Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

Объектно-ориентированный анализ и программирование на языке C# (C_Sharp)

Материалы ко 2-й лекции. Часть 1

Проф. Забудский Е.И.

Москва 2006

Лекция 2, часть 1

Тема 2. Инкапсуляция – базовое понятие объектно-ориентированного программирования.

Инкапсуляция – объектно-ориентированная характеристика модульности. Внешний интерфейс и внутренняя реализация инкапсулированного программного объекта. Характерные признаки эффективной инкапсуляции: абстракция, общедоступный интерфейс и сокрытие реализации.

(в **1-й части** сделан акцент на *практических* вопросах связанных с понятием «Инкапсуляция». Разделы 1, 2, 3 и 4 дополняют друг друга, частично повторяют. Рассмотрены C#-программы (листинги 1..13), иллюстрирующие характерные признаки эффективной инкапсуляции)

КРАТКО

ИНКАПСУЛЯЦИЕЙ называется процесс объединения некоторых взаимосвязанных (на концептуальном уровне) элементов. В результате получается КЛАСС.
На основе класса может быть создано множество ОБЪЕКТОВ – ЭКЗЕМПЛЯРОВ КЛАССА. Каждый объект имеет свои собственный ДАННЫЕ. Для их обработки используются МЕТОДЫ, которые объект получил от своего класса. При этом объект остается независимым от других объектов.
Объект сам решает, когда используются методы. (см. Материалы к Практ. зан. 1, рис. 1.3 на с. 9).

Уважаемые студенты!

Основная цель, которую необходимо достигнуть в результате изучения дисциплины Объектно-ориентированный анализ и программирование – научиться разрабатывать компьютерные модели реальных и концептуальных систем соответствующих направлению Бизнес-информатика.

Необходимым условием усвоения дисциплины является ВАША самостоятельная работа

Советую Вам все материалы, подготовленные мной к лекциям и практическим занятиям, распечатать и прорабатывать их! Приведенные C#-программы реализовать в среде MS VS .NET 2005 и разобраться в них.

// **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. Инкапсуляция

Одним из основных преимуществ объектно-ориентированного подхода является возможность скрывать некоторые атрибуты и поведение одного объекта от других объектов. В хорошо разработанной программе внешние по отношению к объекту структуры имеют доступ только к необходимому для взаимодействия с ним интерфейсу. Доступ к остальным элементам объекта, не относящимся к его использованию, для этих структур закрыт. **Такой подход называется *инкапсуляцией***. К примеру, объект **Square** (**Квадрат**), вычисляющий квадрат некоторого числа, должен предусматривать интерфейс для возвращения значения полученного результата. Однако посылающему запрос объекту не обязательно иметь доступ к внутренним атрибутам и алгоритмам объекта **Square**, использующимся при расчете квадрата (см. далее Листинг 1). Грамотно разработанные классы созданы с применением принципов инкапсуляции. Далее рассмотрим лежащую в основе инкапсуляции концепцию отделения **интерфейса от реализации**.

1.1. Интерфейс

Интерфейс является основным **средством связи между объектами**. Каждый класс определяет интерфейс, позволяющий создавать объекты этого класса и обеспечивать их правильное функционирование. Любое выполняемое объектом действие должно вызываться сообщением, использующим один из предоставленных интерфейсов. Интерфейс должен полностью описывать правила взаимодействия класса с его пользователями. В языке **C#** для объявления методов, которые являются частью интерфейса, используется ключевое слово **public**.

Скрытые данные	Согласно принципам объектно-ориентированного проектирования **все атрибуты объекта** должны быть объявлены с использованием модификатора доступа **private**. Таким образом, атрибуты не являются частью интерфейса. В интерфейс класса входят **только методы**, объявленные как **public**. Объявление **атрибутов** как **public** противоречит концепции затенения данных.

Вернемся к вышеупомянутому примеру о вычислении квадрата заданного числа. В этом случае интерфейс будет включать в себя две части:

- описание способа создания объекта на основе класса **Square**;
- описание способов: сообщения числа объекту и возвращения квадрата этого числа в ответ.

Обычно интерфейс класса не включает в себя атрибуты (то есть данные) объекта - **только** его **методы**. Как отмечалось выше, если пользователю необходимо получить доступ к некоторому атрибуту, в *рамках объекта* должен быть создан метод, позволяющий возвращать значение этого атрибута (метод **возврата** значения). Когда пользователю понадобится соответствующая информация, то вызванный метод вернет требуемое значение. Таким образом, доступ к атрибуту контролируется содержащим его объектом. Контроль над доступом к атрибутам объекта чрезвычайно важен, особенно с точки зрения тестирования и последующего сопровождения программы. Если вы контролируете доступ к атрибуту, то в случае возникновения проблемы вам не придется отслеживать каждый фрагмент кода, который мог бы изменить значение этого атрибута, поскольку существует только один способ его изменения (с использованием метода **присвоения** значения).

Интерфейсы классов	Помимо интерфейсов классов существуют также интерфейсы **методов**. Интерфейс класса – это **public-методы, доступ к ним открыт для объектов**
Интерфейсы методов	Интерфейсы методов описывают **способ вызова последних**. Другими словами, **интерфейс метода – это его заголовок (сигнатура - первая строка)**

1.2. Реализация

Интерфейс составляют только те атрибуты и методы, доступ к которым **открыт** (**public**) **для других объектов**. Пользователь не должен иметь доступ к реализации, взаимодействуя с классом исключительно через его интерфейс. В рассмотреном ранее классе **Item** (см. листинг **5**, Материалы к **1**-й лекциия. Часть **1**, с. **8**) скрытыми (**private**) были только атрибуты (**строки 12…16**), а методы были открыты (**public**). Во многих случаях **скрывать** необходимо также и некоторые **методы**, которые, таким образом, не будут являться частью интерфейса. Вернемся к примеру о вычислении квадрата заданного числа (**см. выше разделы 1 и 1.1**). До тех пор пока возвращаемый результат остается верным, пользователя не интересует конкретный способ расчета его значения. Таким образом, раздел реализации объекта может меняться, *но вносимые в него изменения не повлияют на код пользователя*.

1.3. Пример применения концепции отделения интерфейса от реализации *в реальном мире*

На рис. 1 приведена иллюстрация принципа отделения интерфейса от реализации, где вместо программного кода используются объекты реального мира. Очевидно, что для работы тостера необходима электроэнергия. Для подключения тостера к сети нужно воткнуть его шнур в электрическую розетку, которая в данном случае является интерфейсом. Все, что требуется сделать для обеспечения тостера электричеством, - использовать шнур, который соответствует спецификациям электрической розетки; электрическая розетка представляет собой интерфейс между тостером и электроэнергией. Тот факт, что в действительности электричество вырабатывается угольной электростанцией, никак не влияет на работу тостера. Фактически, с тем же успехом электричество может вырабатываться атомной электростанцией или местным генератором энергии. Как показывает рис. 1, подобная модель позволяет любому устройству получать электроэнергию до тех пор, пока оно соответствует спецификациям соответствующего интерфейса.

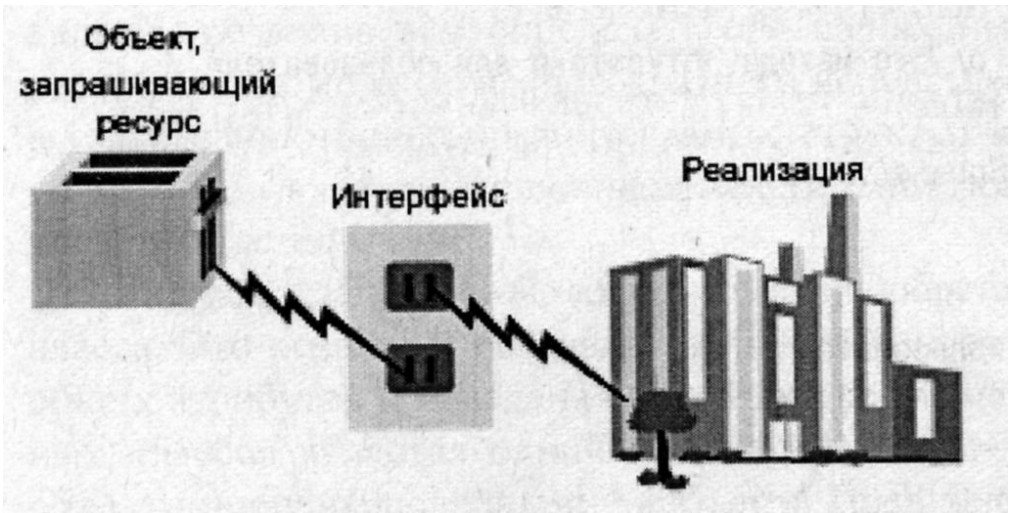


Рис. 1. Пример с электростанцией

1.4. Пример применения концепции отделения интерфейса от реализации

Продолжим изучение класса **Square**. Предположим, вы разрабатываете класс, вычисляющий квадраты целых чисел. Вам необходимо отделить интерфейс от реализации. Это значит, что вы должны предоставить пользователю возможность вызывать соответствующий метод и получать значение квадрата заданного числа. Вы должны также разработать реализацию этого метода, в рамках которой будет вычисляться квадрат; но при этом **пользователь ничего не должен знать о *конкретном способе* вычислений**. Один из возможных вариантов решения подобной задачи

представлен на рис. 2. Отметим, что в диаграмме классов знак плюса (**+**) соответствует слову **public**, а знак минуса (**-**) – слову **private**. Таким образом, вы можете определять входящие в интерфейс методы по знаку плюса перед их именами.

