Лабораторная работа №5

ХАРЬКОВСКИЙ НАЦИОНАЛЬНЫЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И МЕХАТРОНИКИ

Кафедра информационных технологий и мехатроники

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по проведению лабораторных работ по дисциплине
«Объектно-ориентированное программирование»
для студентов специальности 6.050101 “Компьютерные науки”

Разработчик - доцент кафедры информационных технологий и мехатроники
кандидат технических наук, старший научный сотрудник
Тимонин Владимир Алексеевич

Харків 2015

Лабораторная работа №5
Исследование возможностей интегрированной среды разработки Visual C# для создания приложений с производными классами.

Цель работы – исследовать возможности интегрированной среды разработки Visual Studio 2010 и получить практические навыки по созданию приложений с производными классами.

1. Теоретические сведения

Наследование является одним из трех основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций. Благодаря наследованию можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие, более конкретные классы, добавляя в него свои индивидуальные особенности.

В языке C# класс, который наследуется, называется базовым, а класс, который наследует, — производным. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы.

1.1. Основы наследования

Поддержка наследования в C# состоит в том, что в объявление одного класса разрешается вводить другой класс. Для этого при объявлении производного класса указывается базовый класс.

Для любого производного класса можно указать только один базовый класс. Производный класс наследует все члены своего базового класса, в том числе переменные экземпляра, методы, свойства и индексаторы. В C# не предусмотрено наследование нескольких базовых классов в одном производном классе, но, тем не менее, можно создать иерархию наследования, в которой производный класс становится базовым для другого производного класса.

Главное преимущество наследования заключается в следующем: как только будет создан базовый класс, в котором определены общие для множества объектов атрибуты, он может быть использован для создания любого числа более конкретных производных классов, а в каждом производном классе может быть выстроена своя собственная классификация.

Общая форма объявления производного класса имеет вид:

```
class <имя_производного_класса> : <имя_базового_класса>
    {
     // тело класса
    }
```

Рассмотрим простой пример создания производного класса. Создадим класс **TwoDShape**, который содержит следующие элементы: ширину и высоту двухмерного объекта, например квадрата, прямоугольника, треугольника и т.д., а также метод, который предназначен для вывода на экран значений ширины и высоты объекта.

```
class TwoDShape
  {
    public double Width;    // ширина объекта
    public double Height;   // высота объекта
    public void ShowDim() // показать размеры объекта
    {
       Console.WriteLine("Ширина и высота равны " + Width + " и " + Height);
    }
  }
```

Класс **TwoDShape** может стать базовым, т.е. отправной точкой для создания классов, описывающих конкретные типы двумерных объектов. Например, класс **TwoDShape** служит для

порождения производного класса **Triangle** (треугольник).

```
class Triangle : TwoDShape  // класс Triangle, производный от класса TwoDShape

  {
    public string Style;        // тип треугольника
    public double Area()      // расчет площади треугольника
    {
      return Width * Height / 2;
    }
    public void ShowStyle() // показать тип треугольника
    {
      Console.WriteLine("Треугольник " + Style);
    }
  }
```

В классе **Triangle** создается особый тип объекта класса **TwoDShape** (в данном случае — треугольник). Кроме того, в класс **Triangle** входят все члены класса **TwoDShape**, к которым добавляются переменная **Style**, предназначена для хранения описания типа треугольника, методы **Area()**, предназначен для расчета площади переменной, и **ShowStyle()**, предназначен для вывода на экран типа треугольника.

Так как в класс **Triangle** входят все члены его базового класса **TwoDShape**, то переменные **Width** и **Height** класса **TwoDShape** доступны для метода **Area()** класса **Triangle**. Кроме того, объекты **tl** и **t2** в методе **Main()** могут обращаться непосредственно к переменным **Width** и **Height**, как будто они являются членами класса Triangle. На рис. 1 схематически показано, каким образом класс **TwoDShape** вводится в класс **Triangle**.

