Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

C#

Объектно-ориентированный язык программирования

Пособие к практическим занятиям - №4

Проф. Забудский Е.И.

Москва 2005

Тема 4. Объектно-ориентированный подход

к разработке программного обеспечения.

Типы C# . Форматирование числовых значений. Метаданные.

Компонентно-ориентированное программирование

Три практических занятия
(6 часов)

Рассмотрены алгоритм и cs-программа (листинги 4.1 и 4.2), которые могут быть частью таймера *объектно-ориентированной* программы, моделирующей работу лифта (это продолжение Практического занятия - №3).

Сопоставляются два подхода используемые для классификации встроенных типов в C# (листинг 4.3). C# - строго типизированный язык. Форматирование простых типов.

Метаданные (“данные о данных ”) используются для описания и ссылки на все типы, определенные системой VOS (Virtual Object System). Доступ к метаданным посредством метода GetType (листинг 4.4).

За компонентно-ориентированным программированием – будущее (листинги 4.5 - 4.10)

// **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. Алгоритм вычисления дня недели для модели лифта

1.1. Характеристики программного обеспечения

Алгоритм, описанный здесь, может быть частью таймера программы, моделирующей работу лифта. Она моделирует проходящие дни и оценивает, сколько дней длится сеанс. Программа также определяет, является ли день рабочим или выходным. Она позволяет пользователю определить общее число "дней" в одном сеансе, в течение которых работает система "лифт".

Итак, алгоритм должен подсчитывать число "дней" работы моделируемой системы. Кроме того, требуется выводить на экран, сколько "недель" и "дней" длится сеанс. Например, если система функционирует в течение 23 дней, алгоритм должен выдавать значение 3 недели и 2 дня.

В заключение, алгоритм должен выдавать сигнал системе управления, если текущий день является воскресеньем. Это важно, так как в выходной день система (в целях эффективности) переходит в режим "низкой загруженности".

1.2. Разработка программного обеспечения

Здесь можно перейти непосредственно к фазе разработки, поскольку спецификации требуют небольшого изолированного блока ПО (алгоритм), делая излишними этапы определения классов и методов.

1.3. Разработка алгоритма внутреннего метода

Какие переменные должен содержать алгоритм? В первую очередь, ему требуется знать общее время работы системы (в днях). Соответствующая переменная называется **maxSimulationDays** и принадлежит беззнаковому целочисленному типу **uint**, так как в ней хранятся лишь целые положительные числа. Моделирующая программа должна следить за текущим числом дней работы системы, — для этого предназначена переменная dayCounter, также определенная как **uint**.

Далее требуется цикл. При каждой его итерации **dayCounter** увеличивается на единицу, выводится число недель и дней, а в случае, если текущий день является воскресеньем, — выдается предупреждение.

Цикл завершается, когда **dayCounter** достигает значения **maxSimulationDays**

Как же преобразовать значение **dayCounter** в число дней и недель? Число недель получается в результате деления (**dayCounter / 7**), которое является целым, так как операция производится над целыми числами. Чтобы найти число оставшихся дней, применяется оператор деления по модулю (**dayCounter % 7**).

Операция деления по модулю, обозначается символом процента <%> и применяется при решении многих задач. Она используется только с целыми числами и возвращает остаток от деления первого (левого) операнда над вторым (правый).

Например, 20 % 7 равно 6 .

Теперь необходимо определить, является ли текущий день воскресеньем. **Воскресенье** — это каждый **7**-й день недели. А как можно найти каждый 7-й день? Очевидно, что, если (**dayCounter % 7**) равно нулю, — это и есть воскресенье.

Псевдокод, реализующий данный алгоритм, приведен на листинге 4.1. Он выполняет все действия, описанные выше.

1. 4. Листинг 4.1. Псевдокод алгоритма вычисления дня недели

01: Записать в переменную **dayCounter** значение **0**

02: Записать в **maxSimulationDays** общее число дней, в течение которых система должна функционировать

03: Пока **dayCounter** меньше, чем **maxSimulationDays**, повторять следующий блок

04: {

05: Увеличить **dayCounter** на единицу

06: Вычислить и вывести число недель в **dayCounter**, а также оставшееся число дней.

07: Если **dayCounter** делится на **7**, послать сообщение ("**Воскреснье**") диспетчеру и вывести: "**Внимание. Воскресенье!**"

08: Если этот фрагмент кода используется как часть большей моделирующей программы, запустить сеанс моделирования, длящийся один день

09: }

10: Сообщить пользователю, что Сеанс моделирования окончен

Алгоритм содержит цикл (строки 3-9), который повторяет выражения в строках 5-8 до тех пор, пока **dayCounter** меньше, чем **maxSimulationDays**.

Теперь на основе псевдокода будет создаваться код C#. Алгоритм содержится в методе **Main()**. В таблице 4.1 показано соответствие строк псевдокода строкам программы, приведенной на листинге 4.2.

Таблица 4.1

Строка псевдокода	Эквивалентная строка в программе C#
01	16
02	21-23
03	24
04	25
05	26
06	27-30
07	31-33
08	34
09	37
10	38

1.5. Листинг 4.2. Исходный код DayCounter.cs

```
01: using System;
02:
03:/*
04: * Класс DayCounter содержит прототип
05: * алгоритма "Day counter", используемый в
06: * программе имитации лифта.
07: * Он считает количество дней,
08: * выводит общее время работы в неделях и днях
09: * и предупреждает, когда текущим днем является воскресенье
10:  */
11:
```

```
12: class DayCounter
13: {
14:     public static void Main()
15:     {
16:      uint dayCounter = 0;
17:      uint maxSimulationDays;
18:      uint weeks;
19:      byte remainderDays;
20:                                          .
21:     Console.Write("Пожалуйста, введите количество дней " +
22:        "работы моделируемой системы ");
23:       maxSimulationDays = Convert.ToUInt32(Console.ReadLine());
24:     while(dayCounter < maxSimulationDays)
25:      {
26:         dayCounter++;
27:         weeks = dayCounter / 7; // сохраняется целое
28:        remainderDays = (byte)(dayCounter % 7); // сохраняется остаток
29:         Console.WriteLine("Недели: " + weeks +
30:             "  Дни: " + remainderDays);
31:        if( remainderDays == 0 ) // если остаток равен нулю –> день недели воскресенье
32:       .      // ToDo: послать сообщение "Воскресенье" диспетчеру
33:              Console.WriteLine("\t\tВнимание. Воскресенье!");
34:         // ToDo: начать сеанс моделирования длительностью в один день.
35:         // Пауза в программе на 200 миллисекунд
36:        System.Threading.Thread.Sleep(200);
37:      }
38:     Console.WriteLine("Моделирование закончено");
39:     }
40:}
```

Ниже приведен вывод программы, запущенной для моделирования **15** дней работы лифта.

```
Пожалуйста, введите количество дней работы моделируемой системы 15<enter>
Недели: 0   Дни: 1
Недели: 0   Дни: 2
Недели: 0   Дни: 3
Недели: 0   Дни: 4
Недели: 0   Дни: 5
Недели: 0   Дни: 6
Недели: 1   Дни: 0
                Внимание. Воскресенье!
Недели: 1   Дни: 1
Недели: 1   Дни: 2
Недели: 1   Дни: 3
Недели: 1   Дни: 4
Недели: 1   Дни: 5
Недели: 1   Дни: 6
Недели: 2   Дни: 0
```

```
            Внимание. Воскресенье!
Недели: 2   Дни: 1
            Моделирование закончено
```

В строках 24-37 содержится цикл **while**. Его содержимое ограничено фигурными скобками в строках 25 и 37 и выполняется, пока условие (**dayCounter < maxSimulationDays**) имеет значение **true**. Если это условие в строке 24 дает **false**, управление переходит к строке 38.

В строке 26 значение **DayCounter** увеличивается на единицу.

В строке 27 вычисляется количество полных недель в **DayCounter**. Поскольку все переменные в этих вычислениях принадлежат типу **uint**, результатом является целое число.

В строке 28 с помощью операции деления по модулю (%) вычисляется оставшееся число дней. **RemainderDays** принадлежит типу **byte**, а выражение (**DayCounter % 7**) — типу uint, поэтому требуется явное приведение к типу операцией (**byte**).

Поскольку в строке 28 **remainderDays** равно (**DayCounter % 7**), необходимо проверить, является ли это значение нулем. Это выполняет оператор **if** в строке 31.

Программисты часто используют **ToDo** как напоминание о том, что осталось доработать в программе. Комментарии, обозначенные **ToDo**, легко найти в программе средствами поиска текстового редактора. В строке **32** просто указано, что, если программа является частью реальной моделирующей программы, в этой точке необходимо послать сообщение "**Воскресенье**" диспетчеру.

Идентификатор **ToDo** настолько признан среди программистов, что такие среды разработки, как **Visual Studio .NET**, распознают комментарии **ToDo** и вставляют их в нужных местах.

В строке 33 для выравнивания текста "**Внимание. Воскресенье!**" используется символ TAB (**\t**).

Строку 36 на данный момент можно рассматривать просто как своеобразный способ останова программы на 200 миллисекунд. Пауза заставляет программу выводить информацию медленнее, в более удобной манере. Можно поэкспериментировать с этим значением для ускорения (уменьшая число) или замедления (увеличивая) программы.

//ПРИМЕЧАНИЕ

Приведенная реализация (как и большинство алгоритмов) может быть разработана и по-другому, без операции деления по модулю. Однако представленный здесь алгоритм более эффективен.

2. C# — строго типизированный язык

Каждая величина в C# должна иметь тип. Из-за разнообразия имеющихся в C# типов часто приходится *объединять величины различных типов* в одних и тех же выражениях исходной программы. Чтобы помочь программисту правильно применить этот диапазон типов, ускорить разработку и увеличить устойчивость конечного приложения к ошибкам, в C# имеются четко определенные правила совместимости между различными типами. Компилятор действует как полиция — он следит за исполнением этих правил, и при их нарушении сообщает об ошибках и дает предупреждения.

Процедура, выполняемая компилятором C# для проверки строгого исполнения правил совместимости в программе, называется проверкой соответствия типов.

Типичные элементы кода, при наличии которых компилятор C# делает проверку соответствия типов:

1.Операция присваивания.

2. Обращение метода к другому методу с передачей ему аргументов.

3. Выражение.

➢ **Операция присваивания**. Всякий раз, когда переменной присваивается какое-либо значение в операции присваивания, компилятор проверяет совместимость между типом переменной и типом этого значения. При несоответствии выводится сообщение об ошибке.

Нарушение правил совместимости типов называется коллизией типов.

➢ **Передача аргументов при вызове метода**. Передача аргументов формальным параметрам при обращении к методу — другой случай потенциальных коллизий типов, и поэтому тщательно проверяется компилятором.

➢ **Выражение**. Любое выражение в C# имеет конкретный тип. Когда выражения вложены друг в друга, компилятор выбирает тип, подходящий для выражений более высоких уровней, и, естественно, проверяет совместимость.

Неявные преобразования — это преобразования одного типа в другой, которые ***автоматически*** выполняются компилятором C#.

Для выполнения неявного преобразования необходимо, чтобы оно не приводило к потере каких-либо данных и не требовало дальнейшего детального исследования участвующих в этом преобразовании конкретных значений.

Язык программирования, запрещающий использование значения в каком-либо выражении (если значения и выражения несовместимы) называется языком со строгой проверкой типов, или строго типизированным языком.

C# — строго типизированный язык.

Все выражения и операции в C# проверяются, где это важно, на совместимость.

2.1. Типы в C#

Типы C# можно разделить на простые (см. Приложение 2) и производные. Другой важный способ классификации: тип-значение и ссылочный тип.

2.2. Типы-значения и ссылочные типы

Разбивать типы на **простые** и **производные**, строго говоря, неверно. Для представления простых типов в C# используется тип-значение, подобный классу и *называемый* struct. Это позволяет применять к каждому простому типу предопределенные для него методы. Поэтому для числа **5** можно вызвать метод ToString, используя синтаксис

```
5.ToString();
```

метод, определенный для значения **5,**

операция уточнения

который возвращает строку "**5**". Таким образом, простой тип является таким же сложным, как и любой из производных типов.

// ПРИМЕЧАНИЕ

Название **ToString** — часто используемое имя для методов в **.NET Framework**. Такой метод, как следует из его названия, преобразует заданное значение в тип **string**. Описание метода **ToString**, использованного в примере для числа **5**, можно найти в документации по **.NET Framework** в разделе методов структуры **System.Int132**.

Тем не менее, различение простых и производных типов достаточно удобно. Оно берет начало от предыдущих языков, подобных C++ и Java, в которых простые типы являются в прямом смысле простыми (в отличие от языка C#): к примеру, число представляет собой просто число без каких-либо дополнительных функциональных возможностей. В дальнейшем будет приведено более подробное обсуждение механизма работы структуры **struct**, и станет ясно, почему C# можно рассматривать как полностью объектно-ориентированный язык программирования, в котором **все типы**, включая и простые, **являются объектами**.

Переменная типа-значения **содержит значение, хранимое непосредственно в ней** (т.е. в соответствующих ячейках памяти компьютера). Примером может служить тип **int**.

Все это довольно просто, но важно при рассмотрении другого набора — ссылочных типов.

Переменная ссылочного типа **содержит в памяти ссылку на объект**, а не сам объект непосредственно. Ссылка представляет собой позицию (адрес) объекта в памяти. Для иллюстрации того, что же представляет собой ссылочный тип, обратимся к уже изученному типу **string** (который, как выяснится дальше, является ссылочным).

*Переменная типа **string** не содержит строку, а* объявляется для хранения ссылки на строку. Сама строка размещается по определенному адресу в памяти.

Рассмотрим следующий фрагмент кода:

string myText; ← объявление переменной myText типа string .

Этот оператор объявления можно передать словами так: "*Пусть myText содержит ссылку на строку*".

После этого **myText** можно применить в следующем операторе присваивания:

myText = “Рассмотрим C, чтобы понять C#";

в результате чего переменной myText присваивается адрес следующей строки : “Рассмотрим C, чтобы понять C#".

Однако в ячейках памяти, где содержится переменная **myText**, нет никакого текста, а только адрес памяти, называемый также **ссылкой** или **указателем**. Сказанное иллюстрируется рис. 4.1, на котором текст размещен по произвольному адресу **4027**, содержащемуся в переменной **myText**.

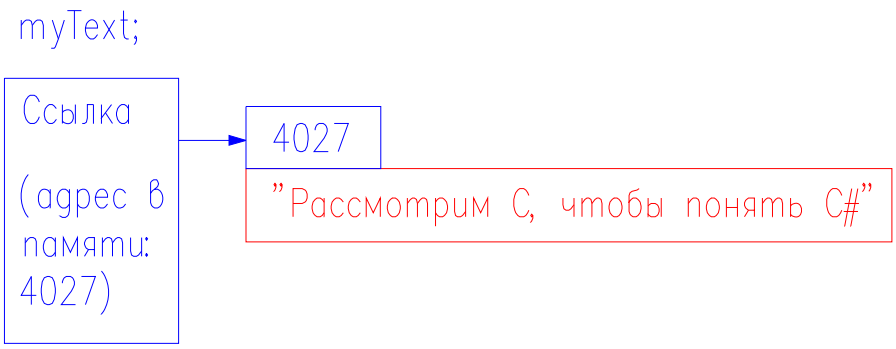


Рис. 4. 1. Тип string является ссылочным

//ПРИМЕЧАНИЕ
Важно отметить, что фактический адрес при использовании ссылок в исходном коде программы никогда не применяется.

Класс string — хороший пример ссылочного типа. Другими примерами могут служить классы **Elevator** или **Person** из листинга 4.1 (см. практическое занятие №3, стр. 13-15).

// ПРИМЕЧАНИЕ
Все классы являются ссылочными типами.

Адрес места в компьютерной памяти, где хранится объект, называется ссылкой или указателем на этот объект.

В большинстве случаев, различия в работе с типами-значениями и ссылочными типами незначительны. Пример этого — возможность применения строк в предыдущих главах без использования понятия ссылки. Иногда, все же, в поведении двух этих видов типов имеются принципиальные различия, которые должен знать каждый объектно-ориентированный программист. Исходный код в листинге 4.3 иллюстрирует подобную ситуацию.

2.3. Листинг 4.3. Исходный код ReferenceTest.cs .

```
01: using System;
02:
03: /*
04:  * Этот класс является примером
05:  * различий между ссылочными типами
06:   * и типами-значениями
07:  */                                          ,
08:
09: class Person
10: {
11:     private int age = 0;
12:
13:     public void SetAge(int newAge)
14:     {
15:     age =   newAge;
16:     }
17:
18:     public int GetAge()
19:     {
20:     return age;
21:     }
22: }
```

```
23:
24: class ReferenceTester
25: {
26:     public static void Main()
27:     {
28:      Person julian;
29:      Person deborah;
30:      // переменные julian и deborah – имеют ссылочный тип –> класс Person
31:      julian = new Person();       // julian - экземпляр класса Person, т.е. объект этого класса
32:      deborah = new Person();   // deborah - экземпляр класса Person, т.е. объект этого класса
33:      julian.SetAge(2);
34:      deborah.SetAge(33);
35:      Console.WriteLine("Julian's age: " + julian.GetAge());          // age = 2
36:      Console.WriteLine("Deborah's age: " + deborah.GetAge());  // age = 33
37:      julian = deborah; // julian и deborah ссылаются на одну и ту же ячейку памяти
38:      Console.WriteLine("Julian's age: " + julian.GetAge());          //  age = 33
39:      Console.WriteLine("Deborah's age: " + deborah.GetAge());  //  age = 33
40:      julian.SetAge(10);
41:      Console.WriteLine("Julian's age: " + julian.GetAge());          //   age = 10
42:      Console.WriteLine("Deborah's age: " + deborah.GetAge());  //   age = 10
43:     }
44:}
```

Результаты работы программы :

```
Julian's age: 2
Deborah's age: 33
Julian's age: 33
Deborah's age: 33
Julian's age: 10
Deborah's age: 10
```

В строке **9** определен класс по имени **Person**, который содержит только одну переменную экземпляра **age**, представляющую возраст (строка 11).

В строках **13-21** два метода по имени **SetAge** и **GetAge** осуществляют доступ к значению **new**.

Класс **ReferenceTester** содержит метод **Main**, в котором объявлены две переменных (**julian** и **deborah** в строках **28** и **29**), содержащие ссылки на объекты класса **Person**.

В строках **31** и **32** создаются экземпляры класса **Person** (посредством ключевого слова **new**). Первый важный момент: в строке **31** ссылка, указывающая на вновь созданный объект **Person**, присваивается переменной **julian**. В строке **32** то же делается с переменной **deborah**. Очевидно, что новый объект Person, созданный в строке **32**, отличается от объекта **Person** из строки **31**.

В строках **33** и **34** переменным экземпляра **age** двух объектов присвоены значения **2** и **33** соответственно. На рис. 4.2 показаны ссылки из **julian** и **deborah** после выполнения строки **34**.

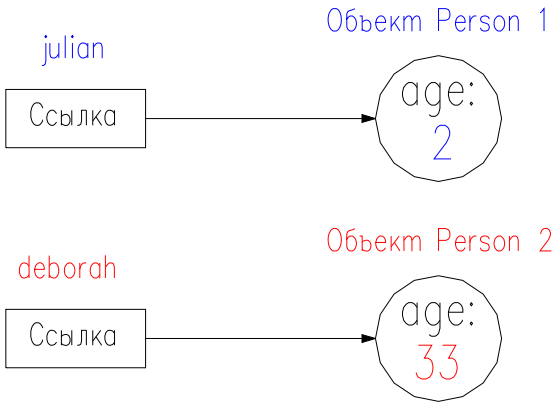


Рис. 4.2. Соотношение ссылок после выполнения строки **34** листинга **4.3**

В строках **35** и **36** возраст **julian** и **deborah** выводится на экран.

В строке **37** программа присваивает значение **deborah** переменной **julian**. Однако, поскольку оно принадлежит к ссылочному типу, это действие просто указывает переменной **julian** *ссылаться на тот же объект*, что и **deborah**. Состояние ссылок показано на рис. 4.3.

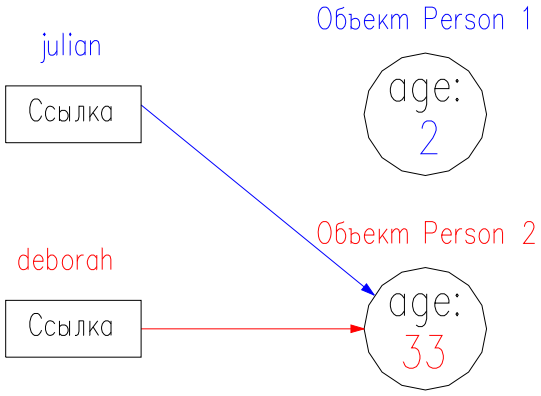


Рис. 4.3. Соотношение ссылок после выполнения строки **37** листинга **4.3**

Таким образом, теперь **julian** и **deborah** ссылаются на один и тот же объект. Это подтверждается выводом в строках **38** и **39**, где и **julian**, и **deborah** имеют один и тот же возраст —. **33**, а также действиями в строках **40-42**. В строке **40** программа заменяет возраст **julian** на **10**, но, как видно из вывода, возраст **deborah** также становится равным **10**.

Понимание различий между типами-значениями и ссылочньми типами позволяет не только быстро и точно находить ошибки в исходном коде, но и создавать более сложные программы.

2.4. Основные типы C#: обзор

Познакомившись с типами-значениями и ссылочными типами, полезно рассмотреть иерархическую структуру основных типов C#, приведенную в таблице 4.2. Поскольку некоторые из них обсуждались выше, здесь они рассматриваются вкратце.

Таблица 4.2. Обзор основных типов C#

Типы-значения	**Ссылочные типы**
Простые типы (например, **int**)	Типы-классы (например, **Elevator** и **Person** из листинга 4.1) Тип **string**
Типы-перечисления*	Типы-массивы*
Типы-структуры*	Типы-интерфейсы* Типы-делегаты*

* еще не обсуждались

Ниже приведено краткое пояснение типов C#:

- **Простые типы**. Знакомый тип **int** является одним из 13 простых типов C# (см. Приложение 2). Простые типы используются для хранения числовых значений и отдельных символов.
- **Типы-перечисления**. Эти типы обеспечивают средства для создания символических констант, в частности, их наборов: дни недели (понедельник, вторник и т.д.), месяцы (январь, февраль, март и т.д.), цвета (зеленый, красный, синий) и многих других.
- **Типы-структуры**. Эти типы содержат методы и данные так же, как и классы. Будучи подобными классам, они имеют и несколько отличий. Одно из наиболее важных состоит в том, что типы-классы являются ссылочными, в то время как структуры — это типы-значения. Фактически, все простые типы — типы-структуры.
- **Типы-классы**. Класс определяет категорию объектов и является "чертежом" для их создания. Классы содержат элементы, которые могут быть: **1)** переменными экземпляра, описывающими состояние объекта, и **2)** методами, определяющими его поведение. Классы могут наследовать элементы других классов. Понятие класса является центральным в ООП.
- **Типы-массивы**. Переменные типа массив — это объекты, предназначенные для хранения коллекций данных. Все элементы массива принадлежат одному и тому же типу.
- **Типы-интерфейсы**. Интерфейс предусматривает абстрактное поведение, определяя один или несколько заголовков метода без сопровождающей их внутренней реализации, которая присутствует в неабстрактных методах класса. Классы могут реализовывать интерфейсы, конкретизируя абстрактное поведение, установленное интерфейсом. Интерфейсы позволяют программисту реализовать наиболее сложные концепции ООП.
- **Типы-делегаты**. Подобно интерфейсам, делегаты используются для определения поведения, но задают заголовок только для одного метода. Экземпляр типа делегат содержит один метод, а сам делегат является ссылкой на него. Делегаты (или ссылки на методы) передаются в программе как обычные ссылки и выполняются как обычные методы. Делегаты жизненно важны для выполнения управляемых событиями программ на языке C#.

// **ЕДИНАЯ СИСТЕМА ТИПОВ** .NET (**COMMOM TYPE SYSTEM** (CTS))

Единая система типов (**CTS**) является составной частью **.NET Framework**. Она определяет все типы, описанные в этом разделе, и содержит правила их применения в приложениях, выполняемых в среде **.NET**. Как следует из названия, все реализуемые на платформе **.NET** языки программирования, включая C#, основаны на типах, определенных **CTS**.

Резюме к разделу 2 (типы в C#)

В ООП *типы используются* как строительные блоки для формирования *других типов*.

Для классификации встроенных типов в C# можно использовать два подхода: **1)** простые типы — производные типы и **2)** типы-значения — ссылочные типы. *Первый* подход более *удобен*, а второй — более корректен.

Все значения, относящиеся к одному типу, имеют общий набор предопределенных характеристик.

Каждая величина должна иметь определенный тип. В C# есть строгие правила для операций, в которых могут принимать участие величины определенных типов. По этой причине C# называют строго типизированным языком.

Строго типизированный язык позволяет использовать мощные операторы объявления.

Переменная типа значения содержит данные, фактически хранящиеся в ней. Ссылочный тип содержит ссылку на блок памяти, где хранятся данные. **int** — ***тип-значение***, **string** — ***ссылочный тип***.

Для ссылки на один и тот же объект можно использовать несколько переменных одного ссылочного типа.

Важными предопределенными типами-значениями в C# являются простые типы и типы-перечисления. Все типы-значения являются структурами.

Важные предопределенные *ссылочные типы*: строки, массивы, интерфейсы и делегаты. Все *ссылочные типы являются классами*.

В C# можно определять собственные классы и структуры.

В языке C# имеется **тринадцать** предопределенных простых типов, используемых для представления большинства величин. Девять из них — целочисленные, два — с плавающей точкой, один — высокой точности и один — логический.

Между разными простыми числовыми типами существует путь неявного преобразования. Неявные преобразования выполняются компилятором **автоматически**. *Явные преобразования*, заданные в исходном коде, необходимы для преобразований по этому пути в обратном направлении.

Целочисленная переменная переполняется сверху, когда ее значение становится большим, чем определенный данным типом верхний предел. Ее значение устанавливается равным нижнему пределу типа. Переполнение снизу — тот же процесс, идущий в обратном направлении.

Работа с простыми типами связана с несколькими тонкими проблемами, о которых следует знать для предотвращения ошибок в исходном коде.

Значение константы определяется в исходном коде и не может быть изменено после компиляции. В тексте программы постоянную величину представляет ее идентификатор.

Язык C# позволяет форматировать числовые величины, делая вывод более понятным и компактным.

Величины типа **bool** могут иметь одно из двух значений: **true** и **false**.

3. Форматирование числовых значений

До сих пор числа здесь выводились с использованием простого встроенного формата, например,

```
Console.WriteLine( "Пройдено расстояние:" + 10000000.432);
```

При этом на консоль отображается следующее:

```
Пройдено расстояние: 10000000.432
```

Однако, изменяя внешний вид числа с помощью запятых (в зарубежной нотации принято отделять запятой разряды, кратные тысячам), и ограниченного количества десятичных цифр, можно улучшить его читаемость и сделать более компактным при выводе на экран. В таблице 4.3 представлено несколько примеров.

Таблица 4.3. Примеры форматированных чисел

Имя переменной	***Число***	*Отформатированное число* (*более удобное для восприятия*)
profit - доход	3000000000.44876	$3,000,000,000.45
distance - расстояние	7000000000000000	7.00E+015
mass - масса	3.8783902983789877362	3.8784
length - длина	20000000	20,000,000

Рассмотрим встроенные возможности C# для форматирования чисел при преобразовании в строки.

3.1. Стандартное форматирование

Напомним, что каждый числовой тип в **.NET Framework** представлен структурой **struct**, что позволяет ему содержать полезную встроенную функциональность. Одним из ее примеров является метод **ToString**. Он позволяет преобразовать любой из простых типов в строку и, кроме того, указать требуемый формат.

Метод **ToString** (в используемой здесь форме) имеет один аргумент типа **string**, задающий формат. Аргумент состоит из символа, называемого символом формата (в данном случае **N**), который задает формат, и необязательного числа, — спецификатора точности, — которое имеет различный смысл для разных символов формата. Используемые символы и соответствующие им форматы приведены в таблице 4.4.

Таблица 4.4. Используемые символы формата

Символ	Описание	Пример
C, c	***Валюта***. Форматирование, специфичное для настроек локализации. Они содержат информацию о типе денежной единицы и других параметрах, которые могут изменяться в зависимости от страны.	2000000.456m.ToString("C") Возвращает "$2,000,000.46" (Если операционная система настроена в соответствии с американскими стандартами.)
D, d	***Целое число***. Спецификатор точности устанавливает минимальное число цифр. Вывод дополняется ведущими нулями, если количество цифр фактического числа меньше, чем спецификатор точности. (Примечание: этот символ формата применяется только для целочисленных типов.)	45687.ToString ("D8") Возвращает: "00045678"
E, e	***Экспоненциальная (научная) нотация.*** Спецификатор точности определяет количество десятичных цифр, по умолчанию равное 6.	345678900000.ToString("E3") Возвращает "3.457E+011" 345678912000.ToString("e") Возвращает "3.456789e+011"
F, f	***Фиксированная точка***. Спецификатор точности показывает количество десятичных цифр.	3.7667892.ToString("F3") Возвращает "3.767"
G, g	***Общий***. Наиболее компактный формат при выборе **E** или **F**. Спецификатор точности устанавливает максимальное количество цифр в представлении числа.	65432.98765.ToString("G") Возвращает "65432.98765" 65432.98765.ToString("G7") Возвращает "65432.99" 65432.98765.ToString("G4") Возвращает "6.543E4"
N, n	***Число***. Число с запятыми-разделителями. Спецификатор точности устанавливает количество десятичных цифр.	1000000.123m.ToString("N2") Возвращает "1,000,000.12"
X, x	***Шестнадцатеричное число***. Спецификатор точности устанавливает минимальное количество цифр, представляемых в строке. Для достижения определенной ширины добавляются ведущие нули.	950.ToString("x") Возвращает "3b6" 950.ToString("X6") Возвращает "0003B6"

Метод **ToString** представляет собой мощное средство для форматирования числовых величин.

Существует еще одна возможность реализации вывода. Если использовать спецификатор {<**N**>}, где <**N**> задает позицию числа в списке чисел, следующих после статического текста в вызове **WriteLine**, строки кода можно записать в таком виде:

```
Console.WriteLine("Длина: {0}  Ширина: {1}  Высота: {2}",
10000000.4324, 65476356278.098746, 4532554432.45684);
```

где {0} относится к первой величине (100000000.4324) в списке чисел после строки текста, {1} — ко второй (65476356278.098746), а {2} — к третьей. Такая запись делает выражение более ясным.

На экран выводится следующее:

```
Длина: 100000000.4324  Ширина: 65476356278.098746  Высота: 4532554432.45684
```

4. Доступ к метаданным компонента: краткое введение

Выражения формируются комбинированием различных типов операндов (констант, переменных экземпляра, вызовов методов и т.д.). Тип каждого операнда необходимо тщательно проверить, чтобы результат конечного выражения имел смысл. Нахождение типа операнда в предыдущих примерах было довольно простым делом. В классах, созданных в небольших программах на C#, достаточно посмотреть на заголовок метода или объявление переменной в соответствующем классе.

Но как быть, если исходный код является **компонентом** (***.dll**), а возможность определения атрибутов типа все так же важна? **Компонент** может передавать данные базе данных или, обмениваться ими с другими **компонентами**, написанными **на других языках**, имеющих другой синтаксис и конфигурацию, или даже быть частью **Web**-службы.

Если **компонент** описывает типы в гибкой, дружественной пользователю манере и исключает необходимость для других контекстов расшифровывать его внутренние стандарты, он сможет легко интегрироваться в различные среды. Встроенная в C# поддержка метаданных позволяет разделить детали реализации и контекста.

В материале к Практическим занятиям №1 (стр. 27, 28, рис. 1.8) кратко охарактеризована концепция метаданных (см. также рис. 4.4 на стр 24). **Метаданные** содержат: **1)** подробное описание методов, **2)** переменных экземпляров и **3)** многих других существенных характеристик каждого типа в программе. Одним из атрибутов, в частности, является просто имя типа, как, например, **System.Int32** или **System.Decimal**.

Процесс доступа к метаданным в течение выполнения **называется отражением**. Здесь рассматривается пример того, как им можно воспользоваться для получения имени типа. Это также знакомит с тем, как осуществляется доступ к метаданным.

Наша цель — определить с помощью отражения тип нескольких выражений в реальной программе.

Обратимся к методу **ToString**. Это один из нескольких полезных методов, содержащихся в простых типах данных. В руководстве по **.NET Framework** можно найти еще один **ценный метод — GetType**.

При вызове **GetType** для выражения, этот метод возвращает объект **класса Type**, предоставляющий доступ ко всем метаданным о его типе.

//ПРИМЕЧАНИЕ

Метод GetType поддерживается для любого значения в C#, принадлежит ли оно к предопределенному простому типу, типу-значению или объекту ссылочного типа.

Класс Type является частью библиотеки классов **.NET Framework** и содержится в пространстве имен **System**. Объект класса Type оснащен методами и свойствами (часть из которых приведена в таблице 4.5), предназначенными для доступа к метаданным.

Таблица 4.5. Три примера информации, доступной посредством объекта класса **Type**

Свойство	Краткое объяснение
FullName	Возвращает **полное** имя типа, включая и пространство имен
IsPrimitive	Возвращает **true**, если тип является **простым**, иначе — **false**
IsClass	Возвращает **true**, если тип является **классом**, иначе — **false**

//ПРИМЕЧАНИЕ

FullName, IsPrimitive и **IsClass** являются **не методами, а свойствами**. Свойство является важной конструкцией в C#. Она предоставляет доступ к переменным объекта снаружи, без нарушений правил инкапсуляции. Обсуждение свойств будет выполнено позже. Свойства во многом подобны методам. Визуальная разница заключается в том, что *при обращении к свойствам не требуются скобки*.

Имена свойств, согласно стилистике Microsoft, необходимо задавать в стиле Pascal.

Листинг 4.4 иллюстрирует как три свойства, приведенные в таблице 4.5, применяются для получения информации о простых типах.

4.1. Листинг 4.4. Исходный код MetadataAccessor.cs

```
01: using System;
02: /*
03: * Данный класс демонстрирует доступ к
04: * метаданным определенного типа.
05: */
06: class MetadataAccessor
07: {
08:     public static void Main()
09:     {
10:      Type anyType;
11:      byte age = (byte)37;          // явное преобразование типа, см. Приложение 2, стр. 45
12:      short energy = (short)4000;   //   -,-
13:      ushort height = (ushort)190;  //   -,-
14:      decimal mass = 398.98765m;
15:
16:      anyType = age.GetType();
17:      Console.WriteLine("Тип переменной age:"
18:      + anyType.FullName);
19:      if(anyType.IsPrimitive)
20:      Console.WriteLine("Переменная age имеет простой тип ");
21:      if(anyType.IsClass  == false)
22:      Console.WriteLine("Переменная age не имеет тип класс ");
23:      anyType = 100.GetType();
24:      Console.WriteLine("Тип литерала 100:" +
25:      anyType.FullName);
26:      anyType = 200.45.GetType();
```

```
27:     Console.WriteLine("Тип литерала 200.45: " +
28:     anyType.FullName);
29:     anyType = (age * mass).GetType();
30:     Console.WriteLine("Тип выражения (age * mass):" +
31:     anyType.FullName);
32:     anyType = (age + height).GetType();
33:     Console.WriteLine("Тип выражения " +
34:     "(age + height) is:" +anyType.FullName);
35:     anyType = ((age * mass) * (energy + height)).GetType();
36:     Console.WriteLine("Тип выражения " +
37:     "((age * mass) * (energy + height)): " +
38:     anyType.FullName);
39:   }
```

Результаты работы программы

```
Тип переменной age: System.Byte
Переменная age имеет простой тип
Переменная age не имеет тип класс
Тип литерала 100: System.Int32
Тип литерала 200.45: System.Double
Тип выражения (age * mass): System.Decimal
Тип выражения (age + height): System.Int32
Тип выражения ((age * mass) * (energy + height)): System.Decimal
```

В строке **10** объявлена переменная **anyType**, содержащая ссылку на объект класса класса **Type**. Переменная anyType ссылается на объект, из которого будет получена информация о метаданных определенного типа.

В строке **16** вызывается метод **GetType**() с переменной **age** и **оператором точки**. **GetType()** возвращает объект класса **Type**, содержащий метаданные о типе **age**, — в данном случае, **byte**. Объект **Type** присваивается переменной **anyType**, из которой можно получить информацию о метаданных типа **byte**.

В строках **17** и **18** вызывается свойство **FullName** объекта **anyType**, которое выводится на консоль. Как и следовало ожидать (см. предыдущий вывод), типом является **System.Byte**.

В строках **19** и **20** представлен оператор **if**. Свойство **IsPrimitive** (см. табл. 4.5) позволяет определить, что тип **byte** является простым. Если это так, **anyType.IsPrimitive** имеет значение **true**, а программа выводит сообщение " **Переменная age имеет простой тип** ". Именно это мы и видим в выводе.

В строках **21** и **22** значение **false** типа **bool** применяется в операторе **if**. Если **anyType.IsClass** возвращает **false**, условие (**anyType.IsClass == false**) будет равно **true** и на консоль будет выведена строка " **Переменная age не имеет тип класс** ", **byte** является типом значений (структурой), а не классом, поэтому **anyType.IsClass** равно **false**, а (**anyType.IsClass == false**), соответственно, — **true**, что и приводит к выводу показанного текста.

В строке **23** метод **GetType()** вызывается напрямую для значения **100**, а возвращаемый объект присваивается **anyType**.

Вывод в строках **24** и **25** подтверждает, что типом **100** является **System.Int32**.

Строки **29—31** демонстрируют возможность доступа к объекту **Type** значения, возвращаемого для выражения, являющегося объединением двух переменных и операции. Переменные имеют тип **byte** и **decimal**, что дает в результате **decimal** (**System.Decimal**).

Строка **34** показывает, что и более длинное выражение можно исследовать посредством метода **GetType().**

В листинге 4.4 применяется довольно длинный путь доступа к метаданным.

Вместо записи

```
10:     Type anyType;
16:     anyType = age.GetType();
17:     Console.WriteLine("Тип переменной age:"
18:     + anyType.FullName);
```

где объект Type возвращается методом GetType() и присваивается anyType, можно отбросить строки 10 и 16 и просто записать

```
17:     Console.WriteLine("Тип переменной age:"
18:     + age.GetType.FullName);
```

После того как метод **age.GetType()** возвращает объект **Type**, конструкцию **age.GetType()** можно рассматривать как объект **Type**. Следовательно, в строке 18 можно использовать конструкцию **age.GetType().FullName**.

// ВЗАИМОДЕЙСТВИЕ РАЗЛИЧНЫХ ЯЗЫКОВ

Свойство **FullName** всегда возвращает имя типа в длинной форме **.NET**-платформы (**System.Int32**), а не в краткой записи C# (**int**). Так происходит, потому что все языки программирования, работающие на **.NET**-платформе, могут вызывать **GetType**, **Type** и **FullName**. Так же, как **int** представляет собой псевдоним **System.Int32** в C#, в других языках для этого же типа могут использоваться другие имена. Например, в языке **Eiffel#** как псевдоним **System.Int32** применяется **INTEGER**.

Каждый язык в **.NET Framework** поддерживает простые типы, обеспечиваемые Единой системой типов **CTS**. Поэтому, применяя **отражение на компонент**, написанный на языке Eiffel, Visual Basic, Perl или любом другом языке **.NET**-платформы, мы получим только знакомые имена или атрибуты. Эта мощная обобщающая концепция позволяет компонентам взаимодействовать на достаточно детализованном уровне.

4.2. Еще о Метаданных

Хотя метаданные используются для описания и ссылки на все типы, определенные системой **VOS** (**Virtual Object System**), у них есть и другое назначение. Определенные при написании программы типы – как значений, так и ссылок - включаются в систему **.NET** при помощи объявлений типов, которые описаны в метаданных, содержащихся в **PE-файле**.

Метаданные используются при выполнении различных задач, таких как: **1)** представление информации, используемой в среде **CLR** (**Common Language Runtime**), **2)** для поиска и загрузки классов, **3)** размещение экземпляров этих классов в памяти, **4)** вызов нужных методов, **5)** трансляция **IL**-кода (**Intermediate Language**) в «родной» код, **6)** повышение безопасности и установка границ контекста среды выполнения.

О создании метаданных не нужно заботиться. Они создаются компилятором из кода C# в **IL**-код (а не **JIT**-компилятором – **Just-in-Time**). Компилятор вносит двоичные метаданные в **PE**-файл.

Основное преимущество включения метаданных в исполняемый код состоит в том, что информация о типах сохраняется **вместе** с самими типами, а не распределяется по нескольким местам. Это также помогает справиться с проблемами управления версиями, существующими в модели **COM**. Кроме того, в среде **CLR** можно использовать различные

версии библиотек в одном и том же контексте, так как обращение к библиотекам происходит не только через записи в системном реестре, но и через содержащиеся в исполняемом коде метаданные.

5. Компонентно-ориентированное программирование

Несмотря на то, что язык C# можно использовать для написания приложений практически любого типа, *одной из самых значительных областей его применения является создание компонентов*. Компонентно-ориентированное программирование настолько существенно для C#, что его иногда называют компонентно-ориентированным языком (component-oriented language). Поскольку C# и среда .NET Framework разрабатывались с ориентацией на компоненты, компонентная модель программирования здесь в значительной степени упрощена по сравнению с более ранними решениями.

5. 1. Повторное использование программного обеспечения

На ранней стадии написания каждая программа представляется совершенно независимым проектом, разрабатываемым с нуля. Однако это не слишком рациональный способ создания программ, и сегодня большинство из них создается по иной методологии.

Существует альтернатива — *повторное использование заранее разработанных и отлаженных программ или их частей.*

Высокая степень повторного использования кода означает, что при создании программы разработчик написал только часть кода, а остальная часть — это вставленные в программу компоненты, написанные и отлаженные **опытными** программистами.

// ПРИМЕЧАНИЕ

Даже в простейших программах на C#, следует всегда придерживаться концепции повторного использования кода.

Повторное использование программного обеспечения принимает множество форм; вот лишь одна из них:

"Класс" как единица повторно используемого кода. Одна из наиболее привлекательных сторон объектно-ориентированных языков (в том числе, C#) — это тщательно продуманная поддержка повторного использования кода. В частности, **class** оказался очень хорошим способом повторно использовать код.

Для иллюстрации этого факта рассмотрим следующую ситуацию.

Компания Big Finance Ltd. построила десятиэтажное офисное здание и установила там лифтовую систему. Для определения нужного размера и количества лифтов *объектно-ориентированный программист* из небольшой компании, производящей ПО, создал программу, которая *моделировала лифтовую систему и таким образом ответила на вопросы* Big Finance Ltd.

Другая компания, Very Big Finance Ltd., узнав об успешном выполнении проекта, предложила смоделировать лифтовую систему для еще большего здания с большим количеством лифтов. Казалось бы, нужно полностью отбросить предыдущий проект и начать работу над новым с нуля. Однако анализ показывает, что новые лифты практически не отличаются от старых. Благодаря своей объектно-ориентированной природе, *класс* **Elevator** из предыдущего проекта содержит и инкапсулирует *все атрибуты и методы*, необходимые для описания лифта в новом проекте. Следовательно, можно вставить **класс Elevator** в новую программу и повторно использовать его в ней в качестве программного компонента.

Повторное использование программного кода заложено в основу программирования в .NET и C#.

5.2. Сборка — основа повторного использования кода в .NET

Растущая популярность повторного использования кода привела к возникновению нового термина, ***компонентно-ориентированное программирование***, который подразумевает не только объектно-ориентированное программирование, но и встроенные механизмы, упрощающие повторное использование кода. Чтобы понять, как в **.NET** реализована поддержка компонентно-ориентированного программирования, нужно более подробно рассмотреть природу классов и повторного использования кода.

//ПРИМЕЧАНИЕ

В данном контексте повторное использование кода подразумевает повторное использование на уровне классов. В этом процессе, очевидно, может участвовать более одного класса.

1. ***Классы часто взаимодействуют***. Объекты одних классов при выполнении своих задач часто взаимодействуют с объектами других классов. Например, при моделировании лифтовой системы, объекты классов **Building** (строение), **People** (люди), **Elevator** (лифт), **Button** (кнопка) и т.д. взаимодействуют между собой.

2. ***Категории классов***. Часто говорят, что класс принадлежит той же категории, что и другой класс. Например, **один класс** позволяет вычислять квадратный корень числа и предоставляет доступ к различным логарифмическим функциям. **Другой класс** обеспечивает вычисление тригонометрических функций, например, синус и косинус. Оба класса принадлежат категории **Math** — классов, реализующих математические операции.

3. ***Библиотеки классов***. **Объединение классов**, принадлежащих одной и той же категории, **в один контейнер** упрощает обращение с ними. Такие **коллекции классов часто называют** *библиотеками классов***.**

4. ***Классы и ресурсы***. Классы или библиотеки классов часто используют различные ресурсы, которые нельзя назвать компьютерными программами. *Примерами ресурсов могут служить изображения и звуки*.

5. ***Классы и файлы***. Классы и библиотеки классов хранятся в виде файлов на диске. Когда классы активизируются и используется их функциональность, они компилируются и исполняются процессором в оперативной памяти компьютера.

Элементом повторного использования кода в **.NET** является сборка (assembly). По этой причине сборку называют компонентом. Любая программа в **.NET** и **C#** состоит из одной или более сборок. Далее показано, как сборка применяет природу классов и их повторное использование для реализации **компонентно**-ориентированного программирования. Но вначале следует ближе познакомиться с тем, *что такое сборка сама по себе*.

Сборка — это логический пакет, содержащий свое описание. Он состоит: **1)** из кода **MSIL** (Microsoft Intermediate Language), **2)** метаданных и, **3)** если необходимо, ресурсов, например изображений. Сборкой является любая программа, написанная для **.NET**, будь то: **1)** компонент для повторного использования (**dll**-файл) или **2)** самодостаточная исполняемая программа (**exe**-файл).

Сборку можно рассматривать с двух точек зрения:

1. С точки зрения разработчиков сборки, рассматривающих ее изнутри, на уровне исходного кода.

2. С точки зрения пользователей сборки, рассматривающих ее снаружи, когда требуется подходящий компонент для повторного использования в том или ином проекте.

5.3. Сборка с точки зрения ее разработчика

Обратившись к процессу компиляции, показанному на **рис. 1.8** (см. Пособие к Практ. занятиям - №1, стр. 28), можно видеть, что сборкой являются результаты работы любого из **.NET**-совместимых компиляторов.

На рис. 4.4 схематически показана сборка, сгенерированная компилятором C#. На рисунке представлены также и менее значимые части данного обсуждения; они включены в рисунок для того, чтобы читатель расширил свое представление обо всем процессе компиляции и исполнения программ на C#.

Вначале разработчик пишет исходный код на C# (**1** на рис. 4.4). Возможно, он хочет повторно использовать код из других, уже существующих, сборок в данной исполняемой программе. Для этого компилятору дается указание включить соответствующие сборки в процесс компиляции (**2** на рис. 4.4). В зависимости от предпочтений программиста, результатом компиляции будет либо **Portable Executable File** (**PE**-файл с расширением **.exe**), либо **Dynamic Link Library File** (**DLL**-файл с расширением **.dll**). **PE**-файл можно исполнять, просто запустив его на исполнение, и повторно использовать в других программах как компонент. **DLL-файл нельзя исполнять сам по себе** — это компонент, предназначенный исключительно для повторного использования в составе какого-либо приложения.

Код **MSIL** генерируется в процессе компиляции и является частью сборки.

Далее компилятор создает **манифест**, состоящий из **метаданных**, предназначенных для представления данной сборки. **Манифест** содержит описательную информацию о классах, методах и ресурсах, доступных в данной сборке. Это позволяет программисту просматривать важные фрагменты каждой сборки и принимать решение, подходят ли они для повторного использования в его проекте. Каждая сборка может зависеть от других сборок (используя их код). В манифесте перечислены все сборки, от которых зависит данная. Последние определяются в шаге **2** на рис. 4.4 (см. также **2** на рис. 4.5 далее). Связи, представленные в виде стрелок, определены в манифесте.

// ПРИМЕЧАНИЕ

Сборка может зависеть от других сборок в части реализации собственной функциональности, как уже упоминалось ранее, — все это определено в манифесте. Но манифест косвенным образом также связывает свою сборку с остальными. Термин "косвенным образом" применяется здесь потому, что связи, используемые в манифесте, имеют чисто описательный характер.

5.4. Сборка с точки зрения пользователя

Несмотря на то, что сборка *объединяет в себе отдельные файлы*, она *логически целостна*, если рассматривать ее извне.

Манифест позволяет среде выполнения **открывать классы и их методы** пользователю сборки (см. рис. 4.5). Последний может затем выбрать, какую функциональность из сборки он будет использовать повторно.

Разделение логической сущности, видимой снаружи, и физических элементов (файлов), видимых изнутри, дает разработчику сборки большую свободу в том, сколько файлов нужно для сборки, каков размер этих файлов и где они расположены.

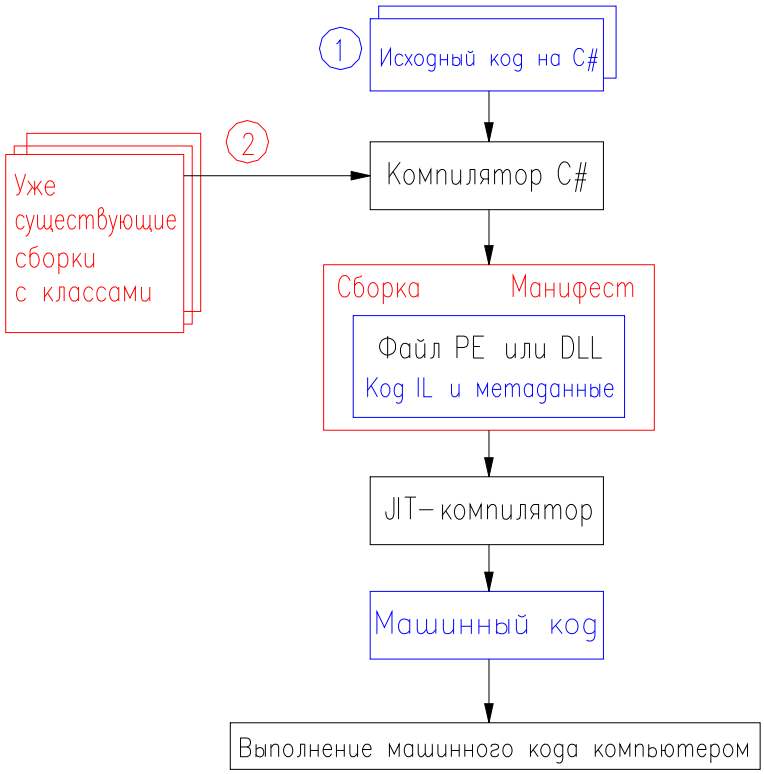


Рис. 4.4. Создание сборки путем использования уже существующих сборок

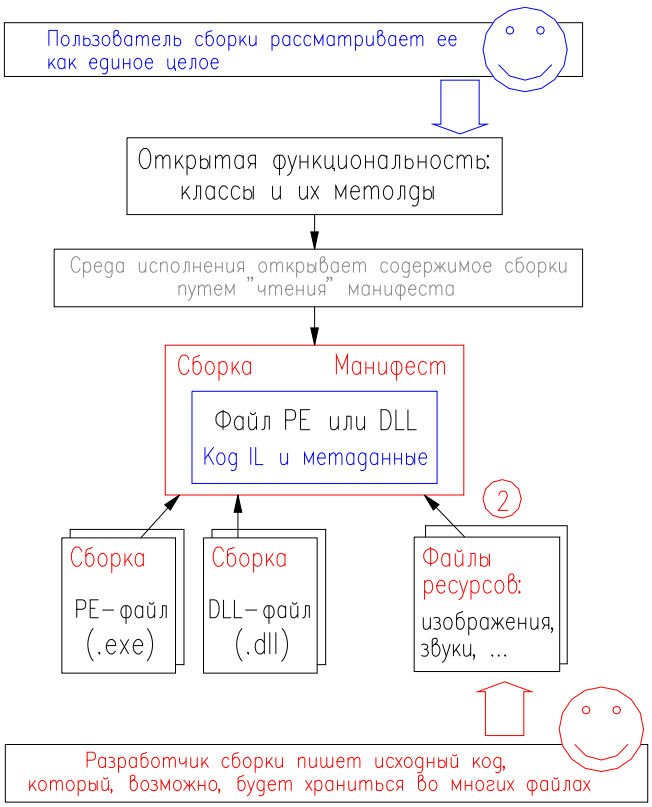


РИС. 4.5. Сборка с точки зрения пользователя и разработчика

Теперь можно оценить сборку как элемент повторного использования кода в **.NET**. За **счет использования метаданных** (их описательных возможностей) сборка открывает свою функциональность пользователю, а также определяет необходимые классы и ресурсы.

Вернемся к пяти пунктам (разд. 5.2, с. 22 с), в которых описывалась природа классов и повторного использования кода, и посмотрим, какое отношение сборки и **.NET** имеют к каждому из них.

1. ***Классы часто взаимодействуют***. Каждый класс, размещенный внутри файла **.exe** или **.dll**, может с помощью метаданных представлять свою функциональность и посредством этого разрешать зависимости для других классов.

2. ***Категории классов***. Формировать сборки, содержащие различные категории классов, достаточно просто.

3. ***Библиотеки классов***. Контейнер, упоминаемый в этом пункте, в **.NET** является сборкой. **Метаданные**, содержащиеся в манифесте, позволяют просматривать классы, методы и переменные экземпляра **в контейнере (сборке).**

4. ***Классы и ресурсы***. Если классам в сборке нужны различные ресурсы, достаточно включить в ее состав файлы с ними.

5. ***Классы и файлы***. Как показано выше, физическое представление сборки обладает исключительной гибкостью. Разработчик свободен в выборе количества, содержимого и расположения файлов в сборке. С точки зрения пользователя сборки эти условия не влияют на ее представление и функциональность.

5.5. Что представляет собой компонент

Начнем с определения термина компонент. Компонент — это независимый модуль, предназначенный для многократного использования и предоставляемый пользователю в двоичном формате. Это определение описывает четыре ключевых характеристики компонента:

Компонент определен как независимый модуль. Это означает, что каждый компонент вполне самодостаточен. Другими словами, компонент обеспечивает полный набор функций. Его внутренняя работа закрыта для "внешнего мира", но при этом реализация может быть изменена без последствий для кода, в котором используется этот компонент.

Компонент предназначен для многократного применения. Это означает, что компонент может использовать любая программа, которой требуются его функции. Программа, которая использует компонент, называется клиентом (**client**). Таким образом, компонент может работать с любым количеством клиентов.

Компонент представляет собой отдельный модуль. Это очень важно. С точки зрения клиента компонент выполняет конкретную функцию или набор функций. Функциями компонента, может воспользоваться любое приложение, но сам компонент не является автономной программой.

Наконец, компонент должен быть представлен в двоичном формате. Это принципиально важно. Хотя использовать компонент могут многие клиенты, они не имеют доступа к его исходному коду. Функциональность компонента открыта для клиентов только посредством его **public**-членов. Другими словами, именно компонент управляет тем, какие функции оставлять открытыми для клиентов, а какие — держать "под замком".

5.6. Компонентная модель

Несмотря на то, что приведенное выше определение точно описывает программный компонент, для полного его понимания (и использования) этого недостаточно. Принципиально важное значение здесь имеет модель, которая реализует компоненты. Для того чтобы клиент мог использовать компонент, необходимо, чтобы и клиент, и компонент использовали один и тот же набор правил. Набор правил, определяющих форму и поведение компонента, и называется **компонентной моделью** (component model). Именно компонентная модель определяет характер взаимодействия компонента и модели.

Для эффективного использования компонентов клиенты и сами компоненты должны подчиняться правилам, определенным компонентной моделью. По сути, компонентная модель представляет своего рода контракт между клиентом и компонентом, который обе стороны согласны выполнять.

5.7. Что представляет собой C#-компонент

Благодаря особенностям работы средств языка C#, любой его класс полностью соответствует общему определению компонента. Для того чтобы класс стал компонентом, он должен следовать компонентной модели, определенной средой **.NET Framework**. Этого совсем не трудно добиться: такой класс должен реализовать интерфейс **System.ComponentModel.**IComponent. При реализации интерфейса **IComponent** компонент удовлетворяет набору правил, позволяющих компоненту быть совместимым со средой **.NET Framework.**

Несмотря на простоту реализации интерфейса **IComponent**, во многих ситуациях лучше использовать альтернативный вариант — класс **System.ComponentModel.**Component. Класс **Component** обеспечивает стандартную реализацию интерфейса **IComponent**. Он также поддерживает другие полезные средства, свойственные компонентам. Опыт показывает, что большинству создателей компонентов удобнее выводить их из класса Component, чем самим реализовать интерфейс **IComponent**, поскольку в первом случае нужно попросту меньше программировать.

5.8. Контейнеры и узлы

С C#-компонентами тесно связаны две другие конструкции: контейнеры и узлы. Контейнер — это группа компонентов. Контейнеры упрощают программы, в которых используется множество компонентов. Узел позволяет связывать компоненты и контейнеры. Подробнее обе эти конструкции рассмотрим далее.

5.9. Интерфейс IComponent

Интерфейс **IComponent** определяет правило, которому должны следовать все компоненты. В интерфейсе **IComponent** определено только одно свойство (**Site**) и одно событие (**Disposed**). Вот как объявляется свойство **Site**:

```
ISite Site {  get;  set; }
```

Свойство **Site** получает (get) или устанавливает (set) узел компонента. Узел идентифицирует компонент. Это свойство имеет **null**-значение, если компонент не хранится в контейнере.

Событие, определенное в интерфейсе **IComponent**, носит имя **Disposed** и объявляется так:

```
event EventHandler Disposed
```

Клиент, которому нужно получить уведомление при разрушении компонента, регистрирует обработчик событий посредством события **Disposed**.

Интерфейс **IComponent** также наследует интерфейс **System.IComponent**, в котором определен метод **Dispose( ) :**

```
void Dispose( )
```

Этот метод освобождает ресурсы, используемые объектом.

5.10. Класс Component

Несмотря на то, что для создания компонента достаточно реализовать интерфейс **IComponent**, намного проще создать класс, производный от класса Component, поскольку он реализует интерфейс **IComponent** по умолчанию. Если класс наследует класс **Component**, значит, он автоматически выполняет правила, необходимые для получения **.NET**-совместимого компонента.

В классе **Component** **определен** только конструктор по умолчанию. Обычно программисты не создают объект класса **Component** напрямую, поскольку основное назначение этого класса — быть базовым для создаваемых компонентов.

В классе **Component** определено два открытых свойства. Объявление первого свойства Container, такое:

```
public  IContainer Container {  get; }
```

Свойство **Container** возвращает контейнер, который содержит вызывающий компонент. Если компонента нет в контейнере, возвращается значение **null**. Следует помнить, что свойство **Container** устанавливается не компонентом, а контейнером.

Второе свойство, **Site**, определено в интерфейсе **IComponent**. В классе **Component** оно реализовано как виртуальное:

```
public virtual  ISite Site {  get;  set; }
```

Свойство **Site** возвращает или устанавливает объект типа **ISite**, связанный с компонентом. Оно возвращает значение **null**, если компонента в контейнере нет. Свойство **Site** устанавливается не компонентом, а контейнером.

В классе Component определено два открытых метода. Первый представляет собой переопределение метода **ToString().** Второй, метод **Dispose(),** используется в двух формах. Первая из них такова:

```
public void Dispose()
```

Метод **Dispose(),** вызванный в первой форме, освобождает любые ресурсы, используемые вызывающим компонентом. Этот метод реализует метод **Dispose(),** определенный в интерфейсе **IDisposable**. Для освобождения компонента и занимаемых им ресурсов клиент вызывает именно эту версию метода **Dispose().**

Вот как выглядит вторая форма метода Dispose():

```
protected virtual public void Dispose(bool  how)
```

Если параметр ***how*** имеет значение **true**, эта версия метода (**1-я**) освобождает как управляемые, так и неуправляемые ресурсы, используемые вызывающим объектом. Если ***how*** равно значению **false**, освобождаются только неуправляемые ресурсы (**2-я версия**).

Поскольку эта версия метода **Dispose()** защищена (**protected**), ее нельзя вызвать из кода клиента. Поэтому клиент использует первую версию. Другими словами, вызов первой версии метода **Dispose()** генерирует обращение к методу **Dispose(bool).**

В общем случае компонент, в котором больше нет необходимости, переопределяет версию **Dispose(bool)**, если он содержит ресурсы, которые нужно освободить. Если компонент не занимает никаких ресурсов, то для его освобождения достаточно стандартной реализации метода **Dispose(bool)**, определенной в классе **Component**.

Класс **Component** наследует класс **MarshalByRefObject,** который используется в том случае, когда компонент создается вне локальной среды, например в другом процессе или на другом компьютере, связанном с первым по сети. Для обмена данными (аргументами методов и возвращаемыми значениями) должен существовать механизм, который определит способ пересылки данных. По умолчанию принимается, что информация должна передаваться по значению, но при унаследовании класса **MarshalByRefObject** данные будут передаваться по ссылке. Таким образом, C#-компонент обеспечивает передачу данных по ссылке.

5.11. Простой компонент

Рассмотрим первый пример создания компонента **CipherLib,** который реализует простую стратегию шифрования текста. Ее суть состоит в том, что каждый символ шифруется путем добавления к его **ASCII**-коду единицы. Дешифрирование заключается в вычитании единицы. Чтобы зашифровать таким способом строку, достаточно вызвать метод **Encode(),** передав незашифрованный текст в качестве аргумента. Чтобы дешифровать зашифрованную строку, вызовите метод **Decode(),** передав ему как аргумент зашифрованный текст. В обоих случаях возвращаются строки, содержащие результат (шифрования или дешифрования).

Листинг 4.5 – исходный код **CipherLib.cs**

```
1:      // File CipherLib.cs.       Making the simple component-coder ( CipherLib.dll )
2:  using System.ComponentModel;
3:
4:  namespace CipherLib
5:  {                                  // Class CipherComp inherits base  class Component
6:      public class CipherComp : Component
7:      {
8:
9:          public string Encode(string msg)                  // We Encode line
10:         {
11:             string temp = "";
12:
13:             for (int i = 0; i < msg.Length; i++)
14:                 temp += (char)(msg[i] + 1);
15:             return temp;
16:         }
17:
18:         public string Decode(string msg)                  // Decover line
19:         {
20:             string temp = "";
21:
22:             for (int i = 0; i < msg.Length; i++)
23:                 temp += (char)(msg[i] - 1);
24:             return temp;
```

```
25:         }
26:     }
27: }
```

Итак, рассмотрим этот код подробнее. Прежде всего, как предлагается в комментарии, назовем этот файл **CipherLib.cs**. Это упростит использование рассматриваемого компонента, если вы работаете в среде разработки **Visual Studio IDE**. Затем обратите внимание на включение пространства имен **System.ComponentModel** (**строка 2**). Как упоминалось выше, оно как раз и предназначено для поддержки программирования компонентов.

Класс **CipherComp** создается в собственном пространстве имен **CipherLib**. Это позволяет защитить глобальное пространство имен от загромождения новыми именами. Несмотря на то, что такой подход формально необязателен, предложенный стиль программирования приветствуется.

Класс **CipherComp** наследует класс **Component** (**строка 6**). Эго означает, что класс **CipherComp**. удовлетворяет всем требованиям для того, чтобы быть **.NET**-совместимым компонентом. Поскольку класс **CipherComp** очень простой, ему не нужно обеспечивать выполнение специфических для компонентов функций. Его действия можно определить как тривиальные.

Обратите также внимание на то, что класс **CipherComp** не распределяет системных ресурсов. Другими словами, он не хранит ссылок на другие объекты. Он просто определяет два метода **Encode()** (**строка 9**) и **Decode()** (**строка 18**) . А поскольку в классе CipherComp ссылки не хранятся, ему не нужно реализовать метод **Dispose(bool)**. Безусловно, оба метода **Encode()** и **Decode()** возвращают ссылки на строки, но эти ссылки принадлежат вызывающему коду, а не объекту класса **CipherComp**.

5.12. Компиляция компонента CipherLib

Компонент должен быть скомпилирован с получением **dll-**, а не **exe**-файла. Если вы работаете в среде разработки **Visual Studio IDE**, то для построения компонента **CipherLib** необходимо создать проект библиотеки классов (**Class Library project**). Если вы предпочитает работать с компилятором командной строки, задайте **/t:library**. Например, чтобы скомпилировать компонент **CipherLib**, используйте следующую командную строку (удобнее воспользоваться файлом 1.bat):

```
C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\csc  /t:library CipherLib.cs
```

В результате выполнения этой команды будет создан файл **CipherLib.dll**, содержащий компонент **CipherLib**.

5.13. Клиент, использующий компонент CipherLib (файл CipherLib.dll)

После создания компонент "готов к употреблению". Например, следующая программа является клиентом компонента **CipherLib**, который она использует для шифрования и дешифрирования символьной строки.

Листинг 4.6 – исходный код CipherClient.cs

```
1:          // File CipherClient.cs.                    Client uses component CipherLib.dll
2: using System;
3: using CipherLib; // Space of the names of the component CipherLib.dll is Imported
4:
5: namespace ConsoleApplication12
6: {
7:     class CipherClient
8:     {
9:         public static void Main()
```

```
10:         {
11:             CipherComp cc = new CipherComp();
12:
13:             string text = " This plain text ";
14:
15:                                 //Calling the method Encode() for cryptooperation of the text
16:             string ciphertext = cc.Encode(text);
17:             Console.WriteLine("\n\n\n\t" + ciphertext);
18:
19:                                     // Calling the method Decode() for decryption of the text
20:             string plaintext = cc.Decode(ciphertext);
21:             Console.WriteLine("\n\n\t" + plaintext);
22:
23:             cc.Dispose();                                   // We Free facility
24:             Console.ReadLine();
25:         }
26:     }
27: }
```

Заметьте, что клиент включает пространство имен компонента **CipherLib (строка 3)**. Благодаря этому компонент **CipherLIb** попадает в "поле зрения" клиента. Можно было бы полностью определять каждую ссылку на компонент CipherLib, но включение его пространства имен упрощает работу с компонентом. Во всем остальном компонент CipherLib используется подобно любому другому классу.

Обратите внимание на обращение к методу Dispose() в конце программы **(строка 23)**. Как разъяснялось выше, посредством вызова метода Dispose() клиент освобождает ресурсы, которые использовал компонент. Компоненты, подобно другим объектам C#, используют один и тот же механизм сбора мусора, который выполняется спорадически. Однако вызов метода Dispose() заставляет компонент немедленно освободить свои ресурсы. Это очень важно в некоторых ситуациях, например, когда компонент удерживает такой ограниченный ресурс, как подключение к сети. Поскольку компонент CipherLib не занимает ресурсы, вызов метода Dispose() в этом примере не актуален. Но так как метод Dispose() является частью контракта компонентной модели, имеет смысл всегда вызывать его при завершении работы с компонентом.

Чтобы скомпилировать программу-клиента, необходимо сообщить компилятору о том, что он должен обратиться к компоненту **CipherLib.dll**. Для этого используется опция **/r:**. Например, нашу программу-клиент можно скомпилировать с помощью следующей командной строки:

```
C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\csc  /r:CipherLib.dll CipherClient.cs
```

При использовании среды разработки Visual Studio IDE необходимо для программы-клиента добавить компонент **CipherLib.dll** как ссылку.

При выполнении программы client получаем следующие результаты:

```
!Uijt!qmbjo!ufyu!   - это зашифрованный текст
This plain text             - это расшифрованный текст .
```

5.14. Компиляция нескольких модулей компиляции в сборку

Этапы создания и запуска программы можно представить так:

1. В редакторе создается исходный код.

2. Он сохраняется в файле (модуле компиляции) с расширением .cs.

3. Файл **.cs** компилируется командой **csc**, в результате чего создается исполняемый файл с расширением **.exe** (**PE**-файл, **PE**-сборка).

4. PE-файл запускается по его имени (без расширения **.exe**).

Фактически в примерах рассмотренных ранее создавался только один исходный файл и одна **PE**-сборка, как показано на рис. 4.6.

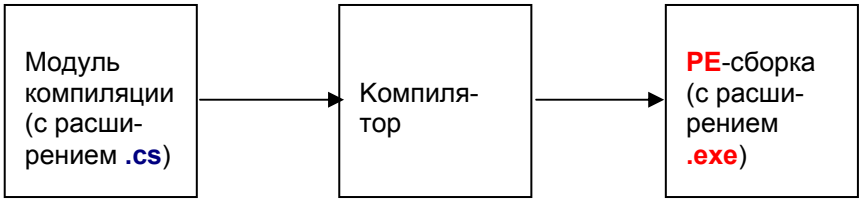


Рис. 4.6. Компиляция одного модуля в **PE**-сборку

Этот подход приемлем для изучения большинства элементов C#. Однако в реальных проектах, которые часто состоят из сотен страниц кода с огромным числом классов, созданных различными разработчиками, он неэффективен. Требуется возможность создавать большое количество физически отдельных исходных файлов, которые в то же время содержат классы, определенные в одном пространстве имен. На рис. 4.7 показано, как из трех модулей компиляции создается одна **PE**-сборка. Средства C# позволяют собирать неограниченное число модулей.

Следующий пример демонстрирует, как создать сборку из трех исходных файлов. Предположим, разрабатывается программа моделирования лифта, которая является частью пространства имен **CSharp**. Пусть для простоты в программе существует только три класса: **Elevator**, **Person** и **ElevatorSimulation** (содержащий метод **Main**). Причем они максимально сокращены. Пусть каждый класс содержится в одноименном исходном файле (см. листинги 4.7, 4.8 и 4.9). Каждый исходный файл имеет стандартное расширение **.cs**. Поскольку они связаны, их следует поместить в каталоге **ElevatorSimulation**.

// СОВЕТ
Принято, что каждый класс содержится в отдельном одноименном файле, например, класс **Elevator** в файле **Elevator.cs**.

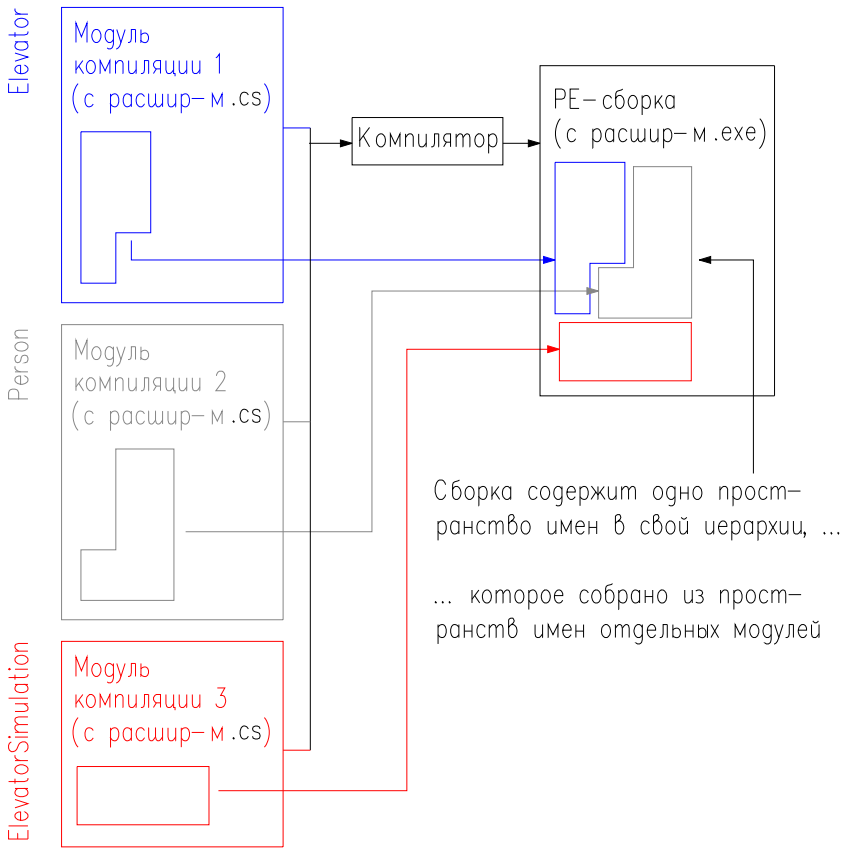


Рис. 4.7. Компиляция трех модулей в одну **PE**-сборку

Листинг 4.7. Исходный код **Elevator.cs**

```
01: using System;
02:
03: namespace CSharp.ElevatorSimulation
04: {
05:     internal class Elevator //"internal" - использование класса только в сборке
06:     {
07:       public static void PrintClassName()
08:       {
09:           Console.WriteLine("Этот класс назван Elevator, а файл назван Elevator.cs");
10:       }
11:     }
12:}
```

Листинг 4.8. Исходный код **Person.cs**

```
01: using System;
02:
03: namespace CSharp.ElevatorSimulation
04: {
05:     public class Person // "public" - использование класса и вне сборки
06:     {
07:       public void Talk()
08:       {
09:         Console.WriteLine("Раз, раз, два, раз");
10:       }
11:     }
12:}
```

Листинг 4.9. Исходный код **ElevatorSimulation.cs**

```
01: using System;
02:
03: namespace CSharp.ElevatorSimulation
04: {
05:     public class ElevatorSimulation
06:     {
07:             public static void Main()
08:             {
09:             Person somePerson = new Person();
10:             somePerson.Talk();
11:             Elevator.PrintClassName();
12:             }
13:     }
14:}
```

Итак, в каталоге **ElevatorSimulation** находятся три исходных файла: **Elevator.cs, Person.cs** и **ElevatorSimulation.cs.**

Все они определяют одно пространство имен: **Sharp.ElevatorSimulation** (см. строку 3 в приведенных листингах), поэтому каждый файл добавляет в него один класс.

Класс **ElevatorSimulation** может ссылаться на классы **Elevator** и **Person** (см. строки 9…11) без длинных имен, поскольку он определен в одном с ними пространстве имен.

Обратите внимание: класс **Elevator** объявлен как **internal** (строка 5 листинга 4.7), а класс **Person** — как **public** (строка 5 листинга 4.8). **internal** и **public** являются спецификаторами доступности: **public** позволяет использовать класс и вне сборки, а **internal** — только в сборке. В данном случае, когда исходный файл становится частью сборки, проявляются различия между этими двумя спецификаторами доступности. В нашем примере класс **Elevator** объявлен как **internal**, только для того, чтобы выявить это свойство.

// СПЕЦИФИКАТОР ДОСТУПНОСТИ INTERNAL

По умолчанию классу, который не имеет спецификатора доступности, присваивается спецификатор **internal**, т. е. его нельзя использовать за пределами сборки. Любой элемент класса может быть объявлен как **internal**. Это позволяет сделать часть класса доступной, объявив его **public**, а затем ограничить доступ к некоторым его элементам, объявив их **internal**.

До того как компилировать три исходных файла в сборку, необходимо рассмотреть несколько команд компилятора, представленных в табл. 4.6.

Их синтаксис предполагает, что они размещаются между командой csc и именем исходного файла. Сейчас рассматривается только команда **/out:**, об остальных — далее.

По умолчанию компилятор использует имя исходного файла как имя сборки. Чтобы задать другое имя и избежать путаницы при одновременной обработке нескольких файлов, можно включить команду **/out:** с последующим именем сборки.

Таблица 4.6. Несколько опций компилятора

Команда компилятора	**Описание**
/out: <Имя_файла>	Имя выходного файла (сборки).
/t[arget]**: exe**	Эта команда создает **PE**-сборку с расширением **.exe**, которая является консольным приложением, автоматически открывающим окно консоли. Данная опция компилятора устанавливается по умолчанию. (Совет: для графических **Windows**-приложений следует использовать команду **/t**[arget]**:winexe**. Это предотвратит открытие окна консоли.)
/t[arget]**: library**	Эта команда создает библиотеку классов, которая не исполнима сама по себе, а предназначена для использования другими приложениями. Созданная сборка имеет расширение **.dll**
/r[eference]**:** <Список_сборок>	Эта команда позволяет задать несколько сборок (с расширением .exe или **.dll**), причем пространство имен каждой из них становится доступным для создаваемого приложения. Примечание: имена сборок разделяются точками с запятой (см. пример в примечании).

Примечания.

- Запись **/t**[arget]**:** означает, что arget необязателен, поэтому **/t:** и **/target:** имеют одно значение. Та же логика применима и к **/r**[eference]**:**.
- Команды являются частью команды **csc**, задаваемой в командной строке с консоли. Каждая команда должна располагаться между командой **csc** и именами исходных файлов C# , как показано далее в обычной синтаксической нотации.

  Чтобы создать **.dll**-модуль сборки, который называется **MyClassLibrary.dll**, ссылается на библиотеки **ClassLibl.dll** и **ClassLib2.dll** и включает исходные файлы **Planet.cs** и **Rocket.cs**, воспользуйтесь командой:

csc **/out:**MyClassLibrary.dll **/t:library /r:**ClassLib1**;** ClassLib2 Planet.cs Rocket.cs

Итак, компилируем три исходных файла из листингов 4.7, 4.8 и 4.9. Для создания приложения **ElevatorSimulation.exe** используется команда:

```
C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\CSC.EXE

/out:ElevatorSimulation.exe /t:exe Elevator.cs Person.cs ElevatorSimulation.cs
```

В данном примере можно и опустить опцию **/t:exe**, поскольку это же ее значение используется по умолчанию.

При запуске программы **ElevatorSimulation.exe** (вводом ее имени в командной строке), на экране появляется:

```
Раз, раз, два, раз
Этот класс назван Лифт
```

5.15. Повторное использование пространств имен из сборки

Группе программистов, работающих над классом **SpaceShuttle**, требуется класс **Person**, способный выводить **Раз, раз, два, раз** (для тестирования акустики внутри **SpaceShuttle**). Они хотят повторно использовать класс **CSharp.ElevatorSimulation.Person**, в котором предусмотрена такая возможность. Для этого им необходимо знать:

1. Полное имя класса для ссылки на него в исходном коде. Другими словами, краткое имя класса и название его пространства имен.

2. Имя и местонахождение сборки, в которой находится пространство имен, чтобы обратиться к этой сборке командой **/r**[eference]**:** (см. таблицу 4.6) при компиляции исходного кода.

Вначале программисты создают фрагмент исходного кода, представленного в листинге 4.10, чтобы убедиться, что класс **CSharp.ElevatorSimulation.Person** можно повторно использовать. Если вы хотите компилировать программу **SpaceShuttle.cs**, нужно поместить этот файл в каталог, где находится сборка **ElevatorSimulation.exe**. Инструкции по компиляции представлены после рассмотрения кода.

Листинг 4.10. Исходный код **SpaceShuttle.cs**

```
01: using CSharp.ElevatorSimulation; // пространство имен – стр. 3 в листингах 4.7, 4.8 и 4.9
02:
03: namespace SpaceShuttleSimulation
04: {
05:     class SpaceShuttle
06:    {
07:     private static Person astronaut;
08:
09:     public static void Main()
10:     {
11:       astronaut = new Person();
12:       astronaut.Talk();
13:          // Elevator.PrintClassName();
14:     }
15:    }
16:}
```

Так же как включение **using System** обеспечивает удобное обращение к классам в пространстве имен **System** библиотеки класса **.NET**, так в данном случае в строке **1** в **SpaceShuttle.cs** указано пространство имен **CSharp.ElevatorSimulation** (созданное из листингов 4.7, 4.8 и 4.9).

Оператор в строке **1** позволяет применять краткое имя **Person** (в строках **7** и **11**) для обращения к классу **CSharp.ElevatorSimulation.Person**.

При компиляции исходного файла **SpaceShuttle.cs** необходимо указать, в каких сборках следует искать пространства имен, к которым происходит обращение в исходном коде. Для этого применяется команда **/r**[eference]**:**. В данном случае пространство имен расположено в сборке **ElevatorSimulation.exe**. В результате, для создания сборки SpaceShuttle.exe из исходного файла **SpaceShuttle.cs** используется команда:

```
C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\CSC.EXE
/r:ElevatorSimulation.exe SpaceShuttle.cs
```

После запуска программы на экран выводится сообщение:
Раз, раз, два, раз

// ПРИМЕЧАНИЕ
В команде компилятора можно указать несколько сборок (когда используемые в исходном коде пространства имен находятся в разных сборках). Пример этого приведен далее.

Напомним, что класс **CSharp.ElevatorSimulation.Elevator** в листинге 4.7 объявлен как **internal**. Созданная сборка **SpaceShuttle.exe** находится за пределами области сборки **ElevatorSimulation.exe**, где содержится класс **CSharp.ElevatorSimulation.Elevator**. Следовательно, при попытке использовать **CSharp.ElevatorSimulation.Elevator** в **SpaceShuttle.cs** для создания **SpaceShuttle.exe**, скорее всего, будет выведено сообщение об ошибке. Для проверки в листинге 4.10 необходимо раскомментировать **13**-ю строку:

```
Elevator.PrintClassName();
```

и попытаться компилировать **SpaceShuttle.cs** с помощью той же команды, что и раньше:

```
C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\CSC.EXE
/r:ElevatorSimulation.exe SpaceShuttle.cs
```

Компилятор выдаст сообщение:

SpaceShuttle.cs(13,24): error CS0122: 'CSharp.ElevatorSirnulation.Elevator' **is inaccessible due to its protection level**

// ПРИМЕЧАНИЕ
Для запуска **SpaceShuttle.exe** требуется исполняемый файл **ElevatorSimulation.exe.** Если файл **ElevatorSimulation.exe** перемещен из исходного каталога, то при попытке выполнить **SpaceShuttle.exe** платформа **.NET** генерирует исключение **System.TypeLoadException**.

5.16. Разделение пространства имен на несколько сборок

После рассмотрения кода **SpaceShuttle.cs** вернемся к исходным файлам **Elevator.cs, Person.cs**, **ElevatorSimulation.cs**, представленным в листингах 4.7, 4.8 и 4.9. Предположим, что листинги этих исходных файлов со временем станут чем-то большим, чем просто демонстрационными примерами. Допустим, будут разработаны классы **Elevator** и **Person**, выполняющие сложное моделирование, и для них будет создана иерархическая структура пространства имен других классов, предназначенная для работы с классом **Elevator** и клас-

сом **Person**. В результате объем исходных файлов будет доходить до сотен страниц. Однако все они будут компилироваться в одну сборку **Elevator.Simulation.exe**. Кроме того, предположим, что и другие программисты заинтересованы в повторном использовании классов **Elevator** и **Person** (и связанных с ними классов). Таким образом, треть сборок **Elevator.Simulation.exe** предназначена для программистов, заинтересованных в повторном использовании классов, связанных только с классом **Elevator** (см. рис. 4.8), другая треть — связанных с классом **Person**, а последняя — для экспертов, заинтересованных в эксплуатации всего приложения.

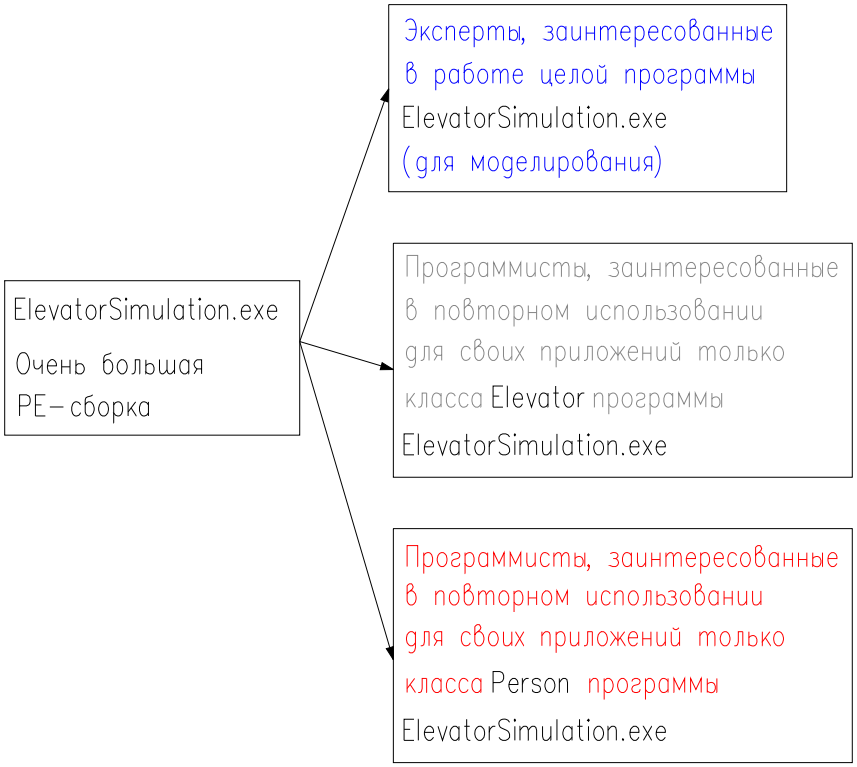


Рис. 4.8. Распространение программы ElevatorSimulation.exe среди клиентов

Однако при такой негибкой системе распределения, когда покупателям продается один и тот же **Elevator.Simulation.exe**, у программистов, заинтересованных в повторном использовании только некоторых частей этого программного продукта — или **Person**, или **Elevator**, имеется, как минимум, три проблемы:

1. Используется только часть сборки **Elevator.Simulation.exe**, а хранить приходится всю программу, загромождая память.

2. Ради части программы нужно покупать все приложение.

3. *Новые версии* классов **Elevator** и **Person**, вероятно, разрабатываются быстрее, чем вся сборка **ElevatorSimulation.exe**. И клиенту-программисту приходится ждать выхода новой версии всего продукта дольше, чем если бы он покупал только части программы.

Поэтому систему распределения нужно менять. Такая возможность есть, так как среда **.NET** и язык C# предусматривают очень гибкую организацию модулей компиляции, пространств имен и сборок. Программу **ElevatorSimulation.exe** необходимо разбить на три сборки (как показано на рис. 4.9): **Elevator.dll** содержит классы, связанные с **Elevator, Person.dll** — с **Person**, а программа **ElevatorSimulation.exe** не содержит модулей **Person** и **Elevator**, от которых она зависит. Вместо этого в ней используются ссылки на **Person.dll** и **Elevator.dll**.

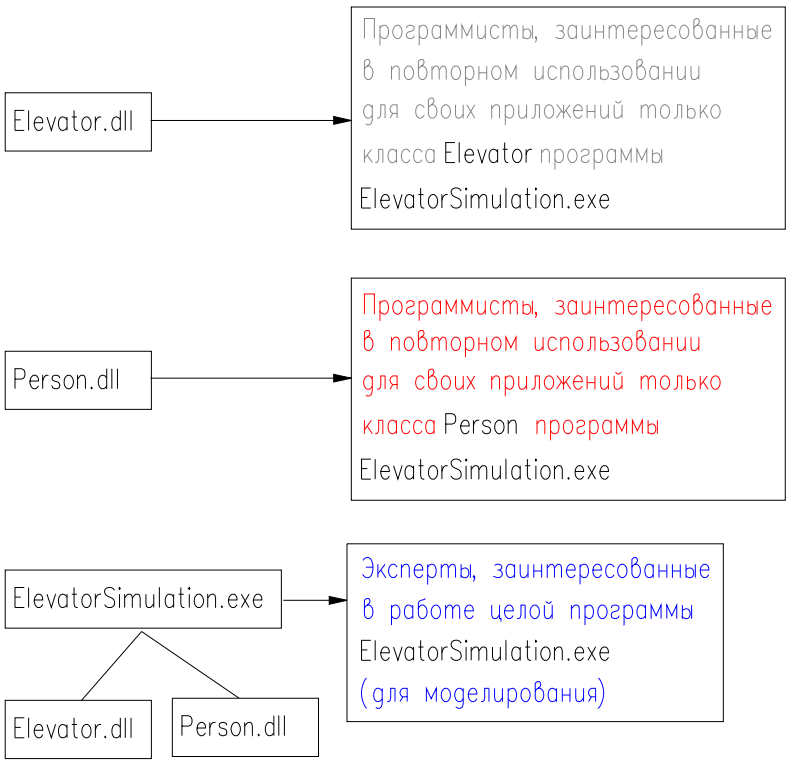


Рис. 4.9 Распределение упрощенных версий DLL-сборок среди программистов, заинтересованных в повторном их использовании

Для упрощения примера части **Elevator.cs, Person.cs** и **ElevatorSimulation.cs** невелики. Однако, реальный проект включает гораздо большее число исходных файлов, классов и т. д. Чтобы класс **CSharp.ElevatorSimulation.Elevator** из листинга 4.7 мог использоваться вне области его сборки, нужно заменить его спецификатор доступности **internal** на **public**. Это позволит повторно использовать этот класс не только программистам, но и в самой сборке **ElevatorSimulation.exe**, которая теперь находится в отдельном файле.

Чтобы отделить исходные файлы от уже использованных ранее (листинги 4.7, 4.8 и 4.9), можно сделать их копии с именами (**Elevator_Pro.cs, Person_Pro.cs** и **ElevatorSimulator_Pro.cs**). Теперь можно запустить компилятор. Итак, требуется создать одну **EXE-** и две **DLL-**сборки. Для создания последних применяется команда **/t**[arget]**: library** (табл. 4.6):

```
C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\CSC.EXE
/out:Elevator.dll /t:library Elevator_Pro.cs

C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\CSC.EXE
/out: Person.dll /t:library Person_Pro.cs
```

Версия программы **ElevatorSimulation.exe**, созданная ранее (листинг 4.9), содержала три исходных файла. Теперь указывается только файл **ElevatorSimulator_Pro.cs**, а также инструкции для поиска классов **Elevator** и **Person** в сборках **Elevator.dll** и **Person.dll**. Для этого используется команда **/r**[eference]**:<Список_сборок>**, как показано ниже:

```
C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\CSC.EXE
/out:ElevatorSimulation_Pro.exe /r:Elevator.dll;Person.dll ElevatorSimulator_Pro.cs
```

После исполнения программы на экране появится сообщение, подтверждающее, что ее результат не изменился:

```
Раз, раз, два, раз
                Этот класс назван Лифт
```

// ПРИМЕЧАНИЕ

Иерархия пространств имен в исполняемой сборке должна содержать один метод **Main**, с которого начинается исполнение программы. В **DLL**-файлах размещать этот метод не нужно, поскольку они не запускаются сами по себе, а предназначены только для повторного использования.

Программисты, разрабатывающие **SpaceShuttle.exe** и заинтересованные в использовании возможностей только класса **Csharp.ElevatorSimulation.Person**, могут теперь использовать сборку **Person.dll**. Для компиляции **SpaceShutlle.exe** (который ссылается на **Person.dll)** необходимо скопировать исходный файл в папку и запустить команду:

```
C:\WINDOWS\MICROSOFT.NET\FRAMEWORK\V3.5\CSC.EXE
/r:Person.dll SpaceShuttle.cs
```

При запуске программы **SpaceShuttle.exe** на экране появляется:

Раз, раз, два, раз

5.17. Просмотр сборок с помощью утилиты ildasm.exe

В **.NET SDK** содержится утилита *деассемблирования* в промежуточный язык (**ildasm.exe**), которая посредством удобного графического интерфейса позволяет просмотреть содержимое сборки: пространства имен, типы, элементы типов и т. д.

Утилита находится в папке **C:\Program Files\Microsoft Visual Studio .NET\SDK\v1.1\Bin**

После ее запуск (**ildasm<enter>**) на экране появится пустое окно **ILDASM**. Для открытия **EXE-** или **DLL**-сборки следует выбрать пункт **Open** меню **File**. Затем в диалоговом окне **File Open** выбрать требуемую сборку и дважды щелкнуть на ее имени.