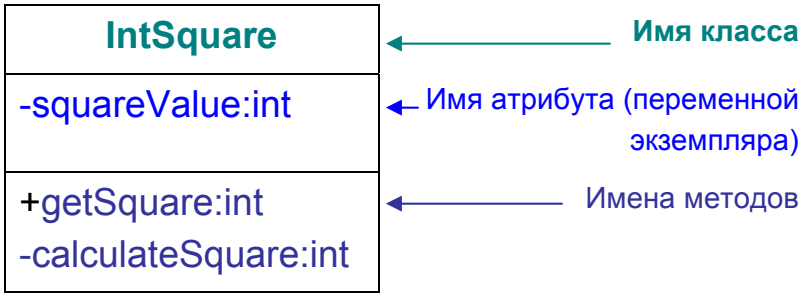
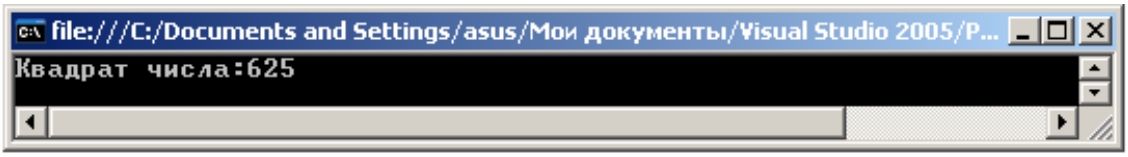


Рис. 2. Диаграмма класса **IntSquare** (КвадратЦелого)

Диаграмма класса **IntSquare** соответствует следующему программному коду:

```
1  using System;                    // Листинг 1. Инкапсуляция: релизация и интерфейс
2
3  namespace ConsAppl_Weisf_c32_IntSquare
4  {
5     public class IntSquare                ////////////////////////////// класс IntSquare ////////////////////////
6     {
7        private int squareValue;                                        // скрытый атрибут
8           // интерфейс класса, состоящит из одного метода, открытого для пользователя
9        public int getSquare(int value)
10       {
11          squareValue = calculateSquare(value);
12          return squareValue;
13       }
14
15       private int calculateSquare(int value)     // реализация, скрытая от пользователя
16    { /* другая реализация метода  –  Math.Exp(2 * Math.Log(value)); эта реализация с существенно
      большими возможностями: любое число возводится в любую степень.
      Необходимо везде заменить int на double* /
17          return value * value;
18       }
19    }
20    public class IntSquareTest
21    {
22       public static void Main(String [ ] args)
23       {
24          IntSquare intSquare = new IntSquare();
25          Console.WriteLine("Квадрат числа:" + intSquare.getSquare(25));
26          Console.ReadLine();
27       }
28    }
29 }
```



Заметьте, что единственной частью класса, к которой пользователь имеет доступ, является метод getSquare (получитьКвадрат, **строки 9…12**), который и представляет собой интерфейс. Реализация алгоритма вычисления квадрата содержится в скрытом методе calculateSquare (вычислитьКвадрат, **строки 15…18**). Отметьте, что *атрибут* squareValue (квадратЧисла, **строка 7**) также является скрытым, поскольку пользователям необязательно знать о его существовании. Таким образом, мы сделали раздел реализации недоступным для других объектов. Объект позволяет пользователю получать доступ только к необходимому для взаимодействия с ним интерфейсу, в то время как не имеющие отношения к использованию объекта подробности скрыты от других объектов.

Если вам понадобится внести изменения в раздел реализации - например, вы решите использовать встроенную в язык C# функцию вычисления квадрата числа, - вам не придется менять интерфейс класса. Пользователь сможет получать результат прежним способом, но реализация вычислений будет другой. Такой подход очень важен при разработке кода, имеющего дело с данными; например, вы можете переносить информацию из файла в базу данных, не заставляя при этом пользователя менять код какого-либо приложения.

2. Инкапсуляция

Есть у вас ручки, бумага, записная книжка и ноутбук в вашей сумке, которую вы берете на занятия? Если так, вы используете инкапсуляцию (encapsulation).

Инкапсуляция — это процесс соединения сходных вещей для формирования нового объекта. Инкапсуляция произвела революцию в способах написания программ и стала краеугольным камнем объектно-ориентированного проектирования.

2.1. Части программы объединяются

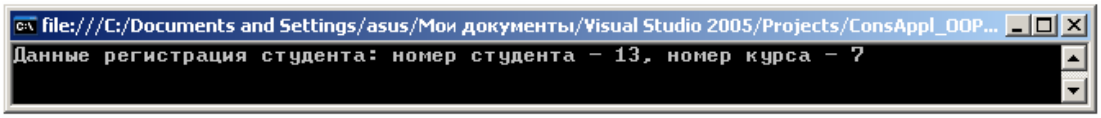
Как известно метод — это определение поведения объекта. Например, для студента существует процедура регистрации на курс. Студент — это объект, а его регистрация на курс — это поведение, осуществляемое студентом. Атрибуты студента, например его идентификационный номер, используются для выполнения метода.

В *реальном* мире объекты и их поведение сгруппированы. Например, вы не можете зарегистрироваться на курс, если вы не студент. Для того чтобы кто-то, не являющийся студентом, не смог получить идентификационный номер студента и зарегистрироваться на курс, предпринимаются некоторые действия. Это происходит, потому что студент и поведение студента объединены вместе и ассоциированы со студентом. Если вы не студент, вы не можете осуществлять поведение студента.

Однако в мире *процедурного* программирования методы и атрибуты не объединены вместе и не ассоциированы с объектами. Это означает, что программист может вызывать метод регистрации для регистрации человека, который не является студентом.

Эту проблему можно проиллюстрировать следующим примером. Это программа на C# в которой определен метод **Registration()**. Метод **Registration()** получает идентификационный номер студента и номер курса в качестве аргументов и выводит эти значения на экран. Список аргументов содержит информацию, необходимую методу для выполнения ее задач.

```
1  using System;                    // Листинг 2 . Иллюстрация – как не следует использовать C#
2  /* Это плохая процедурно-ориентированная программа, так как каждый программист должен
   помнить, что методу Registration можно передавать информацию только о студентах*/
3  namespace ConsAppl_OOP_Keo_c39
4  {
5    class Procedure_Orieted_Programming
6    {
7      static void Registration(string studentID, string courseNumber)
8      {
9    Console.WriteLine("Данные регист-ции студ-та: № студента - " + studentID + ", № курса - " + courseNumber);
10     }
11     static void Main()
12     {
13     string studentID = "13", courseNumber = "7";
14     Registration(studentID, courseNumber);
15     Console.ReadLine();
16     }
17   }
18 }
```



В функции **Main()** определены две переменные (два атрибута) — **studentID** и **courseNumber**. Каждая из них инициализируется значением, которое передается методу **Registration()** в следующем операторе.

Заметьте, что между переменными и методом **Registration()** нет никакой связи за исключением того, что имена переменных и название функции говорят о том, что они делают что-то со студентами.

Отсутствие связи между атрибутами и методами — это недостаток процедурных языков программирования. Это не имеет большого значения, когда один программист разрабатывает все приложение, потому что он знает, что методу **Registration()** не надо передавать переменную, содержащую не студента. Проблемы появляются, когда команда программистов работает над приложением, так как каждый программист в команде *должен помнить*, что метод **Registration()** можно передавать информацию *только о студентах*.

Таким образом, и на прекрасном объектно-ориентированном языке C# можно написать плохую процедурно-ориентированную программу.

Объектно-ориентированные языки программирования позволяют программистам *инкапсулировать атрибуты и методы и связывать их с объектами, что соответствует реальному миру*. Это значительно уменьшает возможность неправильного использования атрибутов и методов.

2.2. Что такое инкапсуляция

Инкапсуляция — это способ связывания атрибутов и методов для формирования объектов. Единственный способ получить доступ к атрибутам и методам объекта состоит в создании экземпляра объекта.

Объекты в программе создаются с помощью определения класса.

На рис. 3 показана диаграмма класса, определяющего объект **Student**. Вверху диаграммы находится список атрибутов, которые описывают студента. Это идентификационный номер студента (**ID**), имя студента **(First name, Last name)** и индикатор того, закончил ли студент обучение **(Graduation)**. Внизу диаграммы находится список методов, связанных со студентом. Они записывают [**Write()**] и отображают [**Display()**] информацию о студенте. Говорят, что эти атрибуты и методы, описанные на рис. 3, инкапсулированы в класс **Student**.

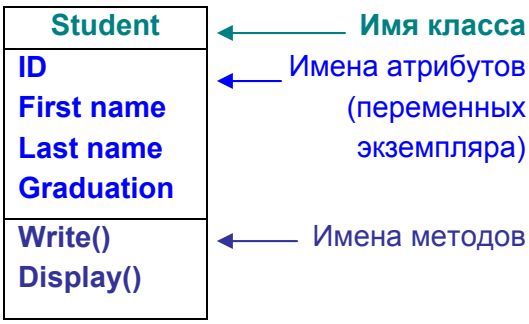


Рис. 3. Диаграмма класса показывает, какие атрибуты и методы инкапсулированы в определении класса **Student**

2.3. Зачем использовать инкапсуляцию

Основная цель использования инкапсуляции — защита.

Инкапсуляция *позволяет* программисту: **1)** после создания в классе атрибутов и методов, **2)** определить в классе правила доступа к ним.

2.4. Защита с использованием спецификаторов (модификаторов) доступа

Программисты контролируют доступ к атрибутам и методам класса с помощью спецификаторов доступа в определении класса. Спецификатор доступа (**access specifier**) — это ключевое слово языка программирования, которое говорит компьютеру, какая часть программы может получить доступ к атрибутам и методам, являющимся членами класса.

Далее рассмотрим следующие спецификаторы доступа — **public**, **private** и **protected**. Спецификатор доступа **public** определяет атрибуты и методы, которые доступны при использовании экземпляра класса. Спецификатор доступа **private** определяет атрибуты и методы, которые доступны только методам, определенным в классе. Спецификатор доступа **protected** определяет атрибуты и методы, которые могут быть наследованы другими классами (производные классы – см. Материалы к Практ. зан. № 6, с. 12,сл). В языке C# есть еще два спецификатора доступа: **internal** (см. Материалы к Практич. занятию №4, **стр. 33, 34**) **и protected internal** (– №6, **стр. 15**).

2.4.1. Спецификатор доступа public

Когда объявляется экземпляр класса можно использовать этот экземпляр для доступа к атрибутам и методам, определенным в части класса со спецификатором доступа **public**. Вы определяете часть класса со спецификатором доступа **public** с помощью ключевого слова **public**, как показано в следующем примере.

Метод **Display()** объявлен с помощью спецификатора доступа **public**. Это означает, что он может быть вызвана напрямую из любой части программы с помощью объявления экземпляра класса **Student**. Вот пример:

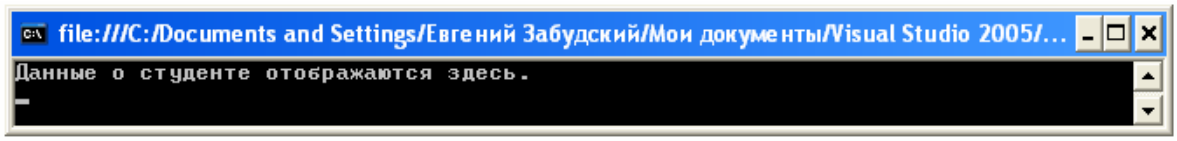
```
5   class Student
6   {
7   public void Display()
8   {
9   // Поместите здесь операторы
10  }
…   ….
```

Можно напрямую обращаться к атрибутам и методам, определенным с ключевым словом **public**, используя в программе имя экземпляра, оператор «**точка**» и название атрибута или метода, к которым надо получить доступ (см. ниже строку **17**).

Следующая программа иллюстрирует использование спецификатора доступа **public** для отображения информации о студенте.

```
1   using System;                    // Листинг 3. Спецификатор доступа public
2
3   namespace ConsAppl_OOP_Keo_c42
4   {
5     class Student
```

```
6    {
7        public void Display()
8        {
9         Console.WriteLine("{0}", "Данные о студенте отображаются здесь.");
10       }
11   }
12    class StudentTest
13    {
14       public static void Main()
15       {                          // myStudent – объект (экземпляр) класса Student
16       Student myStudent = new Student();
17        myStudent.Display();        // для объекта myStudent вызывается метод
18       Console.ReadLine();
19       }
20    }
21  }
```



Первый оператор в методе **Main()** объявляет экземпляр класса **Student** (см. строку **16**). Второй оператор вызывает метод **Display()** класса **Student** (строка **17**).

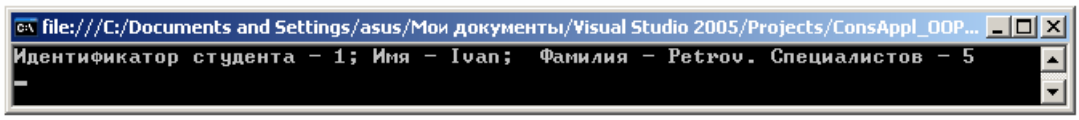
2.4.2. Спецификатор доступа private

Спецификатор доступа **private** ограничивает доступ к атрибутам и методам теми методами, которые являются членами того же самого класса. Следующий пример иллюстрирует, как это делается. Цель — предотвратить прямой доступ к идентификатору студента, его имени и статусу окончания обучения при использовании экземпляра класса **Student**. Это осуществляется с помощью спецификатора доступа **private**.

Спецификатор доступа **private** не запрещает методу **Display()** обращаться к этим атрибутам (строки **6, 7**), так как метод **Display()** (строки **9…12**) является членом класса **Student** (строки **4…13**). Для использования других членов класса (атрибуты и методы) из методов этого же класса создавать экземпляр класса не нужно:

```
1   using System;                                  // Листинг 4. Спецификатор доступа private
2   namespace ConsAppl_OOP_Keo_c43
3   {
4     class Student
5     {
6       int m_ID = 1, m_Graduation = 5;                  // здесь не нужны спецификаторы доступа
7       String m_First = "Ivan" , m_Last = "Petrov";      //  - “ - “- “ -
8
9      public void Display()    // сделать эксперимент: заменить public на private: возникнет ошибка
10      {
11  Console.WriteLine("Идентификатор студента - " + m_ID + "; Имя - " + m_First + ";
                        Фамилия - " + m_Last + ". Специалистов - " + m_Graduation);
12      }
```

```
13    }
14    class StudentTest
15    {
16    public static void Main()
17    {
18        Student myStudent = new Student();
19            myStudent.Display();
20            Console.ReadLine();
21        }
22    }
23  }
```



Техника, показанная в этом примере, — краеугольный камень в объектно-ориентированном программировании, потому что она требует использовать метод-член (в Листинге 4 – это метод **Display()**) для доступа к атрибутам класса. Это позволяет программисту, который создал класс, закодировать в методе-члене правила, управляющие доступом к атрибутам.

Представьте, что нужно отобразить информацию о студенте. Но доступа напрямую к информации о студенте нет. Для получения доступа необходимо вызвать метод-член класса, отображающий информацию о студенте. Это дает программисту, который создал класс, полный контроль над тем, к каким атрибутам можно получить доступ и как они будут отображены.

2.4.3. Спецификатор доступа protected

Спецификатор доступа **protected** определяет атрибуты и методы, которые могут быть использованы: ***1)*** только методами, являющимися членами класса, ***2)*** а также методами, которые являются членами производных классов (см. Материалы к Практ. зан. № 6, с. 12,сл).

Класс, от которого производится наследование, называется *базовым*, а класс, который наследуется от базового класса, *производным* классом.

Наследование рассмотрим на следующей (3-й) лекции, здесь же рассмотрим его поверхностно, чтобы можно было понять, как работает спецификатор доступа **protected**. Предположим, что у нас есть два класса. Один класс называется **Student**, а другой — **GradStudent.**

Класс **Student** содержит атрибуты и варианты поведения, которые характерны для всех студентов. Класс **GradStudent** содержит атрибуты и варианты поведения, которые уникальны для *аспирантов* (рис. 4), а также включает все атрибуты и варианты поведения *студентов*. Аспирант является студентом.

