Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

C#

Объектно-ориентированный язык программирования

Пособие к практическим занятиям - №5

Проф. Забудский Е.И.

Москва 2005

Тема 6. Объектно-ориентированный подход

к разработке программного обеспечения.

Массивы. Анатомия класса

Компонентно-ориентированное программирование

Три практических занятия
(6 часов)

Рассмотрены алгоритмы и две *объектно-ориентированные* cs-программы (листинги 5.2 и 5.7), **моделирующие работу банка:**

- листинг 5.2. Программа **моделирует работу банка**, содержащего несколько счетов;
- листинг 5.7. В исходном коде **реализовано открытие и закрытие банковских счетов** в течение времени исполнения программы.

Рассмотрены алгоритмы и три cs-программы (листинги 5.1, 5.5 и 5.6), которые могут быть частью *объектно-ориентированной* программы, **моделирующей работу лифта** (программа, моделирующая работу лифта изучена на Практическом занятии - **№3**, стр. 11 - 29):

- листинг 5.1. В исходном коде для эффективного **управления коллекцией лифтов** используется **массив объектов;**
- листинг 5.5. В исходном коде реализовано **отслеживание вызовов лифта** в течение каждого часа за семидневный срок;
- листинг 5.6. В исходном коде **реализован сбор количества вызовов лифта каждый день за различное число часов**.

За компонентно-ориентированным программированием – будущее (см. задание на дом, **стр. 34**)

// **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. Массивы и классы

Базовый тип массива может быть не только простым типом (см. Материалы к Практ. занятию №4, стр. 45), а также может быть любым, включая класс. В следующем разделе приведены примеры, как работать с сохраненными в элементах массива ссылками на объекты (с использованием важного свойства массивов в C# играть роль переменных экземпляра в классах).

1.1. Элементы массива как ссылки на объекты

В программе, приведенной на листинге 5.1, для эффективного управления коллекцией лифтов используется массив объектов (строка **37**). Программа демонстрирует синтаксис и приемы работы с массивом, содержащим ссылки на объекты. Напомним пример моделирования лифта – (см. Материалы к Практ. занятию №3, стр. 12, сл.). В нем определен (но только **один**) объект **Elevator**, подобный тем, что создаются в строках **5—31** листинга 5.1. Здесь же метод **Main()** (строки **35—59**), в отличие от предыдущей программы, управляет **10** лифтами.

Чтобы сконцентрировать внимание только на важных вопросах, связанных с массивами объектов, в программе опущены некоторые аспекты, включенные в пример рассмотренный на Практ. занятии №3. Здесь нет, например, объектов **Person**, — вместо этого лифты управляются непосредственно из метода **Main()**, где доступ к объектам **Elevator** осуществляется из массива.

В строках **5—31** определен "скелет" класса **Elevator**. Он следит за своим текущим положением (переменная **currentFloorNumber**, объявленная в строке **7**) и общим числом пройденных этажей (**floorsTraveled** в строке **8**). Конструктор, определенный в строках **10**-**14**, инициализирует обе переменные экземпляра значением **нуль**. (Напомним, что конструктор автоматически вызывается в процессе создания нового объекта и представляет собой удобный механизм для инициализации.)

Объект **Elevator** может перемещаться на новый этаж (**MoveToFloor** в строках **16**-**20**), возвращать общее число пройденных этажей (**GetFloorsTraveled** в строках **22**-**25**) и свое текущее положение (строки **27**-**30**). Теперь можно создать большую группу объектов **Elevator** и управлять ей. Для этого применяется массив базового типа **Elevator**. Программа обрабатывает **10 объектов** этого типа (в методе **Main()**). Для объявления массива достаточно использовать стандартный синтаксис, где вместо имени одного из простых типов применяется **Elevator**. Таким образом, оператор

```
Elevator [ ] elevators
```

в строке **37** определяет переменную **elevators** как массив базового типа **Elevator**. Пока она ссылается на **null**, поэтому далее следует создать массив объектов **Elevator** длиной **10** и присвоить ей ссылку на него.

Обычно для этого применяется ключевое слово **new**. После выполнения строки **37**

```
Elevator [ ] elevators = new Elevator[10];
```

переменная **elevators** ссылается на объект массива с десятью элементами. Но поскольку ни одному из них еще не присвоена ссылка на конкретный объект **Elevator**, они равны **null**.

Для каждого элемента массива **elevators** нужно создать объект **Elevator** и присвоить элементу ссылку на него (строки **38—41**). Это производится в цикле **for**, где индекс **i** пробегает значения от **0** до **9** и каждому элементу **elevators** присваивается ссылка на объект **Elevator** (строка **38**):

```
elevators[ i ] = new Elevator();
```

После этого каждый элемент массива ссылается на объект **Elevator**.

```
for (int i=0; i<elevators.Length; i++)
{
elevators[ i ] = new Elevator();
}
```

Теперь **10** элементов массива **elevators** можно использовать в программе. Для идентификации конкретного объекта **Elevator** необходимо указать индекс в квадратных скобках после имени массива. Например, первый объект:

```
elevators[0]
```

Для вызова метода объекта применяется операция уточнения, за которой следует имя метода (строка **45**):

```
elevators[0].MoveToFloor(10);
```

Здесь вызывается метод **MoveToFloor** первого объекта **Elevator** из массива **elevators** с аргументом **10** (10-й этаж).

В строках **45—48** первый и пятый лифты "перемещаются" несколько раз. Можно было бы предусмотреть более сложный алгоритм перемещений, включающий все **10** лифтов.

В строках **50**-**57** вызываются методы **GetFloorsTraveled** и **GetCurrentFloorNumber** каждого объекта **Elevator**, возвращающие статистику объекта. Пример вывода подтверждает, что в системе перемещаются только первый и пятый лифты.

1.2. Листинг 5.1. Исходный код ElevatorArray.cs.

```
01: using System;  // Пространство имен
02: /*******************************************************/
03: namespace ConsoleApplication_Mic10_13_c382_ElevArr
04: {
05:     class Elevator                    //  Elevator - это пользовательский класс; Elevator - это объект
06:     {
07:     private int currentFloorNumber;    // текущее положение лифта (номер этажа)
08:     private int floorsTraveled;        // количество пройденных этажей
09:
10:     public Elevator()  // конструктор инициализирует обе перем-е экз-ра значением нуль
11:     {                  // констр-р автомат-ки выз-ся при созд-нии объекта и удобен для инициализации
12:     currentFloorNumber = 0;
13:      floorsTraveled = 0;
14:     }                                                  // далее вызывается метод Main (строка 33)
15:                                                        //  toFloorNumber - номер этажа назначения.
16:     public void MoveToFloor(int toFloorNumber)                 // В методе рассчитываются:
17:     {                                                  // 1) общее число пройденных этажей,
18: floorsTraveled += Math.Abs(currentFloorNumber - toFloorNumber);
19: currentFloorNumber = toFloorNumber;                    // 2) и  текущее положение лифта
20:      }
21:
22:     public int GetFloorsTraveled()          // возврат общего числа пройденных этажей
23      {
24:     return floorsTraveled;
25:      }
26:
```

```
27:     public int   GetCurrentFloorNumber()     // возврат текущего положения лифта (номера этажа)
28:      {
29:      return currentFloorNumber;
30:      }
31:    }
32: /*****************************************************/
33: class ElevatorArray
34: {
35:      public static void Main()
36:     {                    // для эффек-го управ-ния коллекцией лифтов исполь-ся массив 10 объектов
37:      Elevator [ ] elevators = new Elevator[10];
38:     for (int i=0; i<elevators.Length; i++)
39:      {
40:      elevators[ i ] = new Elevator();  // Создание массива объектов  Elevator[10]
41:      }                                  // Перем-я elevators ссыл-ся на объект массива с 10-ю элем-ми
42:
43:
44:                                      // 1-й [0] и 5-й [4] лифты "перемещаются" несколько раз
45: elevators[0].MoveToFloor(10);//выз-ся м-д MoveToFloor 1-о об-та Elevator из мас-ва elevators с арг-м 10
46: elevators[4].MoveToFloor(20);     // и т.д.       -'-       5-го              -'-              20
47: elevators[0].MoveToFloor(5);      // и т.д.       -'-       1-го              -'-               5
48: elevators[4].MoveToFloor(7);      // и т.д.       -'-       5-го              -'-               7
49:                  // факт. арг-нт 10, 20, 5 или 7 (этаж назн-я) присваивается форм. арг-ту toFloorNumber
50: Console.WriteLine("Общее число перемещений и текущее положение лифта:\n");
51:   Console.WriteLine("          Число перемещений Текущее положение"); //
52:  for (int i=0; i<elevators.Length; i++)
53:   {                                                    // возврат статистики объекта
54:  Console.WriteLine("Лифт     {0,2}:    {1,4}        {2,4}",
55:    (i + 1), elevators[ i ].GetFloorsTraveled(),              // возврат статистики объекта
56:     elevators[ i ].GetCurrentFloorNumber());              //          -'-
57:    }
58:  Console.ReadLine();
59:     }
60:  }
61: }
```

Результаты работы программы

Общее число перемещений и текущее положение лифта:

	Число перемещений	Текущее положение
Elevator **1:**	**15**	**5**
Elevator 2:	0	0
Elevator 3:	0	0
Elevator 4:	0	0
Elevator **5:**	**33**	**7**
Elevator 6:	0	0
Elevator 7:	0	0
Elevator 8:	0	0
Elevator 9:	0	0
Elevator 10:	0	0

1.3. Массивы как переменные экземпляра в классах

Классы, рассмотренные на всех практических занятиях, имеют следующий стандартный вид (см. Материалы к Прак. занятию №1, рис. 1.5):

```
class AnyClass
{
   <Переменные экземпляра>
    <Методы>
   <Другое>
}
```

Переменные **экземпляра**, используемые в примерах, состояли, в основном, из простых типов (см. Материалы к Прак. Занятию №4, стр. 45). Но они могут быть и объектами, к примеру, класса **System.Array**. Это иллюстрируется в следующем примере.

Рассмотрим моделирующую программу, где определен класс **Building**. Предположим, требуется, чтобы он "содержал" объекты **Elevator** и **Floor** (как реальное здание содержит этажи и лифты). Для этого в определении класса **Building** необходимо объявить две переменных экземпляра типа **System.Array elevators** и **floors**:

```
class Building
{
private Elevator [ ] elevators;
private Floor [ ] floors;
<Другие переменные экземлляра>
<Методы>
<Другое>
}
```

Теперь им можно присвоить объекты массива с определенным числом лифтов и этажей (с помощью одного из методов класса **Building**). Например, объект **Building** здания, имеющего **10** лифтов и **30** этажей, содержит следующие операторы инициализации **elevators** и **floors**:

```
elevators = new Elevator[10];
floors = new Floor[30];
```

Рассмотрим другой пример, связанный с **банками** и **балансами счетов**.

Массив **accountBalances** использовался здесь для демонстрации различных свойств массива. Коллекция балансов счетов (представленная массивом простого типа **decimal**) как переменная экземпляра в классе **Bank** хорошо иллюстрирует синтаксис и семантику массивов в простом контексте. Однако, учитывая сложность ПО для реального банка, необходимо воспользоваться более объектно-ориентированным подходом. Как это сделать? Банк можно рассматривать не просто как список счетов, а как хранилище их коллекции. **Для этого нужно создать:**

1) класс **Account**:

```
class Account
{
<Переменная экземпляра>
<Методы>
}
```

и **2)** класс **Bank** с коллекцией объектов **Account**:

```
class Bank
{
private Account [ ] accounts;
<Другие переменные экземпляра>
<Методы>
 }
```

В следующем примере показана программа моделирования банка, где массив счетов используется как переменная экземпляра класса **Bank**.

1.4. Программа моделирования работы банка

Полный исходный код программы (обсуждается ниже) приведен на листинге 5.2.

1.5. Спецификации программы:

Программа моделирует работу банка, содержащего несколько счетов. Доступ и управление ими осуществляется посредством простого пользовательского интерфейса в окне консоли. С помощью простых команд пользователь должен иметь возможность -

1. Указать исходное число счетов в банке ; **строки 158 → 060 - 069**
2. Положить средства на указанный счет ; **# 1 , строки 200, 165, 166**
3. Снять средства с указанного счета ; **# 2 , строки 201, 168, 169**
4. Установить процентную ставку указанного счета ; **# 3 , строки 202, 171, 172**
5. Добавить проценты ко всем счетам ; **# 4 , строки 203, 174, 175**
6. Вывести балансы всех счетов ; **# 5 , строки 204, 177, 178**
7. Вывести сумму процентов, начисленных по каждому счету ; **# 6 , строки 205, 180, 181**
8. Вывести процентную ставку по каждому счету ; **# 7 , строки 206, 183, 184**
9. Завершить моделирование ; **строки 207, 186, 187**

2. Разработка программы моделирования работы банка

2.1. Разделение каждой подсистемы на **объекты** (**модули**):

Два очевидных **класса** программы — **Account** (строки **6**-**54**) и **Bank** (**строки 56**-**148**), а также третий **класс BankSimulation** (**строки 150**-**210**). Этот класс содержит метод **Main()**, который объединяет различные части программы и обрабатывает пользовательский ввод.

2.2. Идентификация переменных экземпляров классов:

Согласно спецификациям программы, объект **Account** должен хранить следующую информацию:

a) свой баланс

b) свою текущую процентную ставку

c) сумму начисленных процентов

Следовательно, класс **Account** должен содержать переменные экземпляра (**строки 8**-**10**):

a) private decimal **balance**;

b) private decimal **currentInterestRate**;

c) private decimal **totalInterestPaid**;

Класс **Bank** содержит коллекцию счетов. Вся необходимая ему информация содержится в объектах этой коллекции. **Следовательно, классу Bank требуется единственная переменная экземпляра — массив счетов**. Ее объявление выглядит следующим образом (**строка 58**):

```
private Account [ ] accounts;
```

Классу **BankSintulation** также требуется только одна переменная экземпляра — объект **Bank** (объявленный **в строке 152**).

2.3. Идентификация методов в каждом модуле:

2.3.1. Класс **Account**. Все переменные экземпляра должны быть инициализированы во время создания объекта, которому они принадлежат. Для этого предназначен конструктор в строках **12—17**. (Конструктором является метод с тем же именем, что и класс, в котором он находится.) Он выполняется на этапе создания нового объекта **Account**. В данном примере конструктор инициализирует все переменные экземпляра нулевыми значениями.

Все переменные экземпляра **Account** объявлены как **private**, чтобы предотвратить доступ к ним извне объекта. Как следствие, для их **чтения** и **установки** значения необходимо определить специальные методы, называемые "**аксессор**" и "**мутатор**", соответственно. Для переменной **currentInterestRate** определены два метода: **мутатор SetInterestRate** (**строки 19—22**) и **аксессор GetInterestRate** (**строки 24—27**). Значение переменной **balance** изменяется с помощью **мутаторов Deposit** (**строки 40—43**) и **Withdraw** (строки **45—48**), а возвращается — **аксессором GetBalance** (**строки 50-53**). Для переменной **totalInterestPaid мутатором** является **UpdateInterest** (**строки 29-33**), а **аксессором** — **GetTotalInterestPaid** (**строки 35-38**).

2.3.2. Класс **Bank**. При создании нового объекта **Bank** нужно создать и объекты **Account**, которыми он будет управлять, и присвоить ссылки на них элементам массива accounts. Эту задачу можно было бы выполнить стандартным методом, вызвав его после создания объекта **Bank**. Но, поскольку эта процедура должна производиться для любого нового объекта **Bank**, ее может выполнить конструктор класса (строки **60-69**). При создании нового объекта **Bank** (строка **158**), у пользователя автоматически запрашивается количество счетов в этом банке. Следует обратить внимание, что заданное число счетов не может быть изменено во время исполнения программы.

После этого объекты счетов в массиве **accounts** готовы к доступу. Все методы класса Bank предназначены для изменения или чтения информации одного счета или всей коллекции счетов и, следовательно, зависят от методов класса **Account**.

Перечисленные ниже методы работают похоже: все они позволяют пользователю ввести номер нужного счета и выполнить операции над ним.

a) **Deposit** (строки **71-82**)
b) **Withdraw** (строки **84-95**)
c) **SetInterestRate** (строки **97-106**)

Следующие четыре метода проходят по всей коллекции счетов либо для обновления конкретной переменной экземпляра в каждом объекте Account, либо для вывода информации:

a) **PrintAllInterestRates** (строки **108-116**)
b) **PrintAllBalances** (строки **119-127**)
c) **PrintTotalInterestPaidAllAccounts** (строки **129-137**)
d) **UpdateInterestAllAccounts** (строки **139-147**)

2.3.3. Класс **BankSimulation**. Этот класс является контейнером для объекта **Bank** и методов **Main()** и **PrintMenu**. Последний содержит цикл **do-while** и оператор **switch**, обрабатывающие команды пользователя. Следует отметить, что метод **Main()** в такой схеме достаточно прост.

Все основные задачи переданы другим методам, находящимся либо внутри класса **BankSimulation**(метод **PrintMenu**), либо внутри других объектов (**Bank** и **Account**).

3. Разработка внутренних методов:

Методы данной программы не содержат каких-либо сложных алгоритмов, поэтому здесь упоминаются только два незначительных вопроса (связанных с классом **Bank**):

а) Элемент **accounts[0]** содержит ссылку на объект **Account**. Метод этого объекта вызывается посредством стандартного синтаксиса:

```
accounts[0].GetBalance()
```

б) Для пользователя первый счет имеет индекс **1** (строка **208**). Поэтому вводимые или выдаваемые пользователю индексы счетов следует корректировать соответствующим образом. Например, в методе **Deposit** (строки **71-82**) пользователь вводит номер счета, на который нужно перевести средства. Он присваивается переменной **accountNumber**, но так как массив индексируется с нуля, в строках **79** и **81** из нее вычитается **1**.

