Глава 8

Методы методов

В этой главе...

- Определение метода
- Передача аргументов в метод
- Получение результата из метода
- Еще раз о `WriteLine()`

Программистам необходима возможность разбивать большие программы на части, чтобы с ними было легче работать. Например, программы, рассмотренные в предыдущих главах, уже достигли своего предельного размера, с которым, как с единым целым может справиться один человек.

Язык C# позволяет разделить код на части, называемые *методами* (или *функциями*). Правильно спроектированные и реализованные методы существенно облегчают написание сложных программ.

Методы являются эквивалентом того, что в других языках программирования называется функциями, процедурами или подпрограммами. Основное отличие в том, что метод всегда является членом класса.

Определение и использование метода

Рассмотрим следующий пример:

```
class Example
{
   public int anInt;             // Не статический член
   public static int staticInt   // Статический член
   public void MemberMethod()    // Не статический метод
   {
      Console.WriteLine("Это метод экземпляра");
   }
   public static void ClassMethod() // Статический метод
   {
      Console.WriteLine("Это метод класса");
   }
}
```

Элемент `anInt` является членом-данными, с которыми вы познакомились в главе 7, “Немного о классах”. Однако элемент `MemberMethod()` для вас нов. Он известен как *метод экземпляра*, или *функция-член*, представляющая собой набор кода C#, который может быть выполнен с помощью ссылки на имя этого метода. Честно говоря, такое определение смущает даже меня, так что лучше рассмотреть, что такое метод, на примерах.

Примечание. Отличие между статическими и нестатическими методами крайне важно. Частично данная тема будет раскрыта в настоящей главе, но более подробно об этом

речь пойдет в главе 9, “Пару слов об этом”, в которой будут более детально рассмотрены нестатические методы.

Для вызова нестатического метода необходим экземпляр класса. Для вызова статического метода требуется имя класса, а не экземпляр. В следующем фрагменте присваиваются значения члену объекта `anInt` и члену класса (статическому члену) `staticInt`:

```
Example example   = new Example(); // Создание объекта
example.anInt     = 1;             // Инициализация члена с
                                   // использованием объекта
Example.staticInt = 2;             // Инициализация члена с
                                   // использованием класса
```

Практически аналогично в приведенном далее фрагменте происходит обращение (путем вызова) к методам `MemberMethod()` и `ClassMethod()`:

```
Example example = new Example();  // Создание объекта
example.MemberMethod();           // Вызов метода-члена
                                  // с указанием объекта
Example.ClassMethod();            // Вызов метода класса с
                                  // указанием класса

// Приведенные далее строки не компилируются
example.ClassMethod();            // Нельзя обращаться к методу
                                  // класса с указанием объекта
Example.MemberMethod();           // Нельзя обращаться к методу
                                  // экземпляра с указанием класса
```

Выражение `example.MemberMethod()` передает управление коду, содержащемуся внутри метода. Процесс вызова `Example.ClassMethod()` практически такой же. В результате выполнения приведенного выше фрагмента кода на экран выводится следующее:

```
Это метод экземпляра
Это метод класса
```

После того как метод завершает свою работу, он передает управление в точку, из которой был вызван.

В приведенном примере код методов не делает ничего особенного, кроме вывода на экран единственной строки, но в общем случае методы выполняют различные сложные операции, такие как вычисление математических функций, объединение строк, сортировка массивов или отправка электронных писем. Словом, сложность решаемых методами задач ничем не ограничена. Методы могут быть любого размера и любой степени сложности, но все же лучше, чтобы они были небольшими по размеру для удобства работы с ними и уменьшения вероятности ошибок.

Использование методов в ваших программах

В этом разделе для демонстрации того, как разумное определение методов может сделать программу проще для написания и понимания, будет взята монолитная программа `CalculateInterestTable` из главы 4, “Управление потоком выполнения”, и разделена на несколько методов. Такой процесс переделки рабочего кода при сохранении его

Book 1.indb 162 25.09.2008 20:53:26

функциональности называется *рефакторингом*, и Visual Studio 2008 обеспечивает удобное меню Refactor, которое автоматизирует большинство распространенных задач рефакторинга.