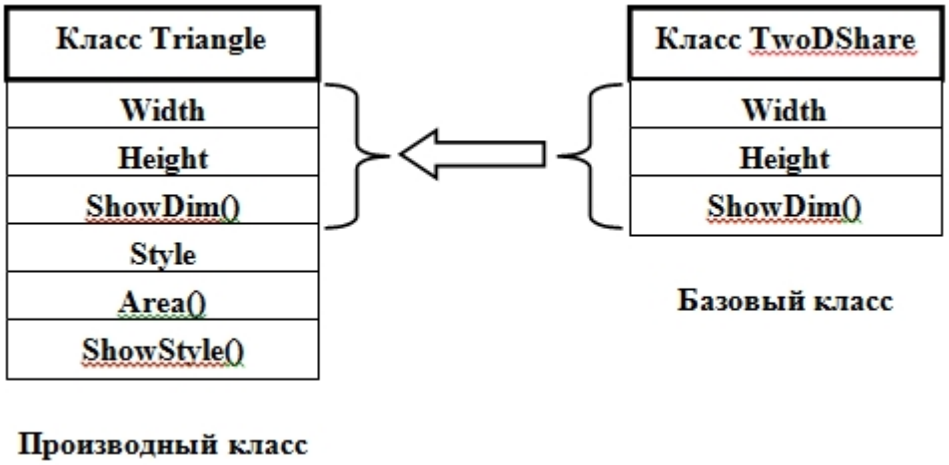

Рис. 1. Схематическое представление класса Triangle

Пример 1. Нижеприведенный код программы «Пример наследования» демонстрирует применение базового **TwoDShape** и производного **Triangle** классов.

Результат выполнения программы представлен на рис. 2.

```
class Program
  {
    static void Main()
    {
      Triangle tl = new Triangle();
      Triangle t2 = new Triangle();
      tl.Width = 4.0;
      tl.Height = 4.0;
      tl.Style = "равнобедренный";
```

```
            t2.Width = 8.0;
            t2.Height = 12.0;
            t2.Style = "прямоугольный";
            Console.WriteLine("Сведения об объекте tl: ");
            tl.ShowStyle();
            tl.ShowDim();
            Console.WriteLine("Площадь равна " + tl.Area());
            Console.WriteLine() ;
            Console.WriteLine("Сведения об объекте t2: ");
            t2.ShowStyle();
            t2.ShowDim();
            Console.WriteLine("Площадь равна " + t2.Area());
            Console.Write("\r\nДля завершения работы приложения нажмите клавишу
<Enter>");
            Console.Read();
        }
    }
```

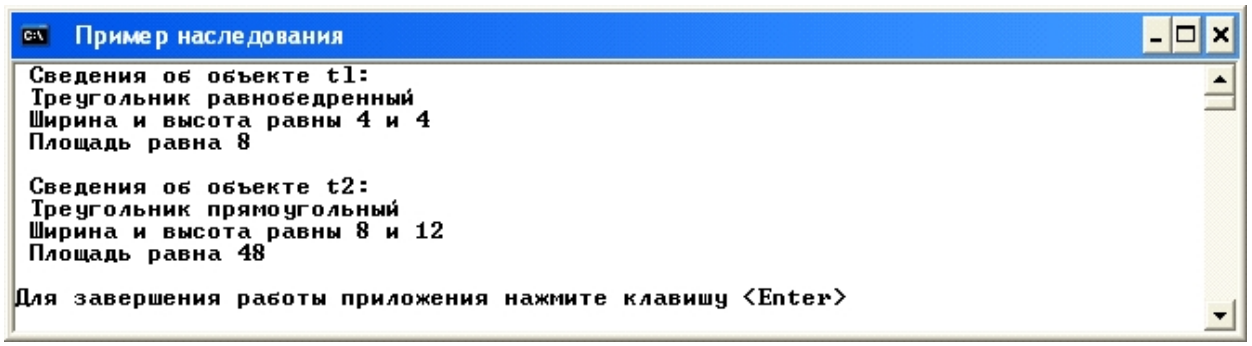

Рис. 2. Результат выполнения программы «Пример наследования»

На базе класса **TwoDShape** множество производных классов, например, производный класс **Rectangle**, инкапсулирующий прямоугольники.

```
class Rectangle : TwoDShape
  {
    public bool IsSquare() // проверка – является прямоугольник квадратом
    {
      if(Width == Height) return true;
      return false;
    }
    public double Area() // расчет площади прямоугольника
    {
      return Width * Height;
    }
  }
```

В класс **Rectangle** входят все члены класса **TwoDShape**, к которым добавлен метод **IsSquare()**, определяющий, является ли прямоугольник квадратом, а также метод **Area()**, вычисляющий площадь прямоугольника.

Несмотря на то, что класс **TwoDShape** является базовым для класса **Triangle**, в то же время он представляет собой совершенно независимый и самодостаточный класс, что означает его самостоятельное использование, например:

```
TwoDShape shape = new TwoDShape();
shape.Width = 10;
```

shape.Height =20;
shape.ShowDim();

Объект класса **shape** никак не связан с каким-либо объектом из классов, производных от класса **TwoDShape**, и вообще не имеет к ним доступа.

1.2. Доступ к членам базового класса при наследовании

Наследование класса не отменяет ограничения, накладываемые на доступ к членам класса. Если в производный класс входят закрытые члены базового класса, то они оказываются недоступными, т.к. закрытый член класса остается закрытым в своем классе . Он не доступен из кода за пределами своего класса, включая и производные классы.