4. Листинг 5.2. Исходный код **BankSimulation.cs.** Программа моделирования работы в банке

```
001: using System; // Программа реализует доступ и управление несколькими счетами
002:
003: namespace ConsoleApplication_ BankSimulation
004: {
005: /*****************************************************/
006: class Account  // Account (счет) - это пользовательский класс; Account - это объект
007: {                    // Объект Account должен хранить (т.е. содержать) след-е переменные экземпляра:
008: private decimal balance;                  // 1) свой баланс (то есть баланс данного счета),
009: private decimal currentInterestRate;    // 2) свою текущую процентную ставку
010: private decimal totalInterestPaid;       // 3) сумму начисленных процентов
011:
012: public Account() // Это конст-р Account, т.е. м-д с тем же им-ем, что и класс Account, в кот-м он нах-ся
013: {                      // Назначение конст-ра - инициализация переменных экземляра  класса Account
014:     balance = 0;
015:     currentInterestRate = 0;
016:     totalInterestPaid = 0;
017: }
018:                          //  # 3.1.1
019: public void SetInterestRate(decimal newInterestRate) // М-д (мутатор) – уст-вить проц-ную ставку
020: {
021:     currentInterestRate = newInterestRate;
022: }
023:
024: public decimal  GetInterestRate()     // Метод (аксессор) - получить процентную ставку   # 4.1.2
025: {                                                                                          //    # 7.1.1
026:     return  currentInterestRate;
027: }
028                          // # 4.1.3
029: public void  UpdateInterest()  // Метод (мутатор) : 1) определение суммы начисленных процентов
030: {                                    // 2) определение баланса с учетом суммы начисленных процентов
031:     totalInterestPaid += balance * currentInterestRate;
032:     balance += balance * currentInterestRate;
```

```
033: }
034:                                    //   # 6.1.1
035: public decimal GetTotalInterestPaid()  // М-д (аксессор) - получить значение суммы начис-х про-тов
036: {
037:     return totalInterestPaid;
038:}
039:                                    //  # 1.1.1
040: public void Deposit(decimal amount)  // М-д (мутатор) - определение баланса (положить вклад)
041: {
042:     balance += amount;
043: }
044:                                    //  # 2.1.1
045: public void Withdraw(decimal amount) // М-д (мутатор) - определение баланса (снять со счета)
046: {
047:     balance -= amount;
048: }
049:
050: public decimal GetBalance()              //   # 4.1.1 - М-д (аксессор) - получение значения баланса
051: {                                        //   # 1.1.2   ;   # 2.1.2   ;     //   # 5.1.1
052:     return balance;
053: }
054: }
055: ////////////////////////////////////////////////////конец класса  Account//////////////////////////////////////////////////
056: class Bank     // кл-с Bank сод-жит колл-цию сч-ов. Все м-ды кл-са пред-ны для изменения или чтения
057: {  // информации одного счета или всей коллекции счетов и, след-но, зависят от мет-ов класса Account
058: private Account [ ] accounts; // кл-су Банк треб-ся только одна перем-я экз-ра – мас-в счетов accounts
059:
060: public Bank() // Констр-р Bank – созд-т новые объекты Account, которыми класс Bank будет управлять
061:  {
062:     Console.WriteLine("Поздравляем! Вы создали новый банк");
063:     Console.Write("Пожалуйста, введите количество счетов в банке: ");
064:     accounts = new Account[Convert.ToInt32(Console.ReadLine())];
065:     for (int i = 0; i < accounts.Length; i++)
066:     {                  // присв-е ссылок на объекты Account элементам массива accounts, строки 012…017
067:     accounts[i] = new Account();
068:     }
069:  }
070:
071: public void Deposit() // Положить средства на указанный счет                    # 1.1
072: {
073:     int accountNumber; // индекс мас-ва accounts. Для польз-ля первый инд-с = 1, для прог-мы - нулю.
074:     decimal amount;
075:     Console.Write("Положить средства. Пожалуйста, введите номер счета: ");
076:     accountNumber = Convert.ToInt32(Console.ReadLine());
077:     Console.Write("Введите объем вклада: ");
078:     amount = Convert.ToDecimal(Console.ReadLine());   // amount - объем вклада
079:     accounts[accountNumber - 1].Deposit(amount);  // 040 – 043; мас-в accounts инд-ся с нуля, поэтому
080:     Console.WriteLine("Новый баланс счета {0}: {1:C}",    // из инд-а выч-ся 1 [accountNumber - 1].
081:             accountNumber, accounts[accountNumber - 1].GetBalance());   // 050 – 053
```

```
082: }
083:
084: public void Withdraw() //  Снять средства с указанного счета                    # 2.1
085: {
086:     int accountNumber;
087:     decimal amount;
088:     Console.Write("Снять средства. Пожалуйста, введите номер счета: ");
089:     accountNumber = Convert.ToInt32(Console.ReadLine());
090:     Console.Write("Введите объем снимаемых средств: ");
091:     amount = Convert.ToDecimal(Console.ReadLine()); // amount - объем снимаемых средств со счета
092:     accounts[accountNumber - 1].Withdraw(amount);                    //  045 - 048
093:     Console.WriteLine("Новый баланс счета {0}: {1:C}",
094:             accountNumber, accounts[accountNumber - 1].GetBalance());   //  050 - 053
095: }
096:
097: public void SetInterestRate()  //  Установить процентную ставку указанного счета   # 3.1
098: {
099:     int accountNumber;
100:     decimal newInterestRate;
101:     Console.Write("Установите процентную ставку. Пожалуйста, введите номер счета: ");
102:     accountNumber = Convert.ToInt32(Console.ReadLine());
103:     Console.Write("Введите процентную ставку: ");
104:     newInterestRate = Convert.ToDecimal(Console.ReadLine());
105:     accounts[accountNumber - 1].SetInterestRate(newInterestRate);          //  019 - 022
106: }
107:
108: public void PrintAllInterestRates()   //   Вывести процентную ставку по каждому счету # 7.1
109: {
110:     Console.WriteLine("Процентная ставка для всех счетов: ");
111:     for (int i = 0; i < accounts.Length; i++)
112:     {
113:             Console.WriteLine("Счета {0,-3}: {1,-10}",
114:                     (i+1), accounts[ i ].GetInterestRate());                    //  024 - 027
115:     }
116: }
118:
119: public void PrintAllBalances()       //   Вывести балансы всех счетов  # 5.1
120: {
121:     Console.WriteLine("Баланс счета для всех счетов: ");
122:     for (int i = 0; i < accounts.Length; i++)
123:     {
124:             Console.WriteLine("Счета {0,-3}: {1:C}",
125:                     (i+1), accounts[ i ].GetBalance());              //   050 - 053
126:     }
127: }
128:
129: public void PrintTotalInterestPaidAllAccounts()  // Выв-ти сумму проц-в, начисл-х по кажд. сч-у  # 6.1
130: {
131:     Console.WriteLine("Общая процентная ставка, оплаченная за каждый индивидуальный счет:");
```

```
132:    for (int i = 0; i < accounts.Length; i++)
133:    {
134:            Console.WriteLine("Счета {0,-3}: {1:C}",
135:                    (i+1), accounts[ i ].GetTotalInterestPaid());                    //   035 - 038
136:    }
137: }
138:
139: public void UpdateInterestAllAccounts()     //  Добавить проценты по всем счетам  # 4.1
140: {
141:    for (int i = 0; i < accounts.Length; i++)
142:    {
143:            Console.WriteLine("Процентная ставка, добавленная к счету номер {0,-3}: {1:C}",
144:                                    (i+1), accounts[i].GetBalance() * accounts[i].GetInterestRate());
145:            accounts[ i ].UpdateInterest();              // 050 – 053;    024 – 027;    029 - 033
146:    }
147: }
148: }
149: ////////////////////////////////////////////////////конец класса  Bank/////////////////////////////////////////////////
150: class BankSimulation // класс Моделирование Банка
151: {
152: private static Bank bigBucksBank; // кл-су BankSimulation треб-ся одна пер-я экз-ра - объект Bank
153:
154: public static void Main()
155:    {
156:    string command;
157:
158:    bigBucksBank = new Bank(); // Создание нового объекта Bank, строки 060…069
159:    do
160:    {
161:            PrintMenu();
162:            command = Console.ReadLine().ToUpper();
163:            switch(command)
164:            {
165:            case "D":
166:                    bigBucksBank.Deposit();                    // 071 – 082 , # 1
167:                    break;
168:            case "W":
169:                    bigBucksBank.Withdraw();                   // 084 - 095,   # 2
170:                    break;
171:            case "S":
172:                    bigBucksBank.SetInterestRate();            // 097 – 106 , # 3
173:                    break;
174:            case "U":
175:                    bigBucksBank.UpdateInterestAllAccounts();  // 139 – 147 , # 4
176:                    break;
177:            case "P":
178:                    bigBucksBank.PrintAllBalances();           // 119 - 127,  # 5
179:                    break;
180:            case "T":
```

```
181:                    bigBucksBank.PrintTotalInterestPaidAllAccounts();  // 129 – 137 , # 6
182:                    break;
183:            case "I":
184:                    bigBucksBank.PrintAllInterestRates();                  // 108 – 116 , # 7
185:                    break;
186:            case "E":
187:                    Console.WriteLine("Bye Bye!");
188:                    break;
189:            default:
190:                    Console.WriteLine("Неправильный выбор");
191:                    break;
192:            }
193:    } while (command != "E");
194:    Console.ReadLine();
195: }//////////////////////////////////////////////////конец метода  Main//////////////////////////////////////////////////
196:
197: private static void PrintMenu()
198: {
199: Console.WriteLine("\nЧто Вы желаете сделать?\n" +
200: "D)eposit - положить средства на указанный счет\n" +
201: "W)ithdraw - снять средства с указанного счета\n" +
202: "S)et interest rate - установить процентную ставку указанного счета\n" +
203: "U)pdate all accounts for interest - добавить проценты ко всем счетам\n" +
204: "P)print all balances - вывести балансы всех счетов\n" +
205: "T)otal interest paid printed for all accounts – выв-ти сумму проц-тов, нач-х по каж-му счету\n" +
206: "I)nterest rates printed for all accounts - вывести процентную ставку по каждому счету\n" +
207: "E)nd session - завершить моделирование\n" +
208: "Note: First account has account number one - первый счет имеет индекс один");
209: }
210: }//////////////////////////////////////////////////конец класса  BankSimulation//////////////////////////////////////////////////
211: }
```