Определение методов и их вызовы будут детально рассмотрены ниже в этой главе, а пока считайте данный пример просто кратким обзором.

Чтение комментариев при опущенном программном коде должно способствовать пониманию намерений программиста. Если это не так, значит, вы плохо комментируете свои программы. (И наоборот: если вы не можете, опустив большинство комментариев, понять, что делает программа, на основании имен методов, значит, вы недостаточно ясно именуете методы и/или делаете их слишком большими).

"Скелет" программы `CalculateInterestTable` выглядит следующим образом:

```
public static void Main(string[] args)
{
     // Приглашение ввести начальный вклад.
     // Если вклад отрицателен, генерируется сообщение об
     // ошибке.
     // Приглашение для ввода процентной ставки.
     // Если процентная ставка отрицательна, генерируется
     // сообщение об ошибке.
     // Приглашение для ввода количества лет.
     // Вывод введенных данных.
     // Цикл по введенному количеству лет.
     while(year <= duration)
     {
          // Вычисление значения вклада с начисленными
          // процентами.
          // Вывод результата вычислений.
     }
}
```

Это пример хорошего метода проектирования методов. Если вы изучите программу, то увидите, что она состоит из следующих трех частей:

- ✓ часть начального ввода данных, в которой пользователи вводят вклад, процентную ставку и срок;
- ✓ раздел, выводящий введенную информацию на экран, чтобы пользователь мог убедиться в корректности ввода;
- ✓ последняя часть кода, создающая и выводящая таблицу на экран.

Это хорошее начало для выполнения рефакторинга. Кроме того, внимательнее рассмотрев часть ввода начальной информации, вы увидите, что код для ввода

- ✓ вклада,
- ✓ процентной ставки и
- ✓ срока

практически один и тот же. Это наблюдение дает еще одну точку для рефакторинга.

Book 1.indb 163 25.09.2008 20:53:26

Все перечисленное позволяет выполнить рефакторинг программы CalculateInterestTable и создать программу CalculateInterestTabl eWithMethods:

```
// CalculateInterestTableWithMethods
// Генерация таблицы роста вклада по тому же алгоритму, что
// и в ранее рассматривавшихся программах, однако в этой
// программе работа распределена между несколькими
// методами.
using System;
namespace CalculateInterestTableWithMethods
{
  public class Program
  {
    public static void Main(string[] args)
    {
      // Раздел 1 - ввод данных, необходимых для создания
      // таблицы
      decimal principal = 0;
      decimal interest = 0;
      decimal duration = 0;
      InputInterestData(ref principal,
                        ref interest,
                        ref duration);


      // Раздел 2 - проверка введенных данных путем вывода
      // их пользователю на экран
      Console.WriteLine();  // Пропуск строки
      Console.WriteLine("Вклад                = " +
                        principal);
      Console.WriteLine("Процентная ставка = " +
                        interest + "%");
      Console.WriteLine("Срок                  = " +
                        duration + " лет");
      Console.WriteLine();

      // Раздел 3 - вывод таблицы вкладов по годам
      OutputInterestTable(principal, interest, duration);

      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
    }

    // InputInterestData - клавиатурный ввод вклада,
    // процентной ставки и срока для расчета таблицы
    // Этот метод реализует раздел 1, разбивая его на три
    // компонента
    public static
         void InputInterestData(ref decimal principal,
                                ref decimal interest,
                                ref decimal duration)
    {
```


Book 1.indb 164 25.09.2008 20:53:26

```
   // 1а Получение вклада
   principal = InputPositiveDecimal("вклад");

   // 1б Получение процентной ставки
   interest = InputPositiveDecimal("процентная ставка");

   // 1в Получение срока
   duration = InputPositiveDecimal("срок");
}


// InputPositiveDecimal возвращает положительное число
// типа decimal с клавиатуры
// Выполняется только одна проверка - на
// неотрицательность введенного значения
public static
     decimal InputPositiveDecimal(string prompt)
{
   // Цикл выполняется, пока не будет введено верное
   // значение
   while(true)
   {
      // Приглашение для ввода
      Console.Write("Введите " + prompt + ":");

      // Получение значения типа decimