimeSpan**, что не позволило бы создать универсальный метод сортировки.

Двоеточие в строке **11** и слово **IComparable** за ним указывают, что **TimeSpan** реализует интерфейс **IComparable**.

В строках **38—48** — реализация метода **CotpareTo**. Сигнатура и тип результата в строке **38** идентичны указанным в объявлении интерфейса **IComparable**. Спецификатор доступности — **public**, поскольку в своем определении **CotpareTo** неявно объявлен как **public**. Задача метода **CotpareTo** в **TimeSpace** — сравнение переменной экземпляра **totalSeconds** из объекта **TimeSpan**, на который указывает параметр **comp**, с переменной экземпляра **totalSeconds** текущего объекта.

//ПРИМЕЧАНИЕ

Метод **CotpareTo**, реализованный в классе **Circle**, может сравнивать радиусы. Класс **Account** может сравнивать балансы, а **TimeSpan** — totalSeconds.

Значение **totalSeconds** текущего объекта доступно непосредственно, а **totalSeconds** объекта **comp** можно получить благодаря свойству **Seconds**. Хотя **comp** и указывает на объект типа **TimeSpan**, свойство **Seconds** недоступно через параметр **comp**. Через **comp** можно вызвать только **CotpareTo**, поскольку это единственный метод, указанный в интерфейсе. Для доступа к свойству **Seconds** необходимо привести **comp** к типу **TimeSpan** (строка **40**). После этого можно обратиться к **Seconds** с помощью **compareTime** в строках **42** и **44**.

Так же как в программе черчения метод **DrawYourself** вызывался прямо из объекта одного из подклассов **Shape** без динамического связывания:

```
Circle myCircle;
myCircle.DrawYourself();        //НЕ требует динамического связывания
```

метод **CotpareTo** можно вызвать прямо из **TimeSpan** (строки **58** и **60**). И как **myCirde.DrawYourself** не показывает всех возможностей полиморфизма, так и вызов **CotpareTo** в строках **58** и **60**

```
myTirne.CompareTo(worldRecord)        // НЕ требует динамического связывания
```

годится только для того, чтобы показать, как работает метод **CotpareTo**. Тот же результат можно получить и без интерфейса **IComparable**.

Хотя переменная **worldRecord** объявлена **TimeSpan** в строке **56**, ее можно (строки **58** и **60**) использовать как аргумент метода **CotpareTo** (его аргументы принадлежат типу **IComparable**), поскольку **TimeSpan** реализует интерфейс **IComparable**.

Листинг 7.9 иллюстрирует синтаксис, но не настоящие возможности интерфейсов. Им посвящен следующий раздел.

9.6. Универсализация программирования при помощи интерфейсов

Предположим, что нужно отсортировать список чисел по возрастанию. Разработка системы сортировки займет время, но создать ее можно. Вскоре понадобится отсортировать список студентов по их оценкам. Окажется, что можно использовать ту же самую систему сортировки, только вместо пар чисел нужно сравнивать пары студентов. На самом деле этим же способом можно сортировать любые списки объектов, сравнивая их попарно. При этом природа сортируемых объектов значения не имеет.

Тот же принцип применяется и в программировании для поддержки повторного использования кода. Если принципиально реализация алгоритма сортировки не зависит от сортируемых объектов, можно создать обобщенную реализацию этой процедуры — ей можно будет воспользоваться для сортировки любых объектов (сравнение которых возможно).

В этом разделе показано, как, применяя интерфейсы, написать обобщенную версию метода **BubbleSortAscending** он приведен в листинге 7.10).

9.7. Листинг 7.10. Исходный код GenericBubbleSort.cs

```
001: using System;
002:
003: namespace ConApp_GenericBubbleSort
004: {
005:/*************************************************************/
006:   public interface IComparable
007:   {                                  // форм-й парам-р comp имеет тип интерфейс IComparable
008:      int CompareTo(IComparable comp);
009:   }
010: /*****************************************************************/
011:   public class TimeSpan : IComparable  // "Сортируемый" класс TimeSpan
012:   {
013:      private uint totalSeconds;
014:
015:      public TimeSpan()
016:      {
017:         totalSeconds = 0;
018:      }
019:
020:      public TimeSpan(uint initialSeconds)
021:      {
022:         totalSeconds = initialSeconds;
023:      }
024:
025:      public uint Seconds  // свойство
026:      {
027:         get
028:         {
029:            return totalSeconds;
030:         }
031:
032:         set
033:         {
034:            totalSeconds = value;
035:         }
036:      }
037:
038:      public virtual int CompareTo(IComparable comp)
039:      {
040:         TimeSpan compareTime = (TimeSpan) comp;
041:
042:         if (totalSeconds > compareTime.Seconds)
043:            return 1;
044:         else if (compareTime.Seconds == totalSeconds)
045:            return 0;
046:         else
047:            return -1;
048:      }
```

```
049:    }
050: /*************************************************************/
051:    class Sorter
052:    {
053:                          // Обобщенный метод сортировки элем-ов массива сравниваемых по возрастанию (строки 054 - 070)
054:       public static void BubbleSortAscending(IComparable[ ] bubbles)
055:       { // формальный параметр bubbles объявлен массивом (сравниваемых объектов) типа IComparable
056:          bool swapped = true;
057:
058:          for (int i = 0; swapped; i++)
059:          {
060:             swapped = false;
061:             for (int j = 0; j < (bubbles.Length - ( i + 1 )); j++)
062:             {
063:                if (bubbles[ j ].CompareTo(bubbles[ j + 1 ]) > 0) // Работает механизм динам-го связывания
064:                {
065:                   Swap( j, j + 1, bubbles) ; // метод Swap() – строки 073 - 080
066:                   swapped = true;
067:                }
068:             }
069:          }
070:       }
071:
072:                                // Перестановка двух элементов массива
073:       public static void Swap(int first, int second, IComparable[ ] arr)
074:       {
075:          IComparable temp;
076:
077:          temp           = arr[ first ];
078:          arr[ first ]   = arr[ second ];
079:          arr[ second ] = temp;
080:       }
081:    }
082: /*************************************************************/
083:    class Tester
084:    {
085:       public static void Main()
086:       {
087:          TimeSpan[ ] raceTimes = new TimeSpan[4];
088:
089:          raceTimes[0] = new TimeSpan(153);
090:          raceTimes[1] = new TimeSpan(165);
091:          raceTimes[2] = new TimeSpan(108);
092:          raceTimes[3] = new TimeSpan(142);
093:
094:          Sorter.BubbleSortAscending(raceTimes);
095:
096:          Console.WriteLine("Список временных промежутков упорядоченных по возрастанию:");
097:          foreach (TimeSpan time in raceTimes)
```

```
098:         {
099:             Console.WriteLine(time.Seconds);
100:         }
101:         Console.ReadLine();
102:     }
103: }
104: }
```

Результаты работы программы

```
Список временных промежутков упорядоченных по возрастанию:
108
142
153
165
```

Метод **BubbleSortAscending** из листинга 7.10 является обобщенным. В строке **063** проверяется условие: "Элемент **j** больше элемента **j+1**?". То есть, если тип элемента массива, переданного методу сортировки, содержит метод с именем **CotpareTo**, который возвращает положительное значение, когда элемент с индексом **j** больше элемента с индексом **j+1**, то объекты в массиве можно сравнить так:

```
063:        if (bubbles[ j ].CompareTo(bubbles[ j + 1 ]) > 0)
```

Как же гарантировать наличие метода **CotpareTo**? Любой класс, реализующий **интерфейс IComparable**, определенный в предыдущем разделе (№№№), содержит метод **CotpareTo**. Таким образом, элементы массива, реализующего **IComparable**, можно сортировать данным методом. Объявляя формальный параметр обобщенного метода сортировки массивом типа **IComparable**, как это сделано в заголовке из строки **54** листинга 7.10, можно указать, что этому методу будут передаваться массивы только сравнимых объектов.

Классу **TimeSpan** достаточно реализовать интерфейс **IComparable**, как указано в строке **011**, и **CompareTo** (строки **038**—**048**), чтобы стать "сортируемым". Разумеется, это относится не только к классу **TimeSpan**. **Account, RacingCar, Circle, Bacterium** и любой подходящий класс могут обеспечивать такую сортировку, если они реализуют **интерфейс IComparable** (достаточно лишь добавить несколько строк кода для реализации метода **CompareTo**). В этом случае каждый класс сможет повторно использовать однажды написанный метод сортировки.

Здесь вновь неявным образом работает механизм динамического связывания (строка **63** листинга 7.10). Хотя массив **bubbles** и содержит элементы **IComparable**, вызов в строке **063** автоматически, через динамическое связывание, вызывает реализацию **CompareTo**, соответствующую типу объектов, хранящихся в элементах массива в момент вызова.

Метод **Main** показывает возможность сортировки массива из четырех объектов **TimeSpan**. Следует обратить внимание: массив **raceTime** имеет тип **TimeSpan**, но принимается механизмом сортировки, поскольку **TimeSpan** реализует интерфейс **IComparable**.

9.8. Экземпляр интерфейса можно породить только опосредованно

Как и экземпляр абстрактного класса, экземпляр интерфейса породить невозможно:

```
IComparable icComp = new IComparable ();        // НЕверно
```

Можно порождать экземпляры только тех классов, которые обеспечивают реализацию всех абстрактных методов, указанных в классах-предках, и реализацию интерфейсов.

9.9. Построение иерархий интерфейсов

Можно расширить существующий интерфейс **A**, разрешив интерфейсу **B** реализовать интерфейс **A**. Тогда **B** содержит элементы **A**, и элементы, определенные в нем самом. В этом случае класс может реализовать интерфейс **A** или **B** в соответствии со своими потребностями. Если класс реализует интерфейс **A**, он должен реализовать его абстрактные функции, если **B** — реализовать не только абстрактные функции, определенные в **B**, но и унаследованные им от **A**.

Например, можно расширить интерфейс **IComparable** из предыдущего раздела новым интерфейсом **IComparableAdvanced**, как показано ниже:

```
interface IComparableAdvanced : IComparable

bool GreaterThan(IComparableAdvanced comp);

bool LessThan(IComparableAdvanced comp);
```

Любой класс, реализующий **IComparableAdvanced**, теперь должен реализовать не только **GreaterThan** и **LessThan**, но и **CompareTo**. Несколько интерфейсов могут расширять друг друга, формируя иерархию интерфейсов.

9.10. Преобразования интерфейсов

Если объект класса **C** реализует интерфейс **I**, экземпляр класса **C** можно присвоить переменной интерфейса **I** без явного приведения типа, например:

```
IComparable icTime = new TimeSpan(392);
```

поскольку **TimeSpan** реализует интерфейс **IComparable**.

Если же нужно двигаться в другую сторону (как в строке **40** листинга 7.10) и преобразовать **icTime** в переменную типа **TimeSpan**, понадобится аналог нисходящего приведения типов. (Нисходящее приведение позволит получить доступ ко всем элементам объекта **TimeSpan**, а не только к указанным в интерфейсе **IComparable**.)

```
TimeSpan myTime = (TimeSpan) icTime;
```

Если **icTime** содержит не объект **TimeSpan** (а объект другого типа, также реализующего интерфейс **IComparable**), система генерирует исключение. Если заранее неизвестно, содержит ли **icTime** объект **TimeSpan**, можно воспользоваться операциями **is** и **as**.

Здесь применяется **is**:

```
TimeSpan myTime;
if (icTime is TimeSpan)
    myTime = (TimeSpan) icTime;
else
    Console.WriteLine("Нельзя назначать icTime на myTime");
```

а здесь — **as**:

```
TimeSpan myTime;
myTime = icTime as TimeSpan;
if (myTime == null)
    Console.WriteLine("Нельзя назначать icTime на myTime ");
```

В таком контексте операция **as** эффективнее, чем **is**.

//ПРИМЕЧАНИЕ

Иногда требуется просто проверить тип интерфейса, не выполняя приведения. Для этой цели подходит операция **is**, поскольку она не осуществляет приведение типа автоматически.

9.11. Переопределение виртуальных реализаций интерфейса

Любая функция, которую класс реализует из интерфейса, может быть объявлена как **virtual**. Например, метод **CompareTo** из **TimeSpan** объявлен как **virtual** в строке **38** листинга 7.10. Класс, производный от **TimeSpan**, может: **1)** переопределить этот виртуальный метод (**override**) обычным образом или **2)** обеспечить новую реализацию (**new**).

Например, можно спроектировать класс **TimeSpanAdvanced**, переопределяющий метод **СотрareTo** класса **TimeSpan**, чтобы обеспечить более подробный ответ (см. листинг 7.11). Новый метод **СотрareTo** возвращает 2 (строка **10**), если объект **A** типа **TimeSpan** больше объекта **B** на 50 секунд или более. И наоборот, если **A** меньше **B** на 50 секунд или более, возвращается значение -2.

9.12. Листинг 7.11. Исходный код TimeSpanAdvance.cs

```
01: public class TimeSpanAdvanced : TimeSpan
02: {                                          // Код из листинга 7.11 не компилируется отдельно
03:    public override int CompareTo(IComparable comp)
04:    {
05:       TimeSpan compareTime = (TimeSpan) comp;
06:
07:       if (base.Seconds > compareTime.Seconds)
08:       {
09:          if (base.Seconds > (compareTime.Seconds + 50))
10:             return 2;
11:          else
12:             return 1;
13:       }
14:       else if (base.Seconds < compareTime.Seconds)
15:       {
16:          if (base.Seconds < (compareTime.Seconds - 50))
17:             return -2;
18:          else
19:             return -1;
20:       }
21:       else
22:          return 0;
23:    }
24: }
```

//ПРИМЕЧАНИЕ

Код из листинга 7.11 не компилируется отдельно. Класс **TimeSpanAdvanced** можно вставить перед классом **TimeSpan** в листинге 7.10 и для проверки сделать несколько вызовов его метода **СотрareTo**.

9.13. Явная реализация функций интерфейса

При реализации метода **СотрareTo** в классе **TimeSpan** неявно подразумевалось, что это метод, указанный в интерфейсе **IComparable**. Обычно в класс достаточно включить заголовок функции с той же сигнатурой, типом возвращаемого значения и спецификатором доступности (**public**), что и у абстрактной функции интерфейса, которая реализуется. Указывать интерфейс явно незачем.

Однако если класс реализует два интерфейса, каждый из которых содержит абстрактную функцию с одним и тем же именем, компилятор не сможет сам определить по заголовку метода, какой из двух интерфейсов требуется реализовать.

Например, если бы космическая игра содержала не только интерфейс **IDrawable** с методом **DrawYourself**, предназначенный для вывода объектов игры (**Planet, Spaceship** и т. д.) на экран

```
interface IDrawable
{
void DrawYourself();
}
```

но и интерфейс **IPrintable** с абстрактным методом **DrawYourself** для печати объектов на принтере

```
interface IPrintable
{
void DrawYourself();
}
```

компилятор не смог бы определить, какой из интерфейсов **DrawYourself** реализован.

```
public class Spaceship : IDrawable, IPrintable
{
    ...
    public void DrawYourself()
    {
        ...
    }

    ...
    public void DrawYourself()
    {
        ...
    }
    ...
}
```

Этот метод реализует **DrawYourself** из интерфейса **IDrawable** или из **IPrintable**?

Этот метод реализует **DrawYourself** из итефейса **IDrawable** или из **IPrintable**?

Для решения этой проблемы необходимо явно указать одну или обе реализации, разместив имя интерфейса перед именем функции.

```
public class Spaceship : IDrawable, IPrintable
{
    …
    void IDrawable.DrawYourself()
    {
        …
    }

    …
    public void DrawYourself()
    {
        …
    }
    …
}
```

Этот метод явно реализует **DrawYourself** из интерфейса **IDrawable**

Этот метод явно реализует **DrawYourself** из интерфейса **IPrintable**?

В данном случае явное указание применено только в реализации **IDrawable**. Теперь компилятор сможет определить, что другой метод **DrawYourself** реализует метод интерфейса **IPrintable**. Как вариант можно явно указать оба метода.

Явные реализации неявно являются открытыми, поэтому с ними нельзя использовать спецификатор доступности **public**. Более того, явная реализация не может был объявлена как **abstract, virtual** или **new**.

Доступ к явно реализованной функции через ее объект невозможен:

```
Spaceship mySpaceShip = new SpaceShip();
mySpaceShip.DrawYourself()      // Вызывает DrawYourself, реализованный неявно, а не
                                // DrawYourself, реализованный явно
```

Вместо этого можно вызвать реализацию **DrawYourself** для **IDrawable** через переменную типа **IDrawable** (используя динамическое связывание):

```
IDrawable idShip = mySpaceShip;

IdShip.DrawYourself()
```

Явная реализация может пригодиться не только для того, чтобы избежать совпадения методов из разных интерфейсов. Иногда требуется реализовать интерфейс только для того, чтобы некоторые или все его методы вызывались путем динамического связывания, но не из самого объекта. Например, если нужно, чтобы реализация **TimeSpace** метода **CompareTo** вызывалась только посредством динамического связывания и не была в числе функций объекта, доступных напрямую, ее можно определить так:

```
public class TimeSpace
{
    …
    int IComparable.CompareTo(IComparable comp)
    {
      …
    }
}
```

Третья строка в приведенном ниже фрагменте теперь некорректна:

```
TimeSpan myTime = new TimeSpan();
TimeSpan yourTirne = new TimeSpan();
 myTime.CompareTo(yourTime) // НЕверно
```

Следующий код корректен:

```
IComparable icTime = myTime;
icTirne.CompareTo(yourTime)
```

//ПРИМЕЧАНИЕ

Как невозможно объявить явную реализацию интерфейса виртуальной, так и производный класс не может переопределить эту функцию, а должен повторно реализовать ее. Например, класс **TimeSpanAdvanced** из листинга 7.11 не может использовать ключевое слово **override** в строке **3**, если **CompareTo** был явно реализован в **TimeSpan**.

Резюме

Рассмотрены понятия **абстрактных функций**, **полиморфизма** и **интерфейсов**.

Абстрактный метод состоит из заголовка метода, но не имеет тела, а, следовательно, и реализации. Свойства и индексаторы также можно объявить **abstract**.

Класс, содержащий одну (или более) абстрактную функцию, и сам должен быть объявлен абстрактным. Абстрактный класс может содержать неабстрактные функции-члены.

Экземпляр абстрактного класса породить невозможно.

Класс, производный от абстрактного, должен реализовать все абстрактные методы базового, иначе он сам становится абстрактным. Абстрактные функции неявно объявлены виртуальными, поэтому реализация производится также, как переопределение виртуальных методов.

Полиморфизм — это существование во множестве форм. В программировании этот термин характеризует способность переменной указывать на объекты разных типов и вызывать различные реализации методов одним и тем же вызовом. Механизм, воплощающий эту идею, называется динамическим связыванием.

Полиморфизм — признанная концепция, позволяющая разработчикам программировать на уровне **интерфейсов** (*наборов вызовов методов*), тогда как вызов конкретных реализаций определяется во время исполнения.

Полиморфная переменная, через которую осуществляются полиморфные вызовы, может указывать на объекты разных классов-потомков. Когда объект класса-потомка присваивается переменной класса-предка, он теряет уникальность. Это называется восходящим приведением типов. В этом случае можно вызывать только элементы класса, определенные в классе-предке, даже если переменная ссылается на объект, содержащий и дополнительные элементы класса. Для идентификации объекта предназначена операция **is**. Чтобы восстановить утраченную информацию, можно использовать нисходящее приведение при помощи операции приведения или операции **as**.

Все классы являются производными от класса **System.Object** из библиотеки классов **.NET Framework** . Поэтому все классы наследуют шесть нестатических методов этого класса.

Когда в производном классе описана функция с той же самой сигнатурой и типом возвращаемого значения, что и невиртуальная функция базового класса, происходит сокрытие метода базового класса. Динамическое связывание не применяется к скрытым функциям. Даже если функция в базовом классе — виртуальная, ее можно скрыть, применив ключевое слово **new** к функции в производном классе.

Обновление библиотек классов, от которых зависит программа, может вызвать массу проблем и, следовательно, потребовать возможных решений. Все это называют управлением версиями. Правильное применение ключевых слов **virtual**, **new** и **override** позволяет предотвратить несколько проблем, часто возникающих при обновлении библиотек.

Несмотря на то что в большинстве случаев функции виртуальны, по умолчанию функция объявляется невиртуальной. Причина в том, что виртуальные функции могут вызвать гораздо больше проблем в производных классах, чем невиртуальные, а, кроме того, исполняются они медленнее.

При множественном наследовании класс может иметь несколько базовых классов. C# не поддерживает множественного наследования. Вместо этого предусмотрены интерфейсы.

Интерфейс содержит только абстрактные функции-члены и события. Таким образом, удается избежать конфликта имен переменных экземпляров и реализаций, свойственного множественному наследованию. В итоге, класс может иметь только один базовый класс, но реализовать несколько интерфейсов.

Интерфейсы следует использовать тогда, когда нужно, чтобы несколько классов содержали общие заголовки функций, отсутствующие в общем классе-предке. Это позволит применять полиморфизм к группе классов вне зависимости от того, в каком месте иерархии они расположены.

Если несколько реализаций идентичны, за исключением типа данных, к которым они применяются, можно создать обобщенную реализацию. Обобщенные реализации возможны благодаря полиморфизму.

Контрольные вопросы

1. Классы **Dog, Cat** и **Duck** являются производными от класса **Animal**. Предположим, что любой объект типа **Animal** может издавать звук. Где нужно разместить метод **Sound**? Следует ли реализовать его или объявить как **abstract**? Почему?

2. Если метод **Sound** класса **Animal** объявлен как **abstract**, можно ли создавать объекты этого класса? Почему?

3. Требуется реализовать метод **Sound** в каждом из трех подклассов класса **Animal** и вызывать их, используя полиморфизм. Какие ключевые слова нужно использовать в объявлении метода **Sound** в **Animal** и в трех подклассах, чтобы воплотить эту идею? Напишите заголовки методов для класса **Animal** и трех подклассов.

4. Что неправильно в следующем фрагменте программы?

```
public abstract void Sound()
{
Console.WriteLine("Quaaakkk quaaakkk");
}
```

5. Если нужно вызвать метод **Sound** трех подклассов полиморфно, следует ли использовать переменную типа **Animal** или переменные трех типов — **Cat, Dog** и **Duck**?

6. Предположим, что метод **Sound** был объявлен в классе **Animal** и реализован в трех подклассах так, что каждая из трех реализаций может быть вызвана посредством динамического связывания. Какая из трех реализаций вызывается во второй строке?

```
Animal myAnimal = new Dog();
myAnimal.Sound;
```

7. Имеется еще один класс — **Lion**, также содержащий метод **Sound**, который нужно вызывать полиморфно через переменную типа **Animal**. Что сделать, чтобы это стало возможным?

8. Как узнать, указывает ли переменная **myAnimal** (объявленная как **Animal myAnimal**;) на объект типа **Dog**?

9. Вам нужно привести **myAnimal** к объекту типа **Cat**, но только в случае, если **myAnimal** содержит **Cat**. Предложите два разных способа.

10. Рассмотрите класс **Cat** из предыдущего вопроса. Вы определили в нем метод **Sound**. Другой программист использует класс **Cat** и пишет такой вызов в своем фрагменте программы (строки **2** и **3**):

```
Cat myCat = new Cat();
myCat.Jump();
Console.WriteLine(rnyCat.ToString());
```

Корректны ли такие вызовы? Если какой-то из них допустим, какой результат будет получен? Объясните ход событий.

11. Если большинство методов виртуально, почему они не объявляются как **virtual** по умолчанию?

12. Почему C# не поддерживает множественное наследование?

13. В чем некорректно следующее объявление интерфейса?

```
interface IRecoverable
{
public void Recover()
{
Console.WriteLine("I am recovering");
```

14. Программист посоветовал вам улучшить вашу программу, заменив класс **Animal** из вопросов **1—3** на интерфейс **IArticulateable**. Правильно ли это? Почему?

Упражнения по программированию

1. Напишите простой класс **Account**, содержащий переменную экземпляра **balance** и свойство **Balance** для доступа к ней. Обеспечьте сортировку для класса **Account** при помощи метода **BubbleSortAscending** (из класса **Sorter** листинга 7.10) посредством реализации интерфейса **IComparable**. Проверьте класс **Account**, создав массив **Accounts** с различными значениями **balance** и передав их в качестве аргументов методу **BubbleSortAscending**.

2. Напишите три класса: **Secretary, Director** и **Programmer**. Каждый из них должен содержать метод **CalculateSalary**. Для простоты пусть каждый из них выводит строку "Теперь расчет жалованья для..." и имя класса.

Предположим, что в вашей программе много объектов этих трех типов и их необходимо хранить в массиве. Как хранить их в одном массиве и вызывать **CalculateSalary** для каждого объекта при прохождении по массиву вызовом одного и того же метода для всех объектов? Напишите программу.

3. Доработайте программу из упражнения **2**, выстроив иерархию так: базовый класс **Building** содержащий две переменные экземпляра — **age** типа **int** и **currentValue** типа **decimal**, и два класса — **House** и **OfficeBuilding**, производных от **Building**. Пусть **House** содержит переменную экземпляра **numberOffBedrooms** типа **ushort**, а **OfficeBuilding** — переменную экземпляра **floorSpace** типа **uint**.

Необходимо записывать объекты типа **Secretary** и **House** в файл и считывать их оттуда. Оба класса должны иметь соответствующие методы **Read** и **Write** (для простоты сделайте так, чтобы **Read** выводил на экран "Теперь чтение Дома" и "Теперь чтение Секретаря", а **Write** — "Теперь чтение Дома" и "Теперь чтение Секретаря"). Требуется создать метод, который принимает в аргументе любой объект, содержащий методы **Read** и **Write**, и полиморфно вызывает их в зависимости от типа объекта. Напишите программу, реализующую изложенный подход.

Основные этапы компьютерного моделирования реальных и концептуальных систем

1. Сформулировать целевую функцию моделирования (например, собрать статистическую информацию о качестве функционирования системы >> см. материалы к Практич. Зан. №**3 с. 11**).
2. Выполнить анализ реальной системы: учесть её важные действия и атрибуты и отбросить вторичные (реализация концепции ООА – абстракция >> см. материалы к Практич. Зан. №**3, с. 4**).
3. Выделить классы объектов, входящих в реальную систему >> см. материалы к Практич. Зан. №**1, с. 9**.
4. Установить связь между классами; выстроить иерархию классов и проанализировать целесообразность повторного использования их кода (реализация концепции ООП - наследование >> см. материалы к Практич. Зан. №**6 с. 4**)
5. Определить объекты классов – экземпляры классов >> см. материалы к Практич. Зан. №**1, с. 2**.
6. Сформулировать переменные экземпляров, которые отражают атрибуты объектов.
7. Установить какие действия должны выполнять объекты, то есть определить методы объектов и их функциональность.
8. Обдумать степень открытости методов и переменных экземпляров (реализация концепции ООП - инкапсуляция >> см. материалы к Практич. Зан. №**3 с. 4**).
9. Выполнить внутреннее проектирование методов; обдумать целесообразность обращения к различным реализациям методов одним и тем же вызовом (реализация концепции ООП – полиморфизм > механизм динамического связывания >> см. материалы к Практич. Зан. №**7 с. 8**)
10. Приступить к разработке исходного кода. На заключительном этапе реализовать парадигму компонентно-ориентированного программирования >> см. материалы к Практич. Зан. №**4 с. 21.**
11. etc.

Примечание.

Указанная последовательность является примерной. По мере осуществления последующих этапов возникает необходимость уточнять реализацию предыдущих.

Примеры.

1. Моделирование системы лифтов здания

См. материалы:

- **Пр_зан_№ 1**, с. 7, 8 **.**
- **Пр_зан_№ 3**, с. 4, 5, 6, 7, 8, 11 – 27, листинг 3.1**.**
- **Пр_зан_№ 4**, с. 4, 5, 6, 21, 22, 31-39, листинги 4.1, 4.2, 4.7-4.10**.**
- **Пр_зан_№ 5**, с. 4-7, 15, 16, 18-21, листинги 5.1, 5.5, 5.6**.**

2. Моделирование работы банка

См. материалы:

- **Пр_зан_№ 5**, с. 7-15, 25-32, 37**,** листинги 5.2, 5.7.

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)

……..