4.1. Результаты работы программы

```
Поздравляем! Вы создали новый банк
Пожалуйста, введите количество счетов в банке: 2

Что Вы желаете сделать?
D)eposit - положить средства на указанный счет
W)ithdraw - снять средства с указанного счета
S)et interest rate - установить процентную ставку указанного счета
U)pdate all accounts for interest - добавить проценты ко всем счетам
P)print all balances - вывести балансы всех счетов
T)otal interest paid printed for all accounts - вывести сумму процентов, начисленных
                                                по каждому счету
I)nterest rates printed for all accounts - вывести процентную ставку по каждому счету
E)nd session - завершить моделирование
Note: First account has account number one - первый счет имеет индекс один
d
Положить средства. Пожалуйста, введите номер счета: 1
Введите объем вклада: 100000
Новый баланс счета 1: 100 000,00р.
```

…..
d
Положить средства. Пожалуйста, введите номер счета: 2
Введите объем вклада: **200000**
Новый баланс счета 2: **200 000,00р**.
…..
s
Установите процентную ставку. Пожалуйста, введите номер счета: **1**
Введите процентную ставку: **0,1**
…..
s
Установите процентную ставку. Пожалуйста, введите номер счета: **2**
Введите процентную ставку: **0,2**
…..
u
Процентная ставка, добавленная к счету номер **1** : **10 000,00р**.
Процентная ставка, добавленная к счету номер **2** : **40 000,00р**.
…..
p
Баланс счета для всех счетов:
Счета **1** : **110 000,00р**.
Счета **2** : **240 000,00р**.
…..
t
Общая процентная ставка, оплаченная за каждый индивидуальный счет:
Счета **1** : **10 000,00р.**
Счета **2** : **40 000,00р**.
…...
i
Процентная ставка для всех счетов:
Счета **1** : **0,1**
Счета **2** : **0,2**
……
w
Снять средства. Пожалуйста, введите номер счета: **1**
Введите объем снимаемых средств: **1000**
Новый баланс счета **1**: **109 000,00р.**
…..
w
Снять средства. Пожалуйста, введите номер счета: **2**
Введите объем снимаемых средств: **2000**
Новый баланс счета **2**: **238 000,00р.**
…..

5. Двумерный массив

Вернемся к программе моделирования лифта (см. Пособие к Практическому занятию №3, стр. 12, сл.). Запрос нужного этажа определяется следующим образом: "человек" запрашивает доставку его на конкретный "этаж" путем "нажатия" на "кнопку" в "лифте". Предположим, нужно

отследить количество вызовов в час в течение семи дней. Обычный человек мог бы просто записать всю статистику (см. табл. 5.1). В этом случае в таблицу уже занесены данные по двум первым часам первого дня: 89 и 65 запросов, соответственно.

Для идентификации данных по каждому часу необходимо задать два индекса — **день** и **час**. Поэтому таблица оказывается двумерной. В данном случае (**day: 1, hour: 0**) равняется 89.

Таблица 5.1. Статистика собранная вручную (количество вызовов в час).

	Час 0	**Час 1**	**Час 2**	**Час 3**	**…**	**Час 23**
День 1	**89**	**65**		**…**		
День 2			**…**			
…	**…**	**…**	**…**	**…**	**…**	**…**
День 7						

П р и м е ч а н и е: Час 0 является часом между полуночью 00:00 и первым часом ночи 01:00.

5.1. Объявление и определение двумерного массива

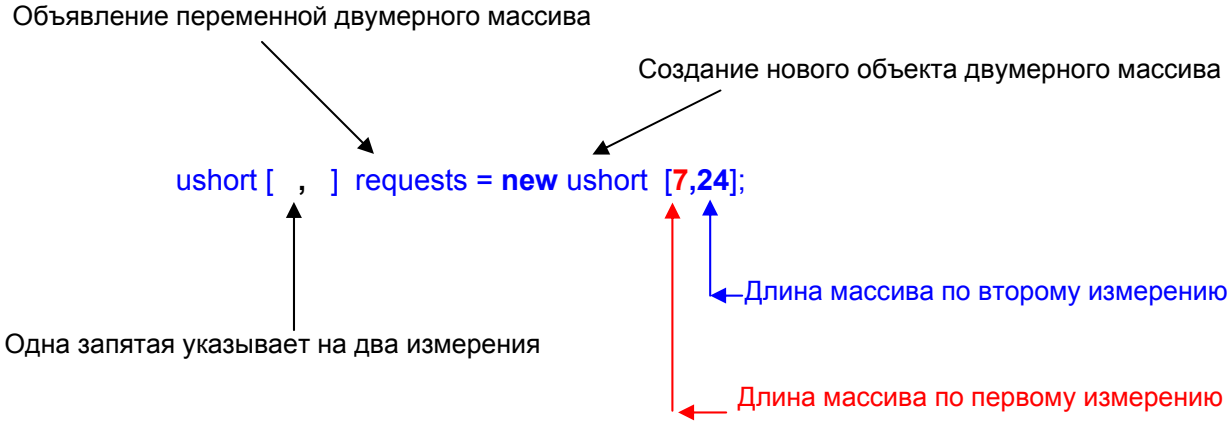
Как эффективно реализовать эквивалент системы отслеживания вызовов лифта в программе C#? Прежде всего требуется структура для хранения данных, показанных в табл. 5.1. Если воспользоваться двумерным массивом, можно назначить **один индекс для строки** и **один — для столбца**, и выражать **день 4**, **час 7** как [**3**,**7**]. (Следует отметить, что дню 4 соответствует значение индекса равное 3, а часу 7 — индекс 7, поскольку первый час обозначен 0, как показано в таблице 5.1.)

//ПРИМЕЧАНИЕ

В многомерном массиве (как и в одномерном) индексация тоже начинается с нуля. Следовательно, первым элементом в двумерном массиве является элемент [0,0].

Объявление и определение двумерного массива **requests** типа **ushort** показано на рис. 5.1.

При создании нового массива с помощью ключевого слова **new** внутри пары квадратных скобок требуется одна запятая, указывающая на два измерения, а два разделенных запятой числа (**7** и **24**) определяют длины измерений массива.



Предыдущий оператор можно разбить на две строки:

```
ushort[ , ] requests;
requests = new ushort [7,24];
```

Рис.5.1. Объявление и определение двумерного массива запросов лифта

Объявление переменной массива, создание нового объекта и присваивание переменной ссылки на объект (первая строка на рис. 5.1) можно (как и в случае одномерного массива) разбить на два оператора. Результат показан в нижней части рис. 5.1.

5.2. Доступ к элементам двумерного массива

После объявления переменной массива и присваивания ей ссылки на двумерный объект можно обращаться к его отдельным элементам.

За исключением того, что для обращения к элементам двумерного массива требуется дополнительный индекс, они используются аналогично элементам одномерного массива. Следовательно, любой из них можно использовать так же, как и отдельную переменную базового типа. Например:

```
requests[0,0] = (ushort) 89;
```

Здесь применяется приведение к типу (**ushort**), поскольку он является базовым типом **requests**. Оно необходимо, так как литерал **89** принадлежит к типу **int**, который не может быть неявно преобразован в **ushort**.

Элементы массива используются и в вычислениях. В следующей строке общее число запросов **за первые три часа четвертого дня** присваивается переменной **subTotal**:

```
subTotal = (ushort) (requests[3,0] + requests[3,1] + requests[3,2]);
```

Как будет видно из листинга 5.5, два вложенных цикла хорошо подходят для прохождения по элементам двумерного массива. Далее рассматривается логика, которой они следуют.