При просмотре сборки **Elevator.dll** в окне будет отображено дерево сборки. Каждая ветвь дерева представляет часть сборки (пространство имен, класс, метод, свойство и т. д.) и обозначена соответствующей пиктограммой. В данном случае напротив **Elevator.dll** отображен голубой ромб. Под ним — щит (символизирующий пространство имен) — рядом со знакомым названием пространства имен, в котором находится класс **Csharp.ElevatorSimulation.Elevator**. Класс **Csharp.ElevatorSimulation.Elevator** (обозначен голубым прямоугольником с тремя соединениями, символизирующим класс) раскрытия узла пространства имен становится видным после. Щелчок на пиктограмме класса раскрывает дерево. В классе виден метод **PrintClassName** — статический метод объекта **Elevator** (обозначен лиловым прямоугольником с символом **'S'**). Лиловый прямоугольник без символа (над **PrintClassName**) обозначает нестатический метод. **Ctor** рядом с ним соответствует конструктору по умолчанию, автоматически созданному для класса **Elevator** компилятором C# (так как исходный код не содержит конструкторов, определенных явно).Обратите внимание на направленные вправо красные треугольники рядом с **MANIFEST** и **.class public auto ansi**. Эти символы говорят о том, что о сборке или типе есть дополнительная информация (метаданные). В манифесте (см. раздел 4, с. 17-21) хранятся данные о сборке **Elevator.dll**, а красный треугольник под голубым прямоугольником свидетельствует о том, что здесь хранится дополнительная информация и о классе **Elevator**.

После двойного щелчка на элементе класса на экране появляется окно с инструкциями промежуточного языка, в которые исходный код был преобразован при компиляции. В них может быть полезная информация.

5.18. Компоненты - это будущее программирования

Организация приложения в виде набора компонентов — это мощное средство программирования, позволяющее программисту справляться со все более сложными задачами. Программисты в начале своей деятельности замечают, что чем больше программа, тем дольше период ее отладки. С увеличением размера программы обычно растет и ее сложность, но известно, что существует некоторый предел сложности, с которым может справиться человек.

Программные компоненты помогают справиться со сложностью программ по принципу "разделяй и властвуй". Путем разделения программы на независимые компоненты программист может понизить видимый уровень ее сложности. При компонентно-ориентированном подходе программа организуется как набор строго определенных "строительных блоков" (компонентов), которые можно использовать, не вникая в детали их внутренней реализации. Суммарный эффект такого подхода состоит в снижении общей сложности программы.

Резюме

(к разделу 5 **Компонентно-ориентированное программирование** | использование **DLL**-файлов |)

Рассматривались модули компиляции, пространства имен и сборки, а также основные подходы к конфигурированию этих важных элементов **C#** и **.NET**. Пространства имен нужны:

- для организации классов и других типов данных в иерархические структуры при создании программы;
- как удобное средство доступа к классам и другим типам, расположенным в других компонентах.

Когда класс содержится в пространстве имен, его имя строится путем объединения названия пространства имен и краткого имени класса. Это предотвращает совпадение имен.

Пространство имен является логическим объектом, который может содержать несколько модулей компиляции и сборок.

Нуль и более пространств имен могут быть расположены последовательно или внутри друг друга в модуле компиляции или другом пространстве имен. Следовательно, глубина вложенности пространства имен может быть произвольной.

Классы и другие типы, определенные вне явно заданного пространства имен, автоматически принадлежат глобальному безымянному пространству имен.

Директива using применяется, во-первых, чтобы в исходном коде можно было использовать краткие и удобные имена классов, и, во-вторых, для создания псевдонимов пространств имен и классов (или других типов).

В одну сборку можно скомпилировать любое число модулей.

Используя команду компилятора **/t**[arget]**:** можно скомпилировать сборку как **DLL** (записав после команды слово **library**), т. е. как библиотеку для повторного использования, или как **EXE** (записав после команды **exe**), т. е. как исполняемый файл.

Содержащиеся в сборке пространства имен и классы (типы) можно сделать доступными для других **DLL**- или **EXE**-сборок посредством команды **/r**[eference]**:**.

Часто удобнее использовать несколько меньших **DLL**-сборок, чем одну большую **DLL**-или **EXE**-сборку.

Утилита **ildasm.exe**, содержащаяся в **.NET SDK**,— это простое средство для просмотра содержимого **DLL**- или **EXE**-сборок.

Контрольные вопросы

1. Почему существует два способа классификации типов в C#?

2. Почему в C# кроме **int** имеются и другие простые типы?

3. Какие атрибуты отличают разные типы друг от друга?

4. Что значит: быть строго типизированным языком наподобие C#?

5. Можно ли присвоить величину типа **int** переменной типа **short** без явного преобразования? Почему?

6. Как определить литерал так, чтобы он принадлежал типу **float**?

7. Напишите строку кода, в которой переменная экземпляра **distance** со спецификатором доступности **private** типа **float** объявлена и инициализирована значением 100.5.

8. Переменная типа **byte** содержит значение **255**. Программа пытается увеличить это значение на единицу. Что случится, если опция компилятора установлена как **unchecked**? Как **checked**?

9. Какой простой тип используется для переменной, хранящей вес человека (в фунтах; 1 фунт равен примерно 0.45 килограмма)?

10. Какой простой тип используется для хранения балансов счетов, которые участвуют в вычислениях процентной ставки, требующих чрезвычайно высокой точности?

11. Предположим, что обе переменные **myNumber** и **yourNumber** принадлежат типу **float**. Напишите выражение, которое складывает эти значения как числа типа **int** и присваивает результат переменной типа **int**.

12. Переменные **number1** и **number2** принадлежат типу **int**, а переменная **text1** — типу **string**. Каким будет тип следующего выражения? **number1** + **text** + **number2**

13. Чему равно следующее выражение в C#: **true** или **false**? Почему? ((10 * 0. 2f) == 2.0)

14. В каких случаях в исходном тексте программы следует использовать константы? В чем их преимущества?

15. Напишите оператор, который выводит на экран число 30000000.326m в формате $30,000,000.33

16. 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 действия.

17. Для чего применяется утилита **idles**?

Упражнение по программированию #1

Измените программу **DayCounter.cs** из листинга 4.2, добавив следующую функциональность:

1. Кроме вывода сообщения о воскресении, необходимо сообщать и о субботе. Подсказка: Программа определяет числа кратные 7: 7, 14, 21, 28 и т.д. Теперь требуется добавить и числа: 6, 13, 20, 27 и т.д. Как они связаны с числами воскресений? Как реализовать их поиск?

2. Пусть каждый год в моделирующей программе имеет 365 дней. Измените программу так, чтобы она выводила информацию в днях, неделях и годах.

Упражнение по программированию #2

Напишите программу, содержащую метод **AvarageAge**, который вычисляет средний из трех возрастов, передаваемых ему в качестве аргументов. Пользователь должен иметь возможность ввести три возраста (как целые числа). Используйте **AvarageAge** для вычисления среднего возраста и выведите результат с точностью до трех десятичных цифр. Какой тип возвращаемого значения используется для метода **AvarageAge**? Как отформатировать вывод?

Упражнение по программированию #3

1. Обратимся к первому объектно-ориентированному коду **SimpleElevatorSimulation.cs**, (см. Пособие к Практическому занятию №3, стр. 12, сл.). Здесь в одном модуле компиляции находятся три класса. В данном случае не было принято во внимание правило, по которому в одном модуле компиляции должен находиться только один класс. Исправьте это и перепишите программу так, чтобы каждый класс находился в отдельном модуле компиляции (**Elevator**, **Person** и **Building**), а также проследите, чтобы все классы были корректно размещены в иерархической структуре пространства имен. Затем скомпилируйте три модуля в одну **DLL**-сборку. (Подсказка: нужно убрать метод **Main** класса **Building**, поскольку сборка предназначена только для повторного использования.) Создайте другой исходный файл, выполняющий моделирование работы лифта так же, как исходная программа. В этом случае новая программа должна использовать пространства имен и классы, расположенные в сборке, созданной из трех классов.

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

Приложение 2

Простые типы C#

В C# определено 13 простых типов, которые перечислены в таблице.
При написании программ придется не раз вернуться к этой таблице.

Хотя тип bool (последняя строка таблицы) рассматривается здесь как простой тип, он связан с управлением потоком выполнения программ.

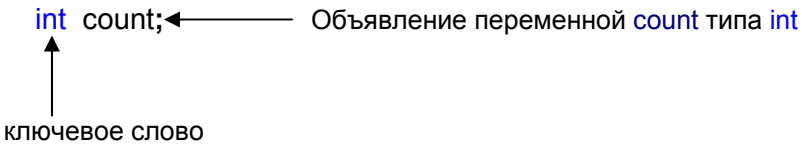
Таблица П2.1

Ключевое слово языка C# - псевдоним типа	**Тип .NET CTS (Common Type System) - полное имя типа**	**Вид значения**	**Исполь-зуемая память**	**Диапазон и точность**
sbyte	System.Sbyte	Целое число	8 битов	От -128 до127
byte	System.Byte	Целое число	8 битов	От 0 до *255*
short	System.Int16	Целое число	16 битов	От -32768 до 32767
ushort	System.Uint16	Целое число	16 битов	От 0 до 65535
int	System.Int32	Целое число	32 бита	От -2147483648 до 2147483647
uint	System.Uint32	Целое число	32 бита	От 0 до 4294967295
long	System.Int64	Целое число	64 бита	От -9223372036854775808 до 9223372036854775807
ulong	System.Uint64	Целое число	64 бита	От 0 до 18446744073709551615
char	System.Char	Целое число (один сим-вол)	16 битов	Все символы **Unicode**
float	System.Single	Число с пла-вающей точ-кой	32 бита	От (+/-)1.5 x 10^{-45} до (+/-) 3.4 x 10^{38}. Примерно 7 значащих цифр
double	System.Double	Число с пла-вающей точ-кой	64 бита	От (+/-)5.0 x 10^{-324} до (+/-)3.4 x 10^{308}. 15 -16 значащих цифр
decimal	System.Decimal	Десятичное число (высо-кой точности)	128 битов	От (+/-)1.0 x 10^{-28} до (+/-)7.9 x 10^{28}. 28 -29 значащих цифр
bool	System.Boolean	true или false	1 бит	Нет

Рекомендуется просмотреть характеристики знакомого типа int в пятой строке таблицы. Это дает хорошее представление о содержании каждого столбца.

Ниже представлена краткая сводка, что означает каждый столбец в таблице:

Столбец "Ключевое слово". Ключевое слово относится к символу, используемому в исходном коде C# при объявлении переменной. В качестве примера рассмотрим хорошо известный оператор объявления с использованием ключевого слова int.



Столбец "Тип .NET CTS". Пространство имен **System** .NET-Framework содержит все простые типы. Каждое ключевое слово, показанное в первом столбце, — это псевдоним типа, определенного в CTS. Например, ключевое слово **int** обозначает **System.Int32** в CTS. Таким образом, в исходном коде можно использовать как короткий псевдоним типа, так и его длинное полное имя. Два следующих выражения идентичны:

одинаковые объявления:

```
int myVariable;
System.Int32 myVariable;
```

Легко видеть, что последняя запись довольно громоздка, поэтому в программах рекомендуется использовать только псевдонимы.

Столбец "Вид значения". В этом столбце определены четыре различных группы содержащихся в C# простых типов: *1)* целое число, *2)* число с плавающей точкой, *3)* true / false и *4)* число высокой точности. Таким образом,

- Целые числа — числа без дробной части.
- Числа с плавающей точкой — числа с дробной частью.
- Числа высокой точности также представляют дробные величины, но, очевидно, с более высокой точностью.
- Величины типа bool могут содержать только два значения — **true** или **false**.

Столбец "Используемая память". Количество битов памяти, используемое величиной каждого типа.

Столбец "Диапазон и точность". Отображает диапазон и точность, обеспечиваемые величинами соответствующего типа. Следует отметить, что, хотя тип **char** разработан для отдельных символов, он рассматривается как целочисленный.

В таблице приведено **девять** целочисленных типов. Они отличаются друг от друга по трем параметрам: *1)* диапазону, *2)* объему занимаемой памяти и *3)* способности хранить отрицательные числа.

В таблице содержатся и **три** типа с плавающей точкой — **float, double** и **decimal**, используемые для хранения чисел, содержащих дробную часть (к примеру, 6.87, 9.0 и 100.01). Основные различия: *1)* диапазон, *2)* используемая память и *3)* точность.

П2.1. Тип decimal

Тип **decimal** — это **128**-битный тип с высокой точностью, предназначенный для финансовых и денежных вычислений. Он может представлять значения в диапазоне от

1.0×10^{-28} до 7.9×10^{28} с 28-29 значащими цифрами. Важно отметить, что точность задана в цифрах, а не в десятичных знаках. Все вычисления выполняются без округления до максимального значения в 28 десятичных знаков.

Как можно видеть, диапазон этих значений уже, чем в типе **double**, но они обладают намного большей точностью. Поэтому не существует неявного преобразования между типами **decimal** и **double**: в одном направлении такое преобразование может вызывать переполнение, в другом - привести к потере точности. Оно должно вызываться явно при помощи приведения типов.

Чтобы при определении переменной и присваивании ей значения обозначить тип значения как **decimal**, используется суффикс m:

```
decimal decMyValue =  1.0m;
```

Если опустить суффикс m, тогда до присваивания переменной значения компилятор будет рассматривать ее как **double**.