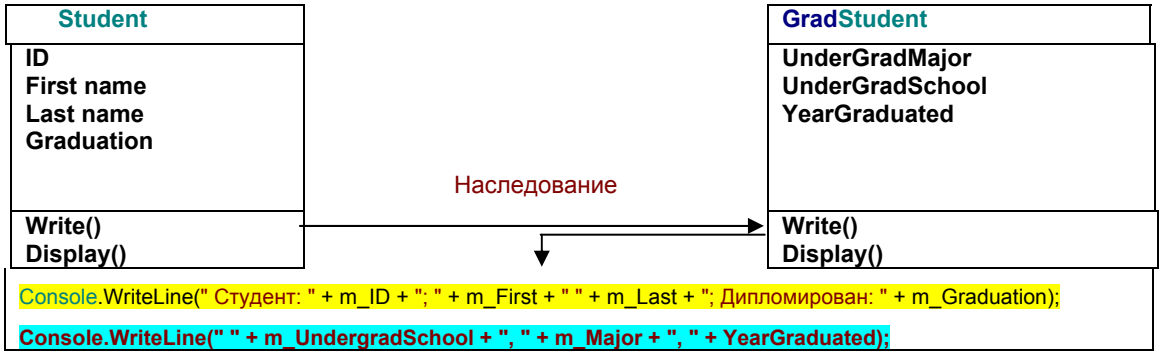


Рис. 4. Класс **GradStudent** содержит атрибуты и варианты поведения студентов, а также уникальные для аспирантов атрибуты и варианты поведения

Вместо того чтобы дублировать атрибуты и варианты поведения класса **Student** в классе **GradStudent**, можно сделать так, чтобы класс **GradStudent** наследовал все или некоторые атрибуты и варианты поведения класса **Student**.

Атрибуты и варианты поведения, определенные с помощью спецификаторов доступа **public** и **protected**, можно напрямую использовать в классе **GradStudent**.

Следущая программа (листинг 5) на C# показывает, как использовать спецификатор доступа **protected**. В ней объявлены три класса. Первые два определения класса такие же, как и в примере со спецификатором доступа **private**. Третье определение класса — новое. Это определение аспиранта, называется этот класс **GradStudent**. Класс **GradStudent** наследует атрибуты и методы-члены класса **Student**, что заложено в строке **18** кода (см. ниже).

Листинг 5. Использование спецификатора доступа **protected** в C#

```
1  using System;                                                    //  Спецификатор доступа protected в действии
2  namespace ConsAppl_OOP_Keo_c51_52_konstrukt
3  {
4     class Student                                                 // базовый класс Student
5        {
6           // значения этих перем-х поступают из класса GradStudent, так как их спец-р доступности protected;
7           protected int m_ID, m_Graduation;
8                                 // Сделать эксперимент: если protected заменить на private, то возникнет ошибка
9           protected String m_First;
10          protected String m_Last;
11
12          public void Display()
13          {                                                       // 4.  Печать 1-й строки (строка 14)
14    Console.WriteLine(" Студент: " + m_ID + "; " + m_First + " " + m_Last + "; Дипломирован: " + m_Graduation);
15          }
16       }                                  ////////////////////////// конец базового класса Student //////////////////////
17
18       class GradStudent : Student                  ////////// производный класс GradStudent ////////////////////
19       {
20          private int YearGraduated;
21          private String m_UndergradSchool;
22          private String m_Major;
23                                // Конструктор GradStudent() класса GradStudent
24          public GradStudent(int ID, int Grad, String Fname, String Lname, int yrGrad, String unSch, String major)
25          {
26             m_ID = ID;                                  // идентификатор
27             m_Graduation = Grad;                        // индикатор окончания вуза
28             YearGraduated = yrGrad;                     // год окончания вуза
29             m_First = Fname;                            // Имя
30             m_Last = Lname;                             // Фамилия
41             m_UndergradSchool = unSch;                  // название вуза
42             m_Major = major;                            // специализация
43          }
44
45          public void GradDisplay()
```

```
46          {
47
48  base.Display();          // 3. Обращение к м-ду Display() базового класса Student     (строки 12…15)
49      Console.WriteLine(" " + m_UndergradSchool + ", " + m_Major + ", " + YearGraduated);
                                                    // 5.  Печать 2-й строки результата (строка 49)
50          }
51      }                                  ////////////////// конец производного класса GradStudent ////////////////////
52
53      class StudentTest                                  ////////////////////// класс StudentTest ////////////////////////
54      {
55          public static void Main(String [ ] args)
56          {       // 1.      Обращение к конструктору GradStudent() и инициализация (строки 24…43)
57      GradStudent myStudent = new GradStudent(10, 1, "Иван", "Петров", 2004, "ГУ-ВШЭ", "Computer Science");
58  myStudent.GradDisplay();      // 2. Обращение к м-ду GradDisplay() класса GradStudent (строки 45…50)
59              Console.ReadLine();
60          }
61      }                                              ////////////////////// конец класса StudentTest ////////////////////////
62  }
```

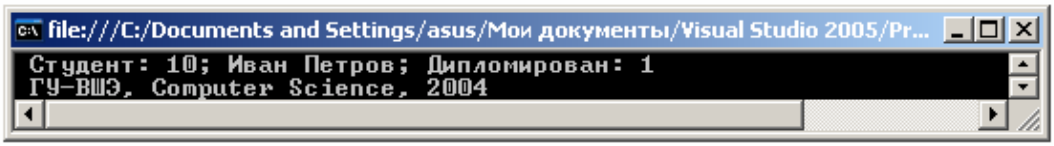
Из строки **18**

18: **class GradStudent : Student**

следует, что класс **GradStudent** является производным от базового класса **Student**. На это указывает следущая конструкция в этой строке – **“ : Student “**. Таким образом, класс **GradStudent** наследует атрибуты **(строки 7…10)** и метод-член **Display() (строки 12…15)** класса **Student.** Для обращения к методу **Display()** в строке **48** использовано ключевое слово “ **base** ” и оператор уточнения “ **.** ” .

48: **base.Display();**

Последовательность работы программы – см. выше пп. 1, 2, 3, 4 и 5, а также пояснение далее.



Определение класса **Student** практически идентично определению этого класса в примере со спецификатором доступа **private** (см. в листинге 4 **строки 6, 7**), за одним исключением. Обратите внимание, что перед названиями атрибутов стоит ключевое слово **protected** (см. в листинге **5 строки 7…10**), которое сообщает компьютеру, что к этим атрибутам можно обращаться из методов-членов класса **GradStudent**.

Экземпляр **myStudent** класса **GradStudent** объявляется в методе **Main()** класса **StudentTest** (**см. строку 57**), а затем используется для присвоения значений атрибутам класса **GradStudent** и атрибутам, унаследованным от класса **Student** (**см. строки 24…43**). Затем эти значения выводятся на экран с помощью вызова метода **GradDisplay()** класса **GradStudent** (**см. строку 58**).

Обратите внимание, что метод **GradDisplay()** класса **GradStudent** (**см. строки 45…50**) немного отличается от метода **Display()** класса **Student** (**см. строки 12…15**) . Рассмотрим это подробнее и начнем с метода **Display()** класса **Student**.

Метод **Display()** класса **Student** отображает значения атрибутов, определенных в классе

Student. Метод **GradDisplay()** класса **GradStudent** расширяет возможности метода **Display()** класса **Student** — он отображает атрибуты и класса **Student**, и класса **GradStudent**. Вот как это делается: вспомните, что класс **GradStudent** может наследовать не только **public**-, но и **protected**-члены класса **Student**. Это означает, что метод **GradDisplay()** класса **GradStudent** может вызвать метод **Display()** класса **Student** (см. **строку 48**). Класс, от которого производится наследование, называется базовым, для доступа к его атрибутам и методам-членам используется ключевое слово **base**.

В этой программе оператор **base.Display()** сообщает компьютеру, что необходимо вызвать метод **Display()** класса **Student**, который выводит атрибуты класса **Student** на экран (1-я строка вывода на экран). Следующий оператор (см. **строку 49**) отображает атрибуты класса **GradStudent** на 2-й строке вывода.

3. Столпы объектно-ориентированного программирования

В любом объектно-ориентированном языке программирования обязательно реализованы три важнейших принципа — «столпа» объектно-ориентированного программирования:

- инкапсуляция: как объекты прячут свое внутреннее устройство;
- наследование: как в этом языке поддерживается повторное использование кода;
- полиморфизм: как в этом языке реализована поддержка выполнения нужного действия в зависимости от типа передаваемого объекта?

Конечно же, все три этих принципа реализованы и в C#. Однако перед тем как начать рассматривать синтаксические особенности реализации этих принципов, постараемся убедиться, что в них не осталось неясностей.