По одномерному массиву можно проходить с помощью одного цикла **for**, как показано в листинге 5.3.

Листинг 5.3. Один цикл **for** для прохождения по одномерному массиву.

```
…
int [ ] myArray = new int [24];
for (int j = 0; j < 24, j++)
{
    ...
    ... myArray[ j ] ...
    ...
 }
…
```

Переменная массива **requests** представляет **часы** не одного дня, а **семи**. Таким образом, повторяя показанный на рис. 5.4 процесс семь раз, можно обратиться к значениям часов всех семи дней. Такую функциональность можно создать, включив предыдущий цикл **for** в другой цикл **for**, который повторяется **семь** раз. Фрагмент исходного кода приведен в листинге 5.4.

Листинг 5.4. Два вложенных цикла **for** для обработки двумерного массива.

```
…
ushort [ , ] requests = new ushort [7,24];
…
for (int i = 0; i < 7; i++)
{
   …
   for (int j = 0; j < 24 ; j++)
   {
      …
      //этот блок повторяется семь раз
      ... requests[i, j] …
```

часы суток
дни недели

```
        ...
      }
    ...
  }
  ...
```

Массив **requests** в листинге 5.5 используется для хранения количества вызовов лифта в течение каждого **часа** за **семи**дневный срок. Для простоты программа генерирует количество вызовов в час с помощью класса **System.Random** (см. строки **12**, **22** и **24**). С **8** утра и до **6** вечера количество вызовов составляет от **20** до **99** (см. строку **24**), в другое время суток — от **1** до **10** (см. строку **22**).

Количество вызовов хранится в массиве **requests** (см. строку **11**) и выводится в таблице с **7-ю строками** и **24-мя столбцами**, как показано в примере вывода после листинга 5.3. Массив применяется для вычисления общего числа вызовов за день и среднего числа вызовов за час, которые также отображаются в выводе.

5.3. Листинг 5.5. Исходный код ElevatorRequestTracker.cs

```
01: using System;
02:
03: namespace ConsAppl_ElevatorRequestTracker
04: {
06:
07: class ElevatorRequestTracker
08: {
09; public static void Main()
10: {
11: ushort[ , ] requests = new ushort[7,24]; // В массиве requests хранятся количества вызовов
12: Random randomizer = new Random();      // лифта в течение каждого часа за семидневный срок
13: int sum;
14:
15:       // Случайно генерировать количество вызовов
16:       // для каждого часа каждого дня недели
17:       for (int i=0; i<7; i++)
18:       {
19:             for (int j=0; j<24; j++)
20:             {
21:                   if(( j < 8 ) || ( j >18 ))                    // вызовы лифта вечером и ночью
22:                         requests[i , j] = (ushort)randomizer.Next(1,10);
23:                   else
24:                         requests[i , j] = (ushort)randomizer.Next(20,99);  // вызовы лифта днем
25:             }
26:       }
27:
28: // Вывести таблицу вызовов
29: Console.WriteLine("\n\nКол-во вызовов лифта в течение каждого часа за 7-дневный срок:\n");
30: Console.WriteLine("                  Часы\n");
31: Console.Write("   ");
32: for (int i=0; i<24; i++)
33: {
34: Console.Write("{0,2} ", i);
```

```
35: }
36:
37:       Console.WriteLine("\nДни");
38:       for (int i=0; i<7; i++)
39:       {
40:                 Console.Write("\n{0}  ", (i+1));
41:                 for (int j=0; j<24; j++)
42:                 {
43:                 Console.Write("{0,2} ", requests[i , j]);
44:                 }
45:        }
46:
47:        // Вычислить и вывести общее число вызовов лифта в день
48:        Console.WriteLine("\n\nОбщее число вызовов лифта в день:\n");
49:       for (int i=0; i<7; i++)
50:       {
51:                 sum = 0;
52:                 for (int j=0; j<24; j++)      // Суммируются строки вызовов лифта за день
53:                           sum += requests[i , j];     // Общее число вызовов лифта в день
54:                 Console.WriteLine("дни {0}: {1}", (i+1), sum);
55:       }
56:
57: // Вычислить и вывести среднее число вызовов в час
58: Console.Write("\nСреднее число вызовов в час:\n\nЧасы:  ");
59: for (int i=0; i<24; i++)
60: {
61: Console.Write("{0,2} ",i);
62: }
63:       Console.Write("\nВызовы:");
64:       for (int j=0; j<24; j++)
65:       {
66:                 sum = 0;
67:                 for (int i=0; i<7; i++)    // Суммируются столбцы вызовов за каждый час
68:                           sum += requests[i , j];  // Среднее число вызовов в час
69:                 Console.Write("{0,2} ", (sum / 7));
70:       }
71: Console.ReadLine();
72: }
73: }
74: }
```

5.4. Результаты работы программы

Количество вызовов лифта в течение каждого **часа** за **7-дневный** срок:

```
                Часы
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Дни
1   8  6  1  3  9  4  3  9 94 33 55 25 32 49 38 32 25 28 37  1  7  2  3  9
2   9  4  4  4  6  5  4  6 46 80 98 28 75 92 57 82 89 84 65  5  3  9  2  9
3   9  8  7  1  9  7  1  6 44 34 48 62 20 86 51 55 81 88 77  4  4  2  5  2
```

```
4  6 4 2 9 6 8 7  1 40 60 59 80 81 50 71 66 73 49 83  5 6 9 8 1
5  4 2 7 4 2 7 5  7 48 65 49 59 64 34 22 61 58 47 42  5 4 6 7 4
6  2 2 7 1 3 9 6  7 71 32 57 56 26 46 85 61 97 76 52  2 1 2 1 5
7  4 3 1 2 6 3 7  8 55 44 30 22 21 60 55 53 97 72 87  4 7 6 4 2

Общее число вызовов лифта в день:
дни 1: 513
дни 2: 866
дни 3: 711
дни 4: 784
дни 5: 613
дни 6: 707
дни 7: 653

Среднее число вызовов в час:
Часы:     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
Вызовы:   6  4  4  3  5  6  4  6 56 49 56 47 45 59 54 58 74 63 63  3  4  5  4  4

// Примечание:
Поскольку количество вызовов лифта генерируется случайным образом, оно будет разным при каждом тестовом запуске программы.
```

В строке **11** объявляется переменная двумерного массива **requests**, а в строке **12** — переменная **randomizer**, содержащая ссылку на объект типа **Random** (или **System.Random**). Переменная **randomizer** подготовлена для генерирования случайных чисел. Строки **17—26** содержат два вложенных цикла **for**. Во внешнем цикле индекс **i** изменяется от **0** до **6** с шагом **1** (см. строку **17**), во внутреннем — **j** от **0** до **23** (см. строку **19**) с шагом **1**. Такая циклическая конструкция позволяет просматривать двумерный массив, обращаясь к каждому элементу как **requests[i, j]**. Именно это и выполняется в строках **22** и **24**, где **requests[i, j]** присваивается случайное число. При этом используется оператор **if** (в начале строки **21**). Если **j** выходит за пределы диапазона между **8** и **18** (**if** ((**j** < **8**) || (**j** > **18**)), элементу массива **requests** присваивается случайное число от **1** до **10** (строка **22**). Если **j** входит в указанный диапазон, элементу присваивается случайное число от **20** до **99** (строка **24**). Напомним, что метод **Next** объекта **randomizer** имеет два аргумента: нижний и верхний пределы для случайных чисел.

В строках **28—45** выводится таблица количества вызовов, содержащихся в массиве **requests**. В строках **28**-**35** печатаются заголовки столбцов, соответствующих **часам**, а в **37**-**45** вновь применяются вложенные циклы **for** — для вывода содержимого массива **requests**.

В строках **49—55** вычисляется общее количество вызовов в течение **дня**: **sum** устанавливается равной нулю (строка **51**), складываются все вызовы за **24** часа (строки **52** и **53**) и полученный результат выводится на консоль (строка **54**).

В строках **58**-**70** вычисляется и выводится на экран среднее число **вызовов в час за семь дней**. Вместо расчета суммы данных по строке таблицы (строки **49**-**55**) нужно просуммировать данные по столбцу и разделить результат на семь (строка **69**). При этом внутренний цикл **for** в строках **49—55** становится внешним, а внешний — внутренним, как показано в строках **64—70**.