Так, если сделать закрытыми переменные класса TwoDShape, они станут недоступными в классе Triangle. Класс Triangle не будет компилироваться, потому что обращаться к переменным Width и Height из метода Area() запрещено, так как переменные Width и Height являются закрытыми и они доступны только для других членов своего класса, но не для членов производных классов.

Для преодоления ограничение на доступ к частным членам базового класса из производного класса в C# предусмотрены разные способы. Один из них состоит в использовании защищенных (**protected**) членов класса, а второй — в применении открытых свойств для доступа к закрытым данным.

1.2.1. Использование свойства класса

Свойство позволяет управлять доступом к переменной экземпляра. Например, с помощью свойства можно ввести ограничения на доступ к значению переменной или же сделать ее доступной только для чтения. Так, если сделать свойство открытым, но объявить его базовую переменную закрытой, то этим свойством можно будет воспользоваться в производном классе, но нельзя будет получить непосредственный доступ к его базовой закрытой переменной.

Модифицируем класс **TwoDShape**, в котором переменные **Width** и **Height** превращены в свойства, выполняющие проверку: являются ли положительными значения свойств **Width** и **Height**.

```
class TwoDShape
  {
    double width;
    double height;
    public double Width
    {
      get { return width; }
      set { width = value < 0 ? -value : value; }
    }
    public double Height
    {
      get { return height; }
      set { height = value < 0 ? -value : value; }
    }
    public void ShowDim()
    {
      Console.WriteLine(" Ширина и высота равны " + Width + " и " + Height);
    }
  }
```

В этом варианте свойства **Width** и **Height** предоставляют доступ к закрытым членам **width** и **height** класса **TwoDShape**, в которых фактически хранятся значения ширины и высоты объекта. Следовательно, значения членов **width** и **height** класса **TwoDShape** могут быть установлены и получены с помощью соответствующих открытых свойств, несмотря на то, что сами эти члены по-прежнему остаются закрытыми.

При использовании модифицированного класса в программе «Пример наследования» результат

выполнения будет точно таким же, как и при использовании класса с открытыми элементами, но в последнем случае элементы **width** и **height** являются закрытыми в классе и доступ к ним осуществляется посредством свойств.

1.2.2. Использование защищенных членов класса

Так как закрытый член базового класса недоступен для производного класса, то для доступа к некоторому члену базового класса из производного класса этот член необходимо сделать открытым, а это означает доступность этого члена для всего кода, что далеко не всегда желательно. Ограничить доступность позволяет создание защищенного члена класса. Защищенный член является открытым в пределах иерархии классов, но закрытым за пределами этой иерархии.

Защищенный член создается с помощью модификатора доступа **protected**. Если член класса объявляется как **protected**, он становится закрытым, но за исключением одного случая, когда защищенный член наследуется. В этом случае защищенный член базового класса становится защищенным членом производного класса, а значит, доступным для производного класса. Таким образом, используя модификатор доступа **protected**, можно создать члены класса, являющиеся закрытыми для своего класса, но всё же наследуемыми и доступными для производного класса.


Пример 2. Нижеприведенный код программы демонстрирует применения модификатора доступа **protected**.

```
class A
  {
    protected int i, j; // члены доступные для производного класса
    public void Set(int a, int b)
    {
      i = a;  j = b;
    }
    public void Show()
    {
      Console.WriteLine (i + " " + j);
    }
  }
  class B : A
  {
    int k;
    public void Setk()
    {
      k = i * j;
      Console.WriteLine(k);
    }
  }
  class Program
  {
    static void Main()
    {
      B ob = new B();
      ob.Set(2, 3); // метод класса A, наследуемый классом B
      ob.Show(); // метод класса A, наследуемый классом B
      Console.WriteLine(i); // Ошибка! Защищенный член i класса A
      ob.Setk(); // метод класса B, в котором используются члены класса A
    }
  }
```

В данном примере класс **A** наследуется классом **B**, а его члены **i** и **j** объявлены как **protected**, и поэтому они доступны для метода **Setk()**. Если бы члены **i** и **j** класса **A** были объявлены как **private**, то они оказались бы недоступными для класса **B**, и приведенный выше код нельзя было бы скомпилировать.

Состояние **protected** сохраняется за членом класса независимо от количества уровней наследования. Поэтому когда производный класс используется в качестве базового для другого производного класса, любой защищенный член исходного базового класса, наследуемый первым производным классом, наследуется как защищенный и вторым производным классом.

Оператор **Console.WriteLine(i);** содержит ошибку по причине недоступности переменной **i**, так как защищенный член класса не может быть доступен вне иерархии классов.

Таким образом, модификатор доступа **protected** следует применять в том случае, если требуется создать член класса, доступный для всей иерархии классов, но закрытый для остального кода.

1.3. Конструкторы при наследовании

В иерархии классов допускается, чтобы у базовых и производных классов были свои собственные конструкторы. В связи с этим возникает следующий резонный вопрос: какой конструктор отвечает за построение объекта производного класса: конструктор базового класса, конструктор производного класса или же оба? На этот вопрос можно ответить так: конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса — производную часть этого объекта. И в этом есть своя логика, поскольку базовому классу неизвестны и недоступны любые элементы производного класса, а значит, их конструирование должно происходить раздельно.

В приведенных выше примерах данный вопрос не возникал, поскольку они опирались на автоматическое создание конструкторов, используемых в C# по умолчанию. Но на практике конструкторы определяются в большинстве классов.

1.3.1. Применение конструктора произвольного класса

Если конструктор определен только в производном классе, то все происходит очень просто: конструируется объект производного класса, а базовая часть объекта автоматически конструируется его конструктором, используемым по умолчанию.

Для вышеприведенного примера модифицируем класс **Triangle**, в котором член **Style** сделаем закрытым, определим конструктор, предназначенный для инициализации членов базового и производного классов. Модифицированный класс имеет вид:

```
class Triangle : TwoDShape
  {
    string Style;
    public Triangle(string s, double w, double h)  // конструктор
    {
      Width = w; // инициализация члена базового класса
      Height = h; // инициализация члена базового класса
      Style = s;    // инициализация члена производного класса
    }
    public double Area() // расчет площади треугольника
    {
      return Width * Height / 2;
    }
    public void ShowStyle() // показать тип треугольника
    {
      Console.WriteLine(" Треугольник " + Style);
    }
  }
```

Модифицированный вариант программы «Пример наследования» имеет вид:

```
static void Main()
```

```
    {
      Triangle tl = new Triangle("равнобедренный", 4.0, 4.0);
      Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);
      Console.WriteLine(" Сведения об объекте tl: ");
      tl.ShowStyle();
      tl.ShowDim();
      Console.WriteLine(" Площадь равна " + tl.Area());
      Console.WriteLine();
      Console.WriteLine(" Сведения об объекте t2: ");
      t2.ShowStyle();
      t2.ShowDim();
      Console.WriteLine(" Площадь равна " + t2.Area());
    }
```

В данном примере конструктор класса **Triangle** инициализирует наследуемые члены класса **TwoDShape** вместе с его собственным полем **Style**.

1.3.2. Применение конструктора базового класса

Когда конструкторы определяются как в базовом, так и в производном классе, процесс построения объекта несколько усложняется, поскольку должны выполняться конструкторы обоих классов. В данном случае приходится обращаться к еще одному ключевому слову языка C#: **base**, которое находит двоякое применение: во-первых, для вызова конструктора базового класса; и во-вторых, для доступа к члену базового класса, скрывающегося за членом производного класса.

С помощью формы расширенного объявления конструктора производного класса и ключевого слова **base** в производном классе может быть вызван конструктор, определенный в его базовом классе. Общая форма этого расширенного объявления имеет вид:

```
<конструктор_производного_класса>(<список_параметров>) : base
                                                    (<список_аргументов>)
    {
      // тело конструктора
    }
```

где <**список_аргументов**> обозначает любые аргументы, необходимые конструктору в базовом классе.

Когда в производном классе указывается ключевое слово **base**, вызывается конструктор из его непосредственного базового класса. Следовательно, ключевое слово **base** всегда обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим классом. Это справедливо даже для многоуровневой иерархии классов.

Аргументы передаются базовому конструктору в качестве аргументов метода **base()**. Если же ключевое слово отсутствует, то автоматически вызывается конструктор, используемый в базовом классе по умолчанию.

Для демонстрации применения ключевого слова **base** модифицируем класс **TwoDShape**, определив в нем конструктор, инициализирующий свойства **Width** и **Height**, и класс **Triangle**, определив в нем конструктор, вызывающий конструктор базового класса. Затем этот конструктор применим в программе «Пример наследования».

Модифицированный класс **TwoDShape** имеет вид:

```
class TwoDShape
  {
    double width;
    double height;
    public TwoDShape(double w, double h) // конструктор
    {
      Width = w;
      Height = h;
```

```
        }
        public double Width
        {
           get { return width; }
           set { width = value < 0 ? -value : value; }
        }
        public double Height
        {
           get { return height; }
           set { height = value < 0 ? -value : value; }
        }
        public void ShowDim()
        {
          Console.WriteLine(" Ширина и высота равны " + Width + " и " + Height);
        }
    }
```

Модифицированный класс **Triangle** имеет вид:

```
class Triangle : TwoDShape
    {
        string Style;
        public Triangle(string s, double w, double h) : base(w, h) // конструктор
        {
           Style = s;
        }
        public double Area() // расчет площади треугольника
        {
           return Width * Height / 2;
        }
        public void ShowStyle() // показать тип треугольника
        {
           Console.WriteLine(" Треугольник " + Style);
        }
    }
```

Модифицированный вариант программы «Пример наследования» имеет вид:

```
static void Main()
        {
           Triangle tl = new Triangle("равнобедренный", 4.0, 4.0);
           Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);
           Console.WriteLine(" Сведения об объекте tl: ");
           tl.ShowStyle();
           tl.ShowDim();
           Console.WriteLine(" Площадь равна " + tl.Area());
           Console.WriteLine();
           Console.WriteLine(" Сведения об объекте t2: ");
           t2.ShowStyle();
           t2.ShowDim();
           Console.WriteLine(" Площадь равна " + t2.Area());
        }
```

В данном варианте конструктор **Triangle()** вызывает метод **base** с параметрами **w** и **h**. Это, в свою очередь, приводит к вызову конструктора **TwoDShape()**, инициализирующего свойства **Width** и **Height** значениями параметров **w** и **h**. Они больше не инициализируются средствами самого класса **Triangle**, где теперь остается инициализировать только его собственный член **Style**, определяющий

тип треугольника.

Благодаря этому класс **TwoDShape** высвобождается для конструирования своего подобъекта любым избранным способом. Более того, в класс **TwoDShape** можно ввести функции, о которых даже не будут подозревать производные классы, что предотвращает нарушение существующего кода.

С помощью ключевого слова **base** можно вызвать конструктор любой формы, определяемой в базовом классе, причем выполняться будет лишь тот конструктор, параметры которого соответствуют переданным аргументам.

Пример 3. Нижеприведенный код программы «Пример наследования» демонстрирует применение конструкторов базового **TwoDShape** и производного **Triangle** классов. В классы **TwoDShape** и **Triangle** включены как используемые по умолчанию конструкторы, так и конструкторы, принимающие один и два аргумента.

Результат выполнения программы представлен на рис. 3.

```
class TwoDShape
   {
      double width;
      double height;
      public TwoDShape() // конструктор по умолчанию
      {
         Width = Height = 0.0;
      }
      public TwoDShape(double w, double h) // конструктор с 2-я аргументами
      {
         Width = w;
         Height = h;
      }
      public TwoDShape(double x) // // конструктор с 1-м аргументов
      {
         Width = Height = x;
      }
      public double Width
      {
         get { return width; }
         set { width = value < 0 ? -value : value; }
      }
      public double Height
      {
         get { return height; }
         set { height = value < 0 ? -value : value; }
      }
      public void ShowDim()
      {
        Console.WriteLine(" Ширина и высота равны " + Width + " и " + Height);
      }
   }
class Triangle : TwoDShape
   {
      string Style;
      public Triangle() // конструктор по умолчанию
      {
         Style = "null";
```

```
    }
    public Triangle(string s, double w, double h) : base(w, h) // конструктор
    {
        Style = s;
    }
    public Triangle(double x) : base(x) // конструктор
    {
        Style = "равнобедренный";
    }
    public double Area() // расчет площади треугольника
    {
        return Width * Height / 2;
    }
    public void ShowStyle() // показать тип треугольника
    {
        Console.WriteLine(" Треугольник " + Style);
    }
}
class Program
{
    static void Main()
    {
        Console.Title = "  Пример наследования";
        Console.BackgroundColor = ConsoleColor.White;
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Black;
        Triangle tl = new Triangle();
        Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);
        Triangle t3 = new Triangle(4.0);
        tl = t2;
        Console.WriteLine(" Сведения об объекте tl: ");
        tl.ShowStyle() ;
        tl.ShowDim() ;
        Console.WriteLine(" Площадь равна " + tl.Area());
        Console.WriteLine() ;
        Console.WriteLine(" Сведения об объекте t2: ");
        t2.ShowStyle();
        t2.ShowDim();
        Console.WriteLine(" Площадь равна " + t2.Area());
        Console.WriteLine();
        Console.WriteLine(" Сведения об объекте t3: ");
        t3.ShowStyle();
        t3.ShowDim();
        Console.WriteLine(" Площадь равна " + t3.Area());
        Console.Write("\r\nДля завершения работы приложения нажмите
                        клавишу <Enter>");
        Console.Read();
    }
}
```

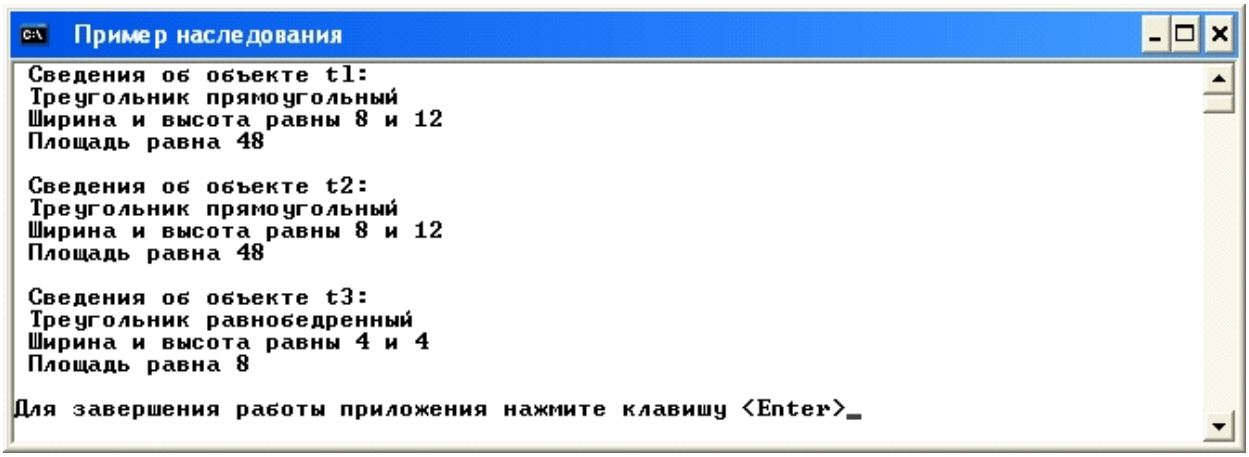

Рис. 3. Результат выполнения модифицированной программы «Пример наследования»

1.4. Наследование и сокрытие имен

В производном классе можно определить член с таким же именем, как и у члена его базового класса. В этом случае член базового класса скрывается в производном классе. И хотя формально в C# это не считается ошибкой, компилятор все же выдаст сообщение, предупреждающее о том, что имя скрывается. Если член базового класса требуется скрыть намеренно, то перед его именем следует указать ключевое слово **new**, чтобы избежать появления подобного предупреждающего сообщения. Следует, однако, иметь в виду, что это совершенно отдельное применение ключевого слова **new**, не похожее на его применение при создании экземпляра объекта.

Нижеприведенный код программы демонстрирует сокрытие имени с наследственной связью.

```
class A
   {
      public int i = 0;
   }
   // Создать производный класс
   class B : A
   {
      new int i; // этот член скрывает член i из класса A
      public B(int b)
      {
         i = b; // член i в классе B
      }
      public void Show()
      {
         Console.WriteLine("Член i в производном классе: " + i);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         B ob = new B(2);
         ob.Show();
         Console.Read();
      }
   }
```

Использование ключевого слова **new** в операторе **new int i;** сообщается компилятору о том, что вновь создаваемая переменная i намеренно скрывает переменную i из базового класса A. Если же

опустить ключевое слово new в этой строке кода, то компилятор выдаст предупреждающее сообщение.

В классе B определяется собственная переменная экземпляра i, которая скрывает переменную i из базового класса A. Поэтому при вызове метода Show() для объекта типа B выводится значение переменной i, определенной в классе B, а не той, что определена в классе A.

В результате выполнения вышеприведенного кода будет выдано сообщение

Член i в производном классе: 2

Имеется еще одна форма ключевого слова **base**, которая действует подобно ключевому слову this, за исключением того, что она всегда ссылается на базовый класс в том производном классе, в котором она используется. Ниже эта форма приведена в общем виде:

base.<член>

где член может обозначать метод или переменную экземпляра. Эта форма ключевого слова **base** чаще всего применяется в тех случаях, когда под именами членов производного класса скрываются члены базового класса с теми же самыми именами.

Нижеприведенный код программы демонстрирует применение ключевого слова **base** для преодоления препятствия, связанного с сокрытием имен.

```
class A
  {
    public int i = 0;
  }
  // Создать производный класс
  class B : A
  {
    new int i; // этот член скрывает член i из класса A
    public B(int a, int b)
    {
      base.i = a; // здесь обнаруживается скрытый член из класса A
      i = b; // член i из класса B
    }
    public void Show()
    {
      // здесь выводится член i из класса A
      Console.WriteLine("Член i в базовом классе:" + base.i);
      // здесь выводится член i из класса B
      Console.WriteLine("Член i в производном классе:" + i);
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      B ob = new B(1,2);
      ob.Show ();
      Console.Read();
    }
  }
```

В результате выполнения вышеприведенного кода будут выданы сообщения

Член i в базовом классе: 1

Член i в производном классе: 2

Несмотря на то что переменная экземпляра i в производном классе B скрывает переменную i из базового класса A, ключевое слово base разрешает доступ к переменной i, определенной в базовом классе.

С помощью ключевого слова **base** могут также вызываться скрытые методы. Например, в

приведенном ниже коде класс B наследует класс A и в обоих классах объявляется метод Show(). А затем в методе Show() класса B с помощью ключевого слова base вызывается вариант метода Show(), определенный в классе A.

Нижеприведенный код программы демонстрирует вызов скрытого метода

```
class A
   {
      public int i = 0;
      // метод Show() в классе A
      public void Show()
      {
         Console.WriteLine("Член i в базовом классе: " + i);
      }
   }
   // Создать производный класс
   class B : A
   {
      new int i; // этот член скрывает член i из класса A
      public B(int a, int b)
      {
         base.i = a; // здесь обнаруживается скрытый член из класса A
         i = b; // член i из класса B
      }
      // Здесь скрывается метод Show() из класса A
      new public void Show()
      {
         base.Show(); // здесь вызывается метод Show() из класса A
         Console.WriteLine("Член i в производном классе: " + i);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         B ob = new B(1,2);
         ob.Show();
         Console.Read();
      }
   }
```

В результате выполнения вышеприведенного кода будут выданы сообщения

Член i в базовом классе: 1

Член i в производном классе: 2

Ключевое слово **new** используется в вышеприведенном коде с целью сообщить компилятору о том, что метод Show(), вновь объявляемый в производном классе B, намеренно скрывает другой метод Show(), определенный в базовом классе A.

1.5. Пример реализации наследования класса

Пример 4. Нижеприведенный код программы демонстрирует работу приложения «Менеджеры компании», в котором используются базовый класс Person и производный класс Manager.

Базовый класс Person содержит поля (fam - имя служащего, tab - номер служащего, salary - оклад служащего (должен быть в диапазоне: от 3000 до 7000)), свойства (Fam, Tab, Salary) и конструкторы.

В производный класс Manager добавляются поля (company - наименование компании, в которой

работает менеджер, sale - объем продаж менеджера), свойства (Company, Sale и свойство Bonus - вычисление размер премии (5% от объема продаж)), метод, обеспечивающий вывод значений полей двух классов на экран монитора, и конструкторы.

```
// базовый класс
class Person
{
    // поля
    string fam;     // фамилия
    int salary;     // зарплата
    int tab;        // табельный номер
    // конструкторы
    public Person() { }
    public Person(string Fam, int Tab, int Salary)
    {
        fam = Fam;
        tab = Tab;
        salary = Salary;
    }
    // свойства
    public string Fam
    {
        set { if (fam == null) fam = value; }
        get { return fam; }
    }
    public int Tab
    {
        set { tab = value; }
        get { return (tab); }
    }
    public int Salary
    {
        get { return salary; }
        set { if ((value > 3000) && (value < 7000)) salary = value; }
    }
}
// производный класс
class Manager : Person
{
    // поля
    string company;     // название компании
    int sale;           // объем продаж
    // конструкторы
    public Manager() { }
    public Manager(string Fam, int Tab, int Salary, string Company, int Sale)
        : base(Fam, Tab, Salary)
    {
        company = Company;
        sale = Sale;
    }
    // свойства
    public string Company
    {
```

```
        set { if (company == null) company = value; }
        get { return company; }
    }
    public int Sale
    {
        set { sale = value; }
        get { return (sale); }
    }
    public double Bonus //свойство для подсчета премии от объема продаж
    {
        get { return sale * 0.05; }
    }
    // метод вывода значений всех полей
    public void Show()
    {
        Console.WriteLine("  Фамилия - " + base.Fam);
        Console.WriteLine("  Табельный номер - " + base.Tab);
        Console.WriteLine("  Зарплата - {0} грв.", base.Salary);
        Console.WriteLine("  Компания - " + company);
        Console.WriteLine("  Объем продаж - " + sale);
        Console.WriteLine("  Бонус - {0} грв.", Bonus);
        Console.WriteLine("  Общая сумма зарплаты - {0} грв.", (Bonus + base.Salary));
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.Title = "Менеджеры компании";
        Console.BackgroundColor = ConsoleColor.White;
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Black;
        int i, N;
        int n = 0; // количество инициализированных объектов
        Console.Write("  Введите количество менеджеров в компании ");
        N = Convert.ToInt32(Console.ReadLine());
        //объявляем массив на N объектов класса Manager
        Manager[] pr = new Manager[N];
        for (i = 0; i < N; i++)
            pr[i] = new Manager();
        String s, subkey, key;
        do                          // выводим меню для работы с программой
            {
            Console.WriteLine("      Меню");
            Console.WriteLine("  1 - Ввод данных");
            Console.WriteLine("  2 - Вывод данных");
            Console.WriteLine("  3 - Поиск данных по табельному номеру");
            Console.WriteLine("  4 - Выход");
            Console.WriteLine();
            Console.Write("Ваш выбор...  ");
            key = Console.ReadLine();
            switch (key)
```

```
{
case "1": // ввод данных выполняется через соответствующие свойства
do
   {
   Console.Write("  Введите фамилию: ");
   s = Console.ReadLine();
   pr[n].Fam = s;
   Console.Write("  Введите табельный номер: ");
   s = Console.ReadLine();
   pr[n].Tab = Convert.ToInt32(s);
   Console.WriteLine("  Внимание! Зарплата должна быть в пределах от
                        3000 до 7000 ");
   Console.Write("  Введите зарплату: ");
   s = Console.ReadLine();
   pr[n].Salary = Convert.ToInt32(s);
   Console.Write("  Введите название компании: ");
   s = Console.ReadLine();
   pr[n].Company = s;
   Console.Write("  Введите объем продаж: ");
   s = Console.ReadLine();
   pr[n].Sale = Convert.ToInt32(s);
   n++;
   Console.Write("  Продолжить?(Y/N)...");
   subkey = Console.ReadLine();
   if (n >= N) Console.WriteLine("  Формирование списка менеджеров
                                    завершено!");
   } while (((subkey == "y") || (subkey == "Y")) && (n<N)); break;
case "2": // вывод данных
for (i = 0; i < n; i++)
   {
   pr[i].Show();
   Console.WriteLine();
   } break;
case "3": // поиск данных по табельному номеру
{
   bool tr = true;
   Console.Write("  Введите табельный номер: ");
   s = Console.ReadLine();
   int num = Convert.ToInt32(s);
   for (i = 0; i < n; i++)
      if (pr[i].Tab == num)
         {
         Console.WriteLine("  Фамилия={0}, таб. номер={1}, бонус={2}",
                              pr[i].Fam, pr[i].Tab, pr[i].Bonus);
         tr = false;
         Console.WriteLine();
         }
   if (tr) Console.WriteLine("  С таким табельным номером нет
                              менеджера!");
   Console.WriteLine();
   } break;
}
```

```
        } while (key != "4");
    }
}
```

2. Рабочее задание

Задание 1. Руководствуясь теоретическим материалом раздела 1 изучить возможности языка C# по созданию приложений, использующие производные классы, и выполнить примеры, описанные в этом разделе.

Задание 2. Разработать приложение «Расчет объема пирамиды», с помощью которого можно рассчитать объемы правильной и усеченной пирамид. В качестве базового класса использовать класс «Прямоугольный параллелепипед», который содержит три свойства (длина и ширина основания, высота), метод расчета объема параллелепипеда и метод отображения результатов расчета на экране монитора.

Примечание. Объем усеченной пирамиды равен одной трети произведения высоты h на сумму площадей верхнего основания S1, нижнего основания усеченной пирамиды S2 и средней пропорциональной между ними:

$$V = \frac{1}{3}h(S_1 + \sqrt{S_1 S_2} + S_2)$$

3. Контрольные вопросы

1. Что такое базовый и производный класс?
2. Как объявляется производный класс?
3. Перечислите способы доступа к членам базового класса?
4. Как объявляется конструктор производного класса?
5. Как используются конструкторы базового класса в производных классах?
5. Каким образом можно скрыть члены базового класса?
6. Каким образом можно получить доступ к скрытым членам базового класса?

Литература

1. Голощапов А.Л. Microsoft Visual Studio 2010. – СПб.:БХВ-Петербург, 2011. – 544 с.: ил.
2. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. Пер. с англ. - М.: «Русская Редакция», 2002.- 576 с.: ил.
3. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 2. Пер. с англ. - М.: «Русская Редакция», 2002.- 624 с.: ил.
4. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0. Пер. с англ. - М.: Издательский дом "Вильямс", 2011. — 1392 с.: ил.
5. Фленов М.Е. Библия C#. - СПб.: БХВ-Петербург, 2011.– 560с.: ил.
6. Шилдт Г. C# Учебный курс. – СПб.: Питер, Издательская группа BHV, 2003. – 512 с.: ил.