3.1. Инкапсуляция

Первый «столп» объектно-ориентированного программирования — это инкапсуляция. Так называется способность прятать детали реализации объектов от пользователей этих объектов. Например, предположим, что создан класс с именем **DBReader** (для работы с базой данных), в котором определено два главных метода: **Open()** и **Close()**.

```
01  /* Класс DBReader скрывает за счет инкапсуляции подробности открытия и закрытия баз данных */
02  DBReader f = new DBReader();
03  f.Open(@"C:\foo.mdf");                    // открываем базу foo.mdf
04                                            // Выполняем с базой данных нужные нам действия
05  f.Close();                                // закрываем базу foo.mdf
```

Вымышленный класс **DBReader** инкапсулирует внутренние подробности того, как именно он обнаруживает, загружает, выполняет операции и в конце концов закрывает файл данных. За счет инкапсуляции программирование становится проще: вам нет необходимости беспокоиться об огромном количестве строк кода, которые выполняют свою задачу скрыто от вас. Все, что от вас требуется — создать экземпляр [**f**] нужного класса [class **DBReader**] и передать ему необходимые сообщения (типа «**открыть файл с именем foo.mdf**», то есть **f.Open(@"C:\foo.mdf");**).

С философией инкапсуляции тесно связан еще один принцип — сокрытия всех внутренних данных (то есть переменных-членов) класса. В идеале все внутренние переменные члены класса должны быть определены как **private**. В результате обращение к ним из внешнего мира будет возможно только при помощи *открытых методов-членов* **public**. Такое решение, помимо всего прочего, защищает от возможных ошибок, поскольку *открытые данные* **public**) очень просто повредить.

3.2. Средства инкапсуляции в C#

Принцип инкапсуляции предполагает, что ко внутренним данным **private** объекта (переменным-членам) нельзя обратиться напрямую через экземпляр этого объекта. Вместо этого для получения информации о внутреннем состоянии объекта и внесения изменений необходимо использовать специальные методы. В **C#** инкапсуляция реализуется на уровне синтаксиса при помощи ключевых слов **public**, **private** и **protected**. В качестве примера мы рассмотрим следующее определение класса:

```
01  // Класс с единственным полем
02  public class BOOK
```

```
03  {
04  public int numberOfPages;  // поле
05  …
06  }
```

Термин «поле» (**field**) используется для открытых данных класса — переменных, объявленных с ключевым словом **public**. При использовании полей в приложении возникает проблема: полю можно присвоить любое значение, а организовать проверку этого значения бизнес-логике вашего приложения достаточно сложно. Например, для открытой переменной **numberOfPages** используется тип данных **int**. Максимальное значение для этого типа данных — это достаточно большое число (2^{32} = 2 147 483 647). Если в программе будет существовать такой код, проблем со стороны компилятора не возникнет:

```
01  // Задумаемся...
02  public static void Main()
03  {
04  Book miniNovel = new Book();
05  miniNovel.numberOfPages = 30 000 000;
06  }
```

Тип данных **int** вполне позволяет указать для книги небольших размеров количество страниц, равное **30 000 000**. Однако понятно, что книг такой величины не бывает, и во избежание дальнейших проблем желательно использовать какой-нибудь механизм проверки, который отсеивал бы явно нереальные значения (например, он пропускал бы только значения между **1** и **2000**). Применение поля — открытой переменной не дает нам возможности простым способом реализовать подобный механизм. Поэтому поля в реальных рабочих приложениях используются нечасто.

Следование принципу инкапсуляции позволяет защитить внутренние данные класса от неумышленного повреждения. Для этого достаточно все внутренние данные сделать закрытыми (объявив внутренние переменные с использованием ключевых слов **private** или **protected**). Для обращения к внутренним данным можно использовать один из двух способов:

- создать традиционную пару методов : *а)* для получения информации (**accessor**),
  б) для внесения изменений (**mutator**);
- определить именованное **свойство**.

Еще один метод защиты данных, предлагаемый **C#**, — использовать ключевое слово **readonly**. Однако какой бы способ не был выбран, общий принцип остается тем же самым — *инкапсулированный класс должен прятать детали своей реализации от внешнего мира*. Такой подход часто называется «программированием по методу черного ящика». Еще одно преимущество такого подхода заключается в том, что можно как угодно совершенствовать внутреннюю реализацию класса, полностью изменяя его содержимое. Единственное, о чем вам придется позаботиться, — чтобы в новой реализации остались методы с той же сигнатурой и функциональностью, что и в предыдущих версиях. В этом случае не придется менять ни строчки существующего кода за пределами данного класса.

3.2.1. Первый способ инкапсуляции: при помощи традиционных методов доступа и изменения

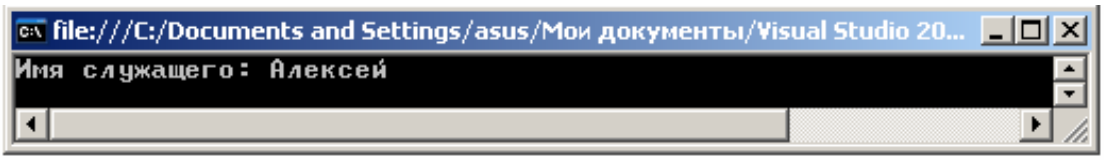
Рассмотрим класс **Employee** (Служащий). Если необходимо, чтобы внешний мир смог работать с внутренними данными **private** этого класса (пусть это будет только одна переменная **fullName**,

для которой используется тип данных **string**), то традиционный подход рекомендует создание метода доступа (**accessor**, или **get method**) и метода изменения (**mutator**, или **set method**). Набор таких методов может выглядеть следующим образом (Листинг 6):

```
01  /*Определение традиционных методов доступа и изменения для закрытой переменной private
                                                    (см. ConsAppl_Troel_c151_инкапсл) */
02  class Employee
03  {
04  private string fullName;                    // внутренняя (private) переменная
05                                              // Метод доступа (accessor)
06  public string GetFullName()
07  {
08  return fullName;
09   }
10                                              // Метод изменения (mutator)
11  public void SetFullName(string n)
12  {
13                          // Логика для удаления неположенных символов (!. @. #, $. % и прочих
14                                              // Логика для проверки максимальной длины и прочего
15  fullName = n;
16  }
17  }
```

Такой подход требует наличия двух методов для взаимодействия с каждой из переменных. Вызов этих методов может выглядеть следующим образом (Листинг 6):

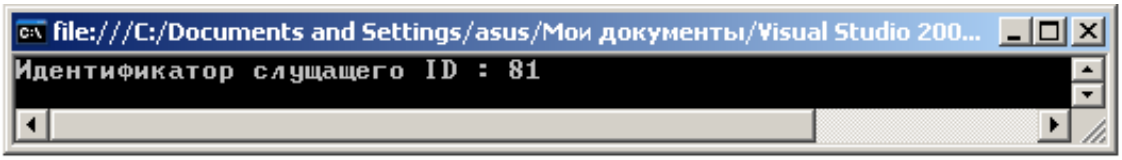
```
18  // Применение методов доступа и изменения                    (см. ConsAppl_Troel_c151_инкапсл)
19   public static void Main(string [ ] args)
20  {
21  Employee p = new Employee();
22  p.SetFullName("Алексей");
23  Console.WriteLine("Имя служащего: " + p.GetFullName());
24
25          // Ошибка! К закрытым данным нельзя обращаться напрямую через экземпляр объекта!
26  // p.FullName;  // в этой строке ошибка: сделать эксперимент – убрать «//» и запуск программы
27  }
```



3.2.2. Второй способ инкапсуляции: применение свойств класса

Помимо традиционных методов доступа и изменения для обращения к закрытым private членам класса можно также использовать свойства (**properties**). Свойства позволяют имитировать доступ к внутренним (**private**) данным класса: при получении информации или внесении изменений через свойство синтаксис выглядит так же, как при обращении к обычной открытой (public) переменной. Но на самом деле любое свойство состоит из двух скрытых внутренних методов. Преимущество свойств заключается в том, что вместо того, чтобы использовать два отдельных метода, пользователь класса может использовать единственное свойство, работая с ним так же, как и с открытой переменной-членом данного класса (см. Материалы к Практ. Зан. №2, дополнение 1, стр. 12…19) (Листинг 7):

```
01  // Обращение к имени сотрудника через свойство   (см. ConsAppl_Troel_c152_инкапсл)
02  public static void Main(string [ ] args)
03  {
04  Employee p = new Employee();
05  p.EmpID = 81;                                    // Устанавливаем значение (set)
06
07  Console.WriteLine("Идентификатор сотрудника: {0}", p.EmpID); // Получаем значение (get)
08  Console.ReadLine();
09  }
```



Если заглянуть внутрь определения класса, то свойства всегда отображаются в «реальные» методы доступа и изменения. А уже в определении этих методов вы можете реализовать любую логику (например, для устранения лишних символов, проверки допустимости вводимых числовых значений и прочего). Ниже представлен синтаксис класса **Employee** с определением свойства **EmpID** (Листинг 7):

```
01  // Пользовательское свойство EmpID для доступа к переменной empID
02  public class Employee          ////////////////////// начало класса Employee ////////////////
03  {                                                    //(см. ConsAppl_Troel_c152_инкапсл)
04  private int empID;                    // empID – закрытая (private) переменная
05                          // Свойство для empID
06  public int EmpID                    ////////////////// начало свойства EmpID ////////////////
07  {
08  get {return empID;}
09  set
10  {
11                      // Здесь вы можете реализовать логику для проверки вводимых
12                                          // значений и выполнения других действий
13  empID = value;
```

```
14  }
15  }                    ////////////////////// конец свойства EmpID // ////////////////
15  }                    ////////////////////// конец класса Employee //////////////////
```

Свойство **C#** состоит из двух блоков — **блока доступа** (**get block**) и **блока изменения** (**set block**). Ключевое слово **value** представляет правую часть выражения при присвоении значения посредством свойства. Как и все в **C#**, то, что представлено словом **value** — это также объект. Совместимость того, что передается свойству как **value**, с самим свойством, зависит от определения свойства. Например, свойство **EmpID** предназначено (согласно своему определению в классе) для работы с закрытым целочисленным **empID**, поэтому число **81** вполне его устроит:

```
// В данном случае типом данных, используемым для value, будет
int p.EmpID = 81;
```

Показать дополнительные возможности применения ключевого слова **value** можно на таком примере (**GetType() и ToString() –** см. Материалы к Практ. Зан. №7, раздел 5, стр. 22…27):

```
01  public int EmpID                                    // Свойство для empID
                                                        (Листинг 8)
02      {                                   //(см. ConsAppl_Troel_c153_инкапсул)
03          get { return empID; }
04          set
05          {
06                                          // Как еще можно использовать value
07      Console.WriteLine("value - указывает тип: {0}", value.GetType());
08              Console.WriteLine("value -  строка: {0}", value.ToString());
09          empID = value;
10          }
11      }
```

Результат работы данной программы представлен на рис. 5.

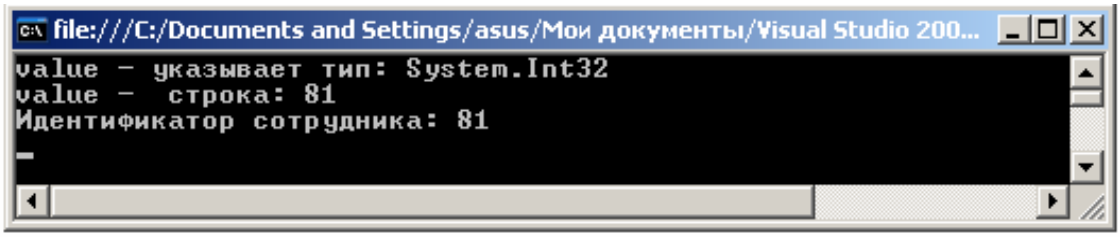


Рис. 5. Значение «**value**» при **EmpID = 81**

Необходимо отметить, что обращаться к объекту **value** можно только в пределах программного блока **set** внутри определения свойства. Попытка обратиться к этому объекту из любого другого места приведет к ошибке компилятора.

3.3. Внутреннее представление свойств C#

Многие программисты стараются использовать для имен традиционных методов доступа и изменения соответственно приставки **get_** и **set_** (например, **get_Name()** и **set_Name()**). Само по себе это не представляет проблемы. Проблему представляет другое: C# для внутреннего представле-

ния свойства использует методы с теми же самыми префиксами. Поэтому подобное определение класса вызовет ошибку компилятора.

3.4. Свойства: только для чтения, только для записи и статические

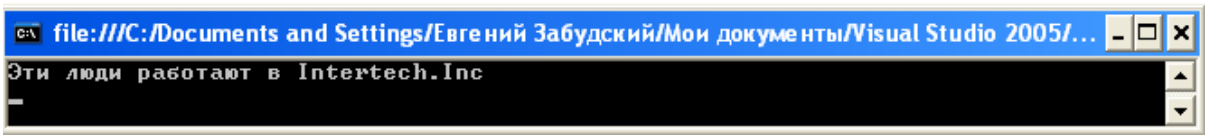
Еще сведения о свойствах классов C#. Как известно, свойство **EmpID** было создано как свойство, доступное: *1)* и для чтения, *2)* и для записи. Однако при создании пользовательских свойств класса часто возникает необходимость создать свойство, которое будет доступно только для чтения. Делается это очень просто: необходимо в определении свойства пропустить блок **set**. В результате свойство станет доступным только для чтения:

```
1   public class Employee
2   /* Будем считать, что исходное значение переменной ssn
       присваивается с помощью конструктора класса */
3   private string ssn;                          // закрытая переменая ssn
4         // А вот так выглядит свойство Ssn только для чтения значения ssn
5   public string Ssn { get { return ssn; } }
…   ….
    }
```

C# также поддерживает статические свойства. Статические переменные предназначены для хранения информации на уровне всего класса, а не его отдельных объектов. Если объявлены статические данные (то есть те же переменные), то обращаться к ним и устанавливать значения должны статические свойства. Предположим, что в классе **Employee** необходимо, помимо всего прочего, хранить еще и информацию об имени организации, в которой работают все сотрудники — объекты класса **Employee**. Для этого будет использована специальная статическая переменная (см. строку **6**). Соответствующее статическое свойство для работы с этой переменной представлено в листинге 9 (см. строки **8…12**). Обращение к статическим свойствам производится так же, как и к статическим методам. В обращении указываются: имя класса, оператор «точка», имя свойства (см. строки **20, 21**).

```
1   using System;          // Со статическими данными должны работать статические свойства
2   namespace ConsAppl_Troel_c155_инкапсул                                  // листинг 9
3   {
4      public class Employee
5      {
6         private static string compName;          // Статическая переменная compName
7
8         public static string Company                    // Статическое свойство Company
9         {
10           get { return compName; }
11           set { compName = value; }
12        }
13        // ...
14     }
15     public class TestEmployee
16     {
17                                    // Задаем и получаем информацию об имени компании
18        public static void Main(string [ ] args)
```

```
19  {    /* Два обращения к свойству Company: объект (экземпляр) класса Employee для
            обращения  к статической переменной создавать не нужно, т.к. в обращении
            указывается имя класса: Employee.Company*/
20          Employee.Company = "Intertech.Inc";     // Задаем информацию на уровне класса (set)
21                                                  // Получаем информацию на уровне класса (get)
            Console.WriteLine("Эти люди работают в {0}", Employee.Company);
22          //...
23          Console.ReadLine();
24      }
25  }
26  }
```



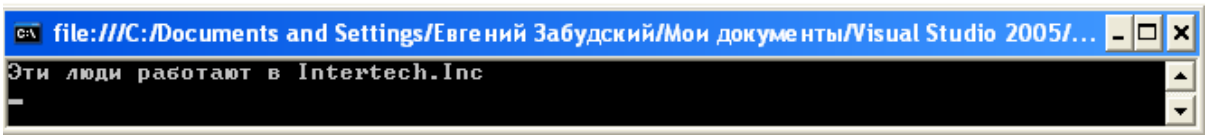
3.5. Статические конструкторы

Само словосочетание «статический конструктор» звучит несколько странно — ведь конструкторы нужны для создания объектов (см. Материалы к Практ. Зан. №2, дополнение 1, стр. **4…12**), а все, что имеет определение «статический», работает на уровне классов, а не отдельных объектов. Однако в C# такие конструкторы вполне имеют право на существование. Единственное **их назначение** — присваивать исходные значения статическим переменным.

С точки зрения синтаксиса статические конструкторы — это достаточно причудливые образования. Например, для них *нельзя* использовать модификаторы области видимости, однако слово static должно присутствовать обязательно (см. строки **9…12**). Вот пример ситуации, в которой может пригодиться статический конструктор: предположим, что необходимо, чтобы статической переменной compName (см. строку **7**) всегда при создании присваивалось значение **Intertech.Inc**. При использовании свойства **Company** (см. строки **13…16**), присваивать ему исходное значение не придется — статический конструктор сделает это автоматически (см. строку **11**). Это можно осуществляется следующим образом (листинг 10):

```
1   using System;                   /* Статические конструкторы используются ТОЛЬКО
                                       для инициализации статических переменных*/
2   namespace ConsAppl_Troel_c156_инкапсул                          //(листинг 10)
3   {
4       public class Employee
5       {
6           //...
7           private static string compName;                 // статическая переменная
8
9           static Employee()   // статический конструктор срабатывает АВТОМАТИЧЕСКИ
10          {
11              compName = "Intertech.Inc";
12          }
13          public static string Company    // Статическое свойство Company только для чтения
14          {
15              get { return compName; }
16          }
```

```
17
18          //...
19      }
20      public class TestEmployee
21  {                   /* Значение ("Intertech.Inc") свойства Company будет АВТОМАТИЧЕСКИ
                                установлено через статический конструктор*/
22          public static void Main(string [ ] args)
23          {
24                                  // Обращение к статическому свойству Company
25              Console.WriteLine(" Эти люди работают в {0}", Employee.Company);
26              Console.ReadLine();
27          }
28      }
29  }
```



Подводя итоги этого раздела, можно отметить, что свойства классов C# используются для тех же самых целей, что и традиционные методы доступа и изменения значений. Главное преимущество свойств заключается в том, что пользователь может работать через них со внутренними (**private**) данными, используя *единственное* имя (вместо двух разных имен методов).

3.6. Псевдоинкапсуляция: создание полей «только для чтения»

Помимо свойств только для чтения, в C# также предусмотрены поля, значения которых *изменять нельзя*. Поля (**fields**) — это открытые (**public**) данные класса. Обычно применение полей в рабочем приложении — не самая лучшая идея, поскольку поля беззащитны — им легко присвоить ошибочное значение и тем самым испортить внутреннее состояние объекта. Однако в C# предусмотрена возможность запретить любую возможность изменять значение поля, объявив его с ключевым словом **readonly**:

```
1   public class Employee
2   {
3   ....
4   // Поле только для чтения (его значение устанавливается конструктором)
5   public readonly string ssnField;
6    }
```

Любая попытка изменить значение такого поля приведет к ошибке компилятора.

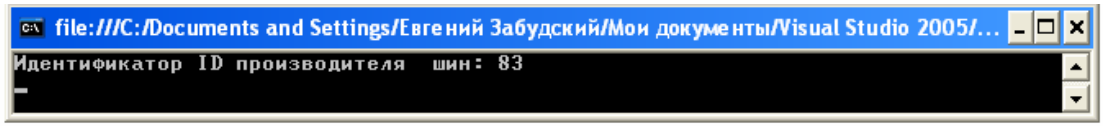
3.7. Статические поля «только для чтения»

Статические поля, определенные как «только для чтения», также вполне имеют право на существование. Обычно они используются в тех ситуациях, когда вы хотите создать некоторое количество постоянных значений, связанных с определенным классом. Очень похожие задачи выполняют обычные константы, которые можно назвать родственниками статических полей только для чтения. Однако между такими полями и константами есть существенное различие: константа заменяется на свое значение уже в процессе компиляции, в то время как значения статических полей только для чтения вычисляются лишь в процессе выполнения программы.

Например, предположим, что есть объект **Car** (автомобиль), который в процессе выполнения

должен создавать объект Tire (шины). Для этой цели можно применить класс Tire, использовав в нем набор статических полей только для чтения (листинг 11):

```
1  using System;  // В классе Tire определен набор статических полей только для чтения
2                                                                    //(листинг 11):
3  namespace ConsAppl_Troel_c157
4  {
5     public class Tire
6     {
7        public static readonly Tire GoodStone = new Tire(90);   //статические поля только для чтения
8        public static readonly Tire FireYear  = new Tire(100);     //  -"-
9        public static readonly Tire ReadyLine = new Tire(43);     //  -"-
10       public static readonly Tire Blimpy    = new Tire(83);      //  -"-
11
12       private int manufactureID;                                  // закрытая переменная
13
14       public int MakeID          // Свойство MakeID для доступа к закрытой переменной
15       {
16          get { return manufactureID; }
17       }
18
19       public Tire(int ID)                              // Конструктор Tire(int ID) класса Tire
20       {
21          manufactureID = ID;
22       }
23    }                    ////////////////////////////////////// конец  класса Tire ////////////////////////////////
24          // Так можно использовать динамически создаваемые поля только для чтения
25    public class Car
26    {
27                                                                  // Какая у меня марка шин?
28       public Tire tireType = Tire.Blimpy;                  // Возвращает новый объект Tire
29    }                     ////////////////////////////////// конец  класса Car ////////////////////////////////////
30    public class CarApp
31    {
32       public static void Main(string[ ] args)
33       {
34          Car c = new Car();
35     //Выводим на консоль идентификатор производителя шин  (в нашем случае - 83)
36 Console.WriteLine("Идентификатор ID производителя  шин: {0}", c.tireType.MakeID);
37          Console.ReadLine();
38       }
39    }                        //////////////////////////// конец  класса CarApp ////////////////////////////////////
40 }
```



4. Инкапсуляция. Управление видимостью элементов типа с помощью модификаторов

Классы языка C# содержат такие элементы, как индексаторы (см. Материалы к Практ. Зан. №2, дополнение 2), методы (см. Материалы к Практ. Зан. №2) и свойства (см. Материалы к Практ. Зан. №2, дополнение 1, стр. 12…19), которые помогают инкапсулировать внутреннее состояние типа. Видимость элементов типа управляется с помощью модификаторов, приведенных в таблице 1. Эти модификаторы управляют открытым интерфейсом типа для внешних типов.

Таблица 1. Модификаторы видимости

Модификатор видимости	Кто может видеть
public (открытый)	Все другие типы
private (закрытый)	Только внутри этого типа
protected (защищенный)	Производные типы
internal (внутренний)	Типы внутри той же программной сборки
protected internal (внутренний защищенный)	Типы внутри той же программной сборки и производные типы

В листингах 12 (клиент) и 13 (компонент) приведены примеры использования модификаторов видимости для управления открытым интерфейсом класса.

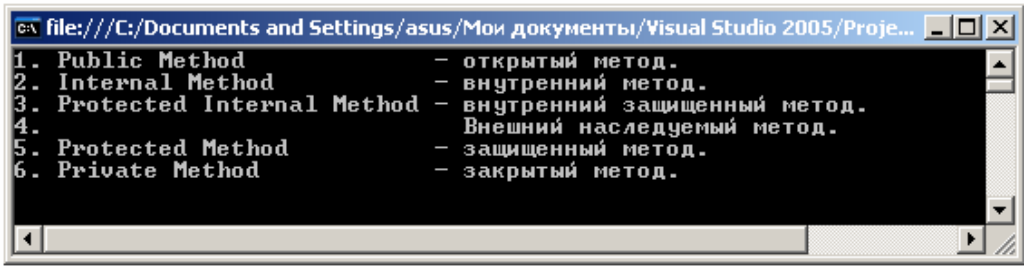
```
01  using System; /* Вызывающий класс для демонстрации модификаторов видимости.  Используется dll-файл ClassLib_Mayo_c80_list3_9.dll – см. листинг 13 */
02  using ClassLib_Mayo_c80_list3_9;
    // присоединение пространства имен dll-файла ClassLib_Mayo_c80_list3_9.dll
03  namespace ConsAppl_Mayo_c80_list_3_8                    // Листинг 12
04  {
05    internal class ExternalDerived : PublicClass   /// базовый класс ExternalDerived
06    {
07      internal void ExternalDerivedMethod()
08      {
09        Console.WriteLine("4.  Внешний наследуемый метод");
10        ProtectedMethod();                    // 6. см. стр 38, 40 л-га 13
11      }
12    }
13
14    class Visibility                ////////////////////////// класс Visibility //////////////////////
15    {
16      static void Main()
17      {
18        PublicClass myPublicClass = new PublicClass();
19        myPublicClass.PublicMethod();                 // 1. см. стр 27, 29 л-га 13
20
21        ExternalDerived myExternalDerived = new ExternalDerived();
```

```
22          myExternalDerived.ExternalDerivedMethod(); // 5. см. стр 07, 09 л-га 12
23
24              Console.ReadLine();
25          }
26      }                          ////////////////////////// окончание класса Visibility //////////////////////
27  }
```

```
01  using System;       /* Вызываемый класс используется для демонстрации модификаторов
                           видимости. Из этого кода сформирован dll-файл – ClassLib_Mayo_c80_list3_9.dll */
02  namespace ClassLib_Mayo_c80_list3_9                                    // Листинг 13
03  {
04      internal class BaseInternal              ////////////// базовый класс BaseInternal //////////
05      {
06          public void BaseInternalMethod()
07          {
08              Console.WriteLine("2. Internal Method - внутренний метод.");
09          }
10
11          protected internal void ProtectedInternalMethod()
12          {
13  Console.WriteLine("3. Protected Internal Method - внутренний защищенный метод.");
14          }
15      }                              ////////////// конец базового класса BaseInternal ////////////////
16
17  internal class DerivedInternal : BaseInternal  ///// производный класс DerivedInternal
18      {
19          internal void DerivedInternalMethod()
20          {
21              ProtectedInternalMethod();                  // 4. см. стр 11, 13 л-га 13
22          }
23      }           ////////////////////////// конец производного класса DerivedInternal ////////////////////////////////
24
25      public class PublicClass                ////////////////////// класс PublicClass //////////////////////
26      {
27          public void PublicMethod()
28          {
29              Console.WriteLine("1. Public Method - открытый метод.");
30
```

```
31          BaseInternal myBaseInternal = new BaseInternal();
32          myBaseInternal.BaseInternalMethod();          // 2. см. стр 06, 08 л-га 13
33
34          DerivedInternal myDerivedInternal = new DerivedInternal();
35          myDerivedInternal.DerivedInternalMethod();    // 3. см. стр 19, 21 л-га 13
36      }
37
38      protected void ProtectedMethod()
39      {
40          Console.WriteLine("5. Protected Method - защищенный метод.");
41          PrivateMethod();                              // 7. см. стр 44, 46 л-га 13
42      }
43
44      private void PrivateMethod()
45      {
46          Console.WriteLine("6. Private Method - закрытый метод.");
47      }
48  }                          ////////////////////// конец класса PublicClass //////////////////////
49  }
```

Ниже приводятся выходные данные работы этой программы, демонстрирующие последовательность выполнения методов:



Посмотрите внимательно каждый модификатор в листингах **12** и **13**, перед тем как читать дальнейшие объяснения. Убедитесь в том, что листинги **12** и **13** относятся к разным проектам. Листинг **12** будет консольным приложением (cs-файл), а листинг **13** — библиотекой классов (dll-файл). Консольное приложение создается в VS .NET 2005 известным образом.

Кратко остановимся на создании библиотеки (dll-файла).

В среде разработки VS .NET 2005 для построения компонента ClassLib_Mayo_c80_list3_9.dll необходимо создать проект библиотеки классов **Class Library** (Build –> Build ClassLib_Mayo_c80_list3_9).

Заметьте, что клиент *ConsAppl_Mayo_c80_list_3_8.cs* включает пространство имен компонента ClassLib_Mayo_c80_list3_9.dll (см. выше строки ***02*** и **02**). Благодаря этому компонент ClassLib_Mayo_c80_list3_9.dll попадает в "поле зрения" клиента.

Для программы-клиента ConsAppl_Mayo_c80_list_3_8.cs необходимо добавить компонент ClassLib_Mayo_c80_list3_9.dll как ссылку (Project –> Add Reference… –> …).

Теперь разберем метод Main листинга **12**. Первым вызывается (строка **19** листинга **12**) метод PublicMethod листинга **13** (строки **27…36** листинга **13**), который объявлен открытым (public). Это единственный метод класса PublicClass, который может быть вызван, поскольку два других мето-

да **ProtectedMethod()** (строки 38…42 листинга **13**) и **PrivateMethod()** (строки 44…47 листинга **13**) недоступны классу Visability (см. листинг **12**, строки **14…26**). К методу PrivateMethod имеют доступ только элементы класса PublicClass (см. листинг **13**, строки **25…48**), а класс Visability не имеет доступа к этому методу, поскольку он не наследует классу PublicClass.

Доступ к коду программной сборки листинга **13** возможен только через открытые (public) и защищенные (protected) методы класса PublicClass, который объявлен открытым (public). Другие классы [DerivedInternal (см. листинг **13**, строки **17…23**) и BaseInternal (см. листинг 13, строки **04…15**)] объявлены внутренними (internal), и поэтому *доступ к ним возможен только из кода их программной сборки*. Кроме того, метод ProtectedInternalMetod (см. листинг **13**, строки **11…14**) класса BaseInternal является единственно доступным из класса DerivedInternal, поскольку его модификатор разрешает доступ только производным типам внутри той же программной сборки.

Метод PublicMethod (см. листинг **13**, строки **27…36**) класса PublicClass имеет доступ к методу BaseInternalMethod (см. листинг **13**, строки **06…09**) класса BaseInternal, поскольку классы PublicClass и BaseInternal находятся внутри одной программной сборки. Аналогичный доступ имеется и к методу DerivedInternalMethod (см. листинг 13, строки **19…22**) класса DerivedInternal, поскольку этот класс также принадлежит этой сборке. Модификаторы методов BaseInternalMethod и DerivedInternalMethod указывают на то, что типы в программной сборке могут иметь доступ к открытым и внутренним элементам внутренних типов внутри той же сборки. Наконец, метод ProtectedMethod (см. листинг **13**, строки **38…42**) класса Public**Class** имеет доступ к методу **PrivateMethod** (см. листинг **13**, строки **44…47**), поскольку оба являются элементами одного класса.

Следующий вызов в методе **Main** класса **Visibility** относится к классу **ExternalDerived**, который сам объявлен **внутренним** (**internal**). Он назван **ExternalDerived** из-за его связи с классом **PublicClass**. Этот класс не является внутренним для программы листинга 13 и наследует классу **PublicClass**. Вспомните, что листинг **13** является отдельной программной сборкой. Поскольку класс **ExternalDerived** (листинг **12**) является производным от класса **PublicClass** (листинг **13**), он может вызывать метод **ProtectedMethod** (см. листинг **13**, строки **38…42**).

Контрольные вопросы

1. Что такое инкапсуляция?
2. Каковы преимущества использования инкапсуляции?
3. Что такое спецификатор (модификатор) доступа?
4. Что такое спецификатор доступа **public**?
5. Что такое спецификатор доступа **private**?
6. Что такое спецификатор доступа **protected**?
7. Какие части базовогом класса могут использоваться производным классом?
8. Почему программисты требуют, чтобы доступ к некоторым атрибутам класса осуществлялся только с помощью методов-членов?

Ответы на контрольные вопросы

1. Инкапсуляция — это способ связывания атрибутов и методов для формирования объекта.

2. Инкапсуляция позволяет программисту реализовать проверку правильности использования атрибутов и методов, поместив атрибуты и методы в класс, затем в классе определив правила для управления доступом к ним.

3. Спецификатор доступа — это ключевое слово языка программирования, которое говорит компьютеру, какая часть программы может получить доступ к атрибутам и методам, являющимся членами класса.

4. Спецификатор доступа **public** определяет атрибуты и методы, которые доступны при использовании экземпляра любого класса.

5. Спецификатор доступа **private** определяет атрибуты и методы, которые доступны только методам, определенным в данном классе.

6. Спецификатор доступа **protected** определяет атрибуты и процедуры, которые могут быть наследованы другими классами (*производными*).

7. Производный класс наследует **public**- и **protected**-члены базового класса.

8. Программистам требуется, чтобы отдельные атрибуты класса были доступны только методам-членам — это позволит проверять значения, присваиваемые эти атрибутам. Программист, который хочет получить доступ к отдельным атрибутам, вызывает метод-член, который выполняет проверку перед присвоением значений атрибутам.

Литература к курсу

Базовый учебник

1. Мейер Б. Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.

Основная

2. Буч Г., Якобсон А., Рамбо Дж. **UML**. С.-Петербург: Питер, 2006.
3. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. С.-Петербург: Питер, 2006.
4. Забудский Е.И. Учебно-методические материалы по дисциплине «Объектно-ориентированный анализ и программирование». М.: Кафедра ОИиППО ГУ-ВШЭ, 2005,

   Internet-ресурс – http://new.hse.ru/C7/C17/zabudskiy-e-i/default.aspx .
5. Кватрани Т. Визуальное моделирование с помощью Rational Rose 2002 и UML. М.: Вильямс, 2003.
6. Лафоре Р. Объектно-ориентированное программирование в C++. С.-Петербург: Питер, 2005.
7. Троелсен Э. C# и платформа .NET. С.-Петербург: Питер, 2006.
8. Синтес А. Освой самостоятельно объектно-ориентированное программирование за 21 день. Москва; С.-Петербург; Киев: Вильямс, 2002.

Дополнительная – **Internet-ресурсы**

9. Новые книги раздела **C#** – http://books.dore.ru/bs/f6sid16.html
10. **C#** и **.NET** по шагам – http://www.firststeps.ru .
11. **UML** – язык графического моделирования – http://www.uml.org/ .
12. **JUnit** – каркас тестирования для испытания *Java*-классов – http://www.junit.org .
13. Пакет объектного моделирования **Rational Rose** – http://www-306.ibm.com/software/rational/

Дополнительная – книги

14. Мэтт Вайсфельд. Объектно-ориентированный подход: Java, **.**NET, C++. М.: КУДИЦ-ОБРАЗ, 2005.
15. Дж. Кьоу, М. Джеанини. Объектно-ориентированное программирование. С.-Петербург: Питер, 2005.

Упражнение по программированию (задание на дом)

1. Реализовать в среде MS VS .NET 2005 и проанализировать рассмотренные программы: **листинги 1… 13**. Обратите внимание на программу в листингах **12** и **13**. Код листинга **13** компилируется в **dll-файл (компонент)**. **CS-файл** листинга **12** является **клиентом** и использует компонент. См. краткую инструкцию на стр. 27.

2. Откорректируйте программу, приведенную в Листинге 1 (см. раздел 1.4) в соответствии с указанием в строке **16**. Интерфейс метода **calculateSquare(int value)** **не изменяйте** (строка **15**). Фантазируйте и дерзайте.