5.5. Свободные ("зубчатые") массивы

Массивы двух (или более) измерений не должны быть обязательно прямоугольными, потому как строки массива не обязательно должны иметь одинаковую длину. Массивы, в которых различные строки имеют разное число столбцов, называются свободными, или "зубчатыми". Чтобы проиллюстрировать свободный массив, обратимся к прямоугольному массиву **requests** из лис-

тинга 5.5, который использовался для сбора вызовов лифта **за каждый час в течение семи дней**. Строка с объявлением такого двумерного массива выглядит следующим образом:

```
ushort [ , ] requests = new ushort [7, 24];
```

Предположим теперь, что требуется собрать вызовы лифта не за все **24** часа каждый день, а, скажем, за **18** часов в четвертый и пятый дни и за **12** часов — в шестой и седьмой дни. Для такого случая можно создать специальный массив. Такой свободный массив **requests** объявляется следующим образом:

```
ushort [ ]  [ ]  requests;
requests = new ushort  [7][ ];

requests[0]  = new ushort[24];
requests[1]  = new ushort[24];   массивы длиной 24 присваиваются первым 3-м элементам
requests[2]  = new ushort[24];

requests[3]  = new ushort[18];  массивы длиной 18 присваиваются 4-му и 5-му элементам
requests[4]  = new ushort[18];

requests[5]  = new ushort[12];  массивы длиной 12 присваиваются 6-му и 7-му элементам
requests[6]  = new ushort[12]
```

Первая строка

```
ushort [ ] [ ] requests;
```

должна содержать две пары квадратных скобок (указывающих на специфическое свойство массива), а в следующей строке

```
requests = new ushort [7][ ];
```

переменной **requests** присваивается ссылка на объект массива с такими характеристиками: он содержит **7** элементов (его длина равна **7**), каждый из которых является массивом **ushort** и может иметь любую длину.

Теперь каждому из этих семи элементов можно присвоить одномерный массив любой длины. Это производится в последующих семи строках кода. К примеру, в строке

```
requests[4] = new ushort [18];
```

одномерный массив длиной **18** присваивается **пятому** элементу массива **requests**.

Строки двумерного массива имеют разную длину, поэтому программисты называют его свободным или "зубчатым". Это понятие распространяется на массивы любой размерности. Если одно из измерений содержит одномерные массивы разной длины, такой массив также называется свободным.

Иногда массивы такого типа называют "рваными".

5.6. Листинг 5.6. Исходный код JaggedElevatorRequests.cs

```
01: using System; // Количество вызовов лифта генерируется случайным
02:                 // образом и присваивается элементам массива requests
03: namespace ConsAppl_JaggedElevatorReguests
04: {
05: class JaggedElevatorRequest  // "Зубчатый" запрос Лифта
06: {
07: public static void Main() //  Зубчатые массивы - различные строки имеют различное число столбцов
08: {
```

```
09:     Random randomizer = new Random();
10:     ushort [ ] [ ] requests;              // В массиве requests хранятся количества вызовов
11:     requests = new ushort[7] [ ];
12:     requests[0] = new ushort[24];
13:     requests[1] = new ushort[24];
14:     requests[2] = new ushort[24];
15:     requests[3] = new ushort[18];
16:     requests[4] = new ushort[18];
17:     requests[5] = new ushort[12];
18:     requests[6] = new ushort[12];
19:
20:     // Присваивание случайно сгенерированного количества вызовов
21:     // каждому элементу массива requests
22:     for (int i=0; i<requests.Length; i++)
23:     {
24:             for (int j=0; j<requests[ i ].Length; j++)
25:             {
26:                     if (( j < 8) || ( j >18 ))                    // вызовы лифта вечером и ночью
27:                             requests[ i ][ j ] = (ushort)randomizer.Next(1,10);
28:                     else
29:                             requests[ i ][ j ] = (ushort)randomizer.Next(20,99); // вызовы лифта днем
30:             }
31:      }
32:     // Вывод таблицы вызовов за каждый час ежедневно
33:     Console.WriteLine("                    Часы\n");
34:     Console.Write("   ");
35:     for (int i=0; i<24; i++)

36:     {
37:             Console.Write("{0,2} ", i);
38:     }
39:     Console.WriteLine("\nДни");
40:     for (int i=0; i<requests.Length; i++)
41:     {
42:             Console.Write("\n{0}  ", (i+1));
43:             for (int j=0; j<requests[ i ].Length; j++)
44:             {
45:                     Console.Write("{0,2} ", requests[ i ] [ j ]);
46:             }
47:     }
48:     Console.ReadLine();
49: }
50: }
51: }
```

5.7. Результаты работы программы

```
        Часы < 18                  8<=Часы <= 18          Часы > 18
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Дни
1   2  3  2  1  3  5  5  3 50 20 71 55 32 90 29 54 53 81 68  4  2  9  5  6
```

2 1 8 8 7 3 2 3 6 26 32 73 45 50 75 53 62 47 25 44 2 7 1 4 7
3 9 9 6 2 8 7 6 2 78 60 93 50 68 82 91 34 75 23 48 1 2 8 1 7

4 4 1 1 2 6 3 9 1 56 31 84 47 65 32 83 52 64 56
5 6 8 7 1 9 5 2 2 60 70 72 51 58 30 93 92 83 76

6 6 5 6 5 6 8 9 5 97 63 67 62
7 3 9 4 7 2 2 9 3 32 62 94 59

6. Анатомия класса

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

Ранее было показано, как при решении разных вычислительных задач объекты различных классов взаимодействуют друг с другом, внося свои уникальные свойства в общую программу. Конструкция класса позволяет объединять **данные** (называемые состоянием объекта) с **функциями** (представляющими его поведение) для создания объектов, составляющих структуру разрабатываемого ПО. До этого момента классы, используемые в примерах, состояли из переменных и методов экземпляра, как показано на рис. 1.5 (см. Материалы к Практич. занятию №1, стр.20).

Элементы класса являются языковыми конструкциями, составляющими тело класса. К примеру, переменные и методы экземпляра представляют собой два фундаментальных элемента класса. Однако классы настолько разнообразны, что **C#** содержит и несколько других элементов, придающих классу гибкость и расширяющих его возможности по взаимодействию с другими классами программы. Далее рассмотрим именно их, а также важные аспекты создания методов, которые ранее не обсуждались.

```
class <Идентификатор_класса>
{
<Переменные_экземпляра>
<Методы>
}
```

Рис. 5.2. Класс с переменными экземпляра и методами.

6.1. Анатомия класса: обзор

Сделаем обзор всех возможных элементов класса.

В синтаксическом блоке 5.1 расширен синтаксис, показанный на рис. 1.5 (см. М атериалы к Практич. занятию №1, стр.20), и описаны элементы, которые можно включать в определение класса. В первых строках отображен уже знакомый синтаксис: ключевое слово **class**, **имя** (идентификатор) **класса** и **фигурные скобки**, формирующие тело класса. Элементы класса можно разделить на три широкие категории: **1)** данные-члены, **2)** функции-члены и **3)** вложенные типы. Далее приведено их краткое описание. Данные-члены состоят из переменных-членов, констант и событий.

1) Переменные-члены (называемые также полями) используются для представления данных. Такая переменная может принадлежать: **1)** или конкретному экземпляру (объекту) класса — в этом случае она называется переменной экземпляра, **2)** или самому классу — тогда она называется статической (объявленной **static**) переменной (или переменной класса). Напомним, что статический метод принадлежит классу, а не объекту (и вызывается для класса). То же самое справедливо и для статической переменной.

2) Переменная-член может быть объявлена только для чтения (с ключевым словом **readonly**). Такие переменные тесно связаны с константами-членами, но существует важное различие —

значения последних устанавливаются в программе в процессе компиляции и существуют на протяжении ее исполнения. А значения readonly переменных-членов присваиваются им при создании объекта, и поэтому существуют, пока существует объект.

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

Синтаксический блок 5.1. Обзор элементов класса

Определение_класса::=
class <Идентификатор_класса>
{
  <**Члены_класса**>
}

где

1. <**Члены_класса**>
   1.1. ::= <Данные-члены>
   1.2. ::= <Функции-члены>
   1.3. ::= <Вложенные_типы>

1.1. <**Данные-члены**>
   1.1. 1. ::=<Переменные-члены>
   1.1. 2. ::=<Константы>
   1.1. 3. ::=<События>

1.2. <**Функции-члены**>
   1.2.1. ::= <Методы>
   1.2.2. ::= <Конструкторы>
   1.2.3. ::= <Деструктор>
   1.2.4. ::= <Свойства>
   1.2.5. ::= <Индексаторы>
   1.2.6. ::= <Операции>

1.3. <**Вложенные_типы**>
   1.3.1. ::= <Вложенные_классы>
   1.3.2. ::= <Вложенные_структуры>
   1.3.3. ::= <Вложенные_перечисления>

Примечания:

1. В приведенном здесь определении класса основное место занимают его внутренние особенности, поэтому в нем опущены синтаксические элементы, связанные с модификацией доступа, наследованием и интерфейсами.

2. Порядок элементов класса может быть любым внутри него, он не меняет семантики.

3. <Функция-член> может быть либо <Функция_экземпляра>, либо <Статическая_функция>

(называемая также <Функция_класса>). Функция экземпляра всегда исполняется по отношению к конкретному объекту, поэтому последний необходим для ее вызова.

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

1. **Метод** — знакомая конструкция, которая, тем не менее, обладает многими еще неизученными свойствами, как, например, ссылочные и выходные параметры, массивы параметров, перегрузка метода и ключевое слово **this**. Их изучению посвящена большая часть этой главы.

2. **Конструкторы** были рассмотрены ранее.

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

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

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

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

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

6.2. Переменные-члены

Ранее уже рассматривались *переменные экземпляра*, поэтому здесь речь пойдет о статических переменных и их использовании.

6.2.1. Переменные экземпляра

Когда создается объект, значения переменных экземпляра являются характерными именно для него. Они остаются таковыми в течение всего времени существования объекта. Этот механизм уже знаком, однако его стоит повторить, чтобы проследить отличие от статических переменных, обсуждаемых в следующем разделе.

Обратимся к программе **BankSimulation.cs** (листинг 5.2, стр. 10 -14). Она состояла из объекта **Bank**, содержащего несколько объектов **Account** в массиве **accounts**. В классе **Account** определена переменная экземпляра **balance**:

```
class Account
{
    …
    private decimal balance;
    …
}
```

Объект **Bank** может содержать любое количество объектов **Account** (**семь** в этом примере). Значение переменной **balance** любого объекта **Account** можно изменить (переводя средства на или со счета). В примере были изменены балансы первого и второго счетов (они стали равным **110 000,00р** и **240 000,00р**). Значение **balance** в остальных объектах **Account** было нулевым. Продолжая моделирование, можно было бы работать со всеми счетами, получив, таким образом семь балансов:

Счет **1**: **balance** составляет **110 000,00р**

Счет **2**: **balance** составляет **240 000,00р**

Счет **3**: **balance** составляет **35 000,00р**

Счет **4**: **balance** составляет **55 000,00р**

Счет **5**: **balance** составляет **300,00р**

Счет **6**: **balance** составляет **2 000,00р**

Счет **7**: **balance** составляет **3 500,00р**

Объект **Bank** ссылается на семь объектов **Account**, каждый из которых содержит свое уникальное значение переменной **balance**.

В следующем разделе рассмотрим важное различие между переменными **экземпляра** и **статическими** переменными.

6.2.2. Переменные static

Переменные экземпляра используются для хранения данных индивидуального объекта. Как быть, если имеется блок данных, связанный с группой объектов одного класса? Вернемся вновь к программе **BankSimulation.cs** (листинг 5.2). В ней количество счетов задается вначале и неизменно в течение исполнения программы. Предположим, что этого ограничения нет, т.е. в программе можно открывать и закрывать счета, следя за общим числом созданных объектов **Account**. Такой элемент данных описывает уже не свойство конкретного объекта (**как переменная экземпляра balance**), а **связан с группой объектов** класса **Account**. Воспользуемся переменной **экземпляра**, **totalAccountsCreated**:

```
class Account
{
…
public uint totalAccountsCreated;    // Неэффективно
…
}
```

При создании нового объекта **Account** необходимо увеличивать **totalAccountsCreated** на единицу. Такой подход имеет два недостатка:

1. ***Дублирование данных***. Значение **totalAccountsCreated** описывает общий атрибут объектов **Account**. Поэтому программе требуется лишь одна переменная **totalAccountsCreated**. Однако объявление **totalAccountsCreated**, приведенное ранее, создает копию этой переменной **в каждом экземпляре объекта Account**. Это избыточно. Избыточность состоит в том, что для семи объектов **Account** существуют **семь** переменных, содержащих одно и то же значение (**7**).

Это не только засоряет память, но и запутывает других программистов.

2. ***Из-за необходимости обновления всех объектов*** **Account** код становится неэффективным. В данном подходе при создании нового объекта объект **Bank** должен обновлять (прибавляя единицу к **totalAccountsCreated**) все объекты **Account**. Это несложно при небольшом количестве счетов, но в банке, где их число достигает сотен тысяч, оператору цикла потребуется обновить огромную коллекцию объектов **Account**.

Как же решить эти проблемы неэффективности? Значение **totalAccountsCreated** нужно хранить лишь в одном месте: за пределами отдельного объекта, но во взаимосвязи с группой объектов **Account**. Поскольку имеется только один класс **Account**, и он определяет атрибуты, общие для всех его объектов, **он является естественным местом, где следует хранить и обновлять значение totalAccountsCreated**. Именно для этого и предназначено ключевое слово **static**. Объявление переменной **totalAccountsCreated** как **static**

```
class Account
{
...
public static uint totalAccountsCreated;  // Эффективно
...
}
```

указывает, что:

1. существует только одна копия **totalAccountsCreated**, которая принадлежит всему классу **Account**, а не отдельным объектам
2. переменная **totalAccountsCreated** совместно используется всеми объектами **Account**
3. переменная **totalAccountsCreated** имеет определенное значение, даже если не создано ни одного объекта **Account**
4. значение **totalAccountsCreated** доступно из класса и из любого объекта **Account**

Хотя переменная **totalAccountsCreated** и не содержится в каждом конкретном экземпляре **Account**, она остается доступной и может быть изменена любым объектом этого класса. Ниже приведен фрагмент кода, где демонстрируются различия между переменной **экземпляра** **accountCreationNumber** и **статической** переменной **totalAccountsCreated**.

```
class Account
{
    ...
    private uint accountCreationNumber;
    public static uint totalAccountsCreated;
    ...
    public void accountsCreatedAfterMe()
    {
        Console.WriteLine("Кол-во счетов, создан-х в течение исполнения программы: {0}",
            totalAccountsCreated - accountCreationNumber);  //Одинаковый синтаксис
    }
    ...
}
```

Доступ к **totalAccountsCreated** извне объекта **Account** осуществляется посредством операции уточнения после имени класса (**Account**):

```
class Bank
{
    ...
    ... Account.totalAccountsCreated...
    ...
}
```

Такой способ доступа к **totalAccountsCreated** невозможен, если переменная объявлена как **private** (и возможен, если она объявлена как **public**).

//ПРИМЕЧАНИЕ

Для объявления переменной-члена **static** имеется два спецификатора — **public** и **private**. Как правило, следуя принципу инкапсуляции, все статические переменные объявляются как **private** (подобно переменным экземпляра). В целях демонстрации в данном разделе это правило нарушено: переменная **totalAccountsCreated** объявлена **public**. Если статическая переменная-член объявлена **private**, прочесть или изменить ее значение можно только с помощью стандартных методов, встречавшихся ранее, или методов **static** (которые также ассоциированы с классом, а не с его объектами).

//ПРИМЕЧАНИЕ

Статическая переменная-член (наподобие **totalAccountsCreated**) недоступна при применении операции уточнения к имени отдельного объекта:

```
class Bank
{
    ...
    Account myAccount = new Account();
    ...myAccount.totalAccountsCreated...      // неверно
    ...
}
```

Листинг 5.7 демонстрирует чтение и модификацию переменной-члена **totalAccountsCreated**, объявленной **public static** (строка **8**). Программа **DynamicBankSimulation.cs** содержит сокращенные версии трех классов — **Account**, **Bank** и **BankSimulation** — из программы **BankSimulation.cs** (листинг 5.2). В отличие от **BankSimulation.cs** программа **DynamicBankSimulation.cs** позволяет открывать и закрывать банковские счета в течение времени исполнения. Программа сохраняет информацию об общем числе созданных счетов (в переменной **totalAccountsCreated**) и их текущем количестве (переменная **bigBucksBank**).

6.3. Листинг 5.7. Исходный код DynamicBankSimulation.cs.

```
01: using System; // Прог-ма мод-ния работы в банке: доступ и управление несколькими счетами
02: using System.Collections;
03: namespace ConsAppl_DinamicBankSimulation
04: {
05:
06: class Account  // Account (счет)  - это польз-кий класс; Account - это объект  (см. файл Mic10_14c387.cs)
07: {             // Объект Account (счет) должен хранить след-ю статическую переменную:
08: public static uint totalAccountsCreated = 0;      // общее количество созданных объектов (счетов)
09:
10: public Account()  // Это кон-р Account, т.е. м-д с тем же именем, что и кл-с Account, в кот-м он наход-ся
11: {               // Назначение конст-ра - инициализация переменных экземляра  класса Account
12:       totalAccountsCreated++;
13: }
14: }             // окончание - class Account
15: /*****************************************************/
```

```
16: class Bank  // класс Bank содержит коллекцию счетов. Все м-ды класса Bank предн-ны для изме-ния или чтения
17; {          // инфор-ции одного счета или всей кол-ции счетов и, след-но, зависят от м-дов кл-са Account
18: private ArrayList accounts; // классу Bank треб-ся только одна перем-я экз-ра – мас-в счетов accounts
19:
20: public Bank() // Конст-р Bank - создает новые объекты Account, кот-ми класс Bank будет управлять
21: {
22: Console.WriteLine("Поздравляем! Вы создали новый банк");
23:     accounts = new ArrayList();
24: }
25:
26: public void AddNewAccount() // Добавьте Новый Счет
27: {
28:     accounts.Add(new Account());     //
29:     Console.WriteLine("Добавить новый счет!");
30:     PrintStatistics();                                    // строки 47- 51
31: }
32:
33: public void RemoveFirstAccount() // Первый Счет Удаления
34: {
35:     if (accounts.Count > 0)
36:     {
37:             accounts.RemoveAt(0);
38:             Console.WriteLine("Счет удален!");
39:             PrintStatistics();                            // строки 47- 51
40:     }
41:     else
42:     {
43:             Console.WriteLine("Извините, нет больше текущих счетов");
44:     }
45: }
46:
47: public void PrintStatistics()
48: {
49:     Console.WriteLine("Номер текущего счета: " + accounts.Count +
50:             "\nКоличество новых созданных счетов: " + Account.totalAccountsCreated);
51: }
52: }                   // окончание - class Bank
53: /*******************************************/
54: class DynamicBankSimulation // класс Динамическое Моделирование Банка
55: {    // классу  BankSimulation требуется только одна перем-я экз-ра - объект Bank
56: private static Bank bigBucksBank;
57:
58: public static void Main()
59: {
60:     string command;
61:
62:     bigBucksBank = new Bank(); // Создание нового объекта Bank
63:     do
64:     {
```

```
65:             PrintMenu();
66:             command = Console.ReadLine().ToUpper();
67:             switch(command)
68:             {
69:                     case "A":
70:                             bigBucksBank.AddNewAccount();       // строки 26- 31
71:                             break;
72:                     case "E":
73:                             Console.WriteLine("Bye Bye!");
74:                             break;
75:                     case "R":
76:                             bigBucksBank.RemoveFirstAccount();  // строки 33 - 45
77:                             break;
78:                     case "P":
79:                             bigBucksBank.PrintStatistics();         // строки 47- 51
80:                             break;
81:                     default:
82:                             Console.WriteLine(" неправильный выбор");
83:                             break;
84:                     }
85:     } while (command != "E");
86:      Console.ReadLine();
87: }
 88:
89: private static void PrintMenu()
90: {
91:     Console.WriteLine("\nЧто Вы желаете сделать?\n" +
92:     "A) Добавить новый счет\n" +
93:     "R) Удалить счет\n" +
94:     "P) Печать статистики\n" +
95:     "E) Сессия окончена\n");
96: }
97: }       // Окончание класса DynamicBankSimulation
98: }
```

6.4. Результаты работы программы

```
Поздравляем! Вы создали новый банк

Что Вы желаете сделать?
A) Добавить новый счет
R) Удалить счет
P) Печать статистики
E) Сессия окончена

p  <Enter>
Номер текущего счета: 0
Количество новых созданных счетов: 0

Что Вы желаете сделать?
...
```

a <Enter>
Добавить новый счет!
Номер текущего счета: **1**
Количество новых созданных счетов: **1**

Что Вы желаете сделать?
…
a <Enter>
Добавить новый счет!
Номер текущего счета: **2**
Количество новых созданных счетов: **2**

Что Вы желаете сделать?
…
r <Enter>
Счет удален!
Номер текущего счета: **1**
Количество новых созданных счетов: **2**

Что Вы желаете сделать?
…
e <Enter>
Bye Bye!

Примечание:
Для экономии места меню приведено только один раз, а затем заменено троеточием (**...**).

Для сокращения кода класс **Account** не содержит переменной **экземпляра balance**. В строке **8** переменная-член **totalAccountsCreated** типа **uint** объявлена как **static**. При запуске программы она инициализируется нулем, что позволяет обращаться к ней прежде, чем созданы какие-либо объекты класса **Account**. Это демонстрируется командой **P**<enter> в начале вывода примера: пользователь узнает, что на данный момент открыто и существует ноль счетов. Команда **P** посредством **switch**-раздела в строках **78—80** вызывает метод **PrintStatistics** (строки **47—51**) объекта **bigBucksBank** класса **Bank**.

Обращение к переменной **totalAccountsCreated** производится в строке **50** с помощью операции уточнения: **Account.totalAccountsCreated**.

Значение переменной **totalAccountsCreated** можно прочесть и изменить из объекта **Account** так же, как значение любой стандартной переменной экземпляра. Сказанное демонстрируется конструктором **Account** (строки **10**-**13**). Напомним, что конструктор вызывается при создании каждого нового объекта. В данном случае он увеличивает значение переменной **totalAccountsCreated** на единицу (строка **12**).

Стандартный массив **C#**, представленный в листинге 5.2, после создания имеет фиксированную, неизменяемую длину. Здесь же требуется конструкция, подобная массиву (содержащая коллекцию объектов), но способная при необходимости увеличиваться или уменьшаться. Такой функциональностью обладает класс **Arraylist**, содержащийся в пространстве имен **System.Collections.NET Framework.** Его полное имя **System.Collections.ArrayList**, но благодаря декларации в строке **2** (**using System.Collections;**), можно применять сокращение **ArrayList** (в строках **18** и **23**). Подобно классам **string** и **Array**, **ArrayList** содержит большой набор встроенных методов. Однако в данной программе используются два собственных

метода: **Add** (строка **28**) и **RemoveAt** (строка **37**), а также свойство **Count** (строки **35** и **49**). Когда в конструкторе объекта **Bank** (строка **23**) создается экземпляр **accounts** (объявленный в строке **18**), его длина равна **0**.

Объект **ArrayList** может хранить любые объекты, поэтому его базовый тип (как в случае массива) определять не нужно. Для добавления в список нового объекта **Account** применяется метод **Add**, аргументом которого является ссылка на объект (см. строку **28**). Данный метод добавляет объект в конец существующего списка (аналогично тому, как клиенты становятся в очередь). Метод **RemoveAt** удаляет объект, индекс которого задан в аргументе (строка **37**).

Свойство **Count** содержит текущее число объектов в списке **ArrayList**.

Между **Count** и **totalAccountsCreated** существует смысловое различие**:** **totalAccountsCreated** подсчитывает, **сколько раз в программе создавался новый счет**, а **Count** следит за текущим количеством объектов в списке **accounts**. Таким образом, значение **Count** увеличивается, когда объекты добавляются, и уменьшается при их удалении, а **totalAccountsCreated** увеличивается в первом случае, *но не уменьшается во втором*.

Контрольные вопросы

1. В каждом из трех модулей компиляции находится следующее определение пространств имен:

Модуль компиляции 1:

```
namespace MyCompany
{
   public class Bicycle
   {
   …
   }
}
```

Модуль компиляции 2:

```
namespace MyCompany.Design
{
   public class Drawer
   {
   …
   }
}
```

Модуль компиляции 3:

```
namespace MyCompany.Design.Tools
{
   public class Cutter
   {
   …
   }
}
```

Все модули необходимо собрать в одну сборку. Вместо трех модулей компиляции нужно создать один с такой же структурой пространств имен, как и объединяемые. Для решения задачи используйте следующее определение вложенных пространств имен в **C#**:

```
namespace <Внешнее_пространство_имен>
{
      namespace <Внутреннее_пространство_имен>
      {
            и т.д.
```

2. Вы создали два исходных файла **Bicycle.cs** и **Person.cs**. Их нужно скомпилировать в **DLL**-сборку **healthlib.dll.** При компиляции необходимы ссылки на две сборки — **mathlib.dll** и **anatomy.dll**. Напишите команду компилятора, выполняющую эти дей ствия.

3. Для чего применяется утилита **ildasm**?

Задание по самостоятельной работе #1

Напишите базовый код программы автомобильной игры. Программа должна включать класс **Car** со следующими элементами:

1. переменная экземпляра **position** типа **int**
2. метод с заголовком **public void MoveForward(int distance),** который добавляет расстояние **distance** к переменной экземпляра **position**
3. метод с заголовком **public void Reverse(int distance),** который вычитает расстояние **distance** из позиции **position**
4. метод **GetPosition,** возвращающий величину **position** в точку вызова

Кроме того, программа должна содержать класс **CarGame,** который (используя массив) содержит 5 объектов типа **Car.** Этот класс должен позволять перемещать каждый автомобиль (вперед и назад) и возвращать положение каждого из автомобилей (заданного посредством индекса массива).

Программа также должна содержать класс **CarGameTester,** содержащий метод **Main.**

Напишите небольшую тестовую программу (она включает классы: **Car, CarGame** и **CarGameTester**) и убедитесь, что оба класса (**Car, CarGame**) функционируют правильно.

Задание по самостоятельной работе #2

1. Выполните упражнение (аналогично упр-нию #3 к Практ-му занятию №4, стр. 42) для исходного файла **BankSimulation.cs** из листинга 5.2. Создайте по одному модулю компиляции для каждого класса **Account** и **Bank** и разместите их в подходящей иерархической структуре пространств имен. Соберите эти части в одну **DLL**-сборку. Создайте исходный файл для программы моделирования банка, использующий пространства имен и классы **DLL**-сборки. Функциональные возможности данной программы должны быть идентичны возможностям оригинальной (из листинга 5.2).

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.

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

Что значит освоить объектно-ориентированное программирование? Для этого недостаточно выучить синтаксис языка C#, Java или C++. Нужно разобраться в принципиальных положениях объектного подхода, понять, чем он отличается от других. И предлагаемая книга будет в этом отличным помощником. В ней на конкретных примерах разбираются все основные понятия объектно-ориентированного подхода. Советую прочесть эту книгу [**6**].

C# & .NET по шагам (Web-ресурс)

1 | 2 | 3 | 4

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

1 | 2 | 3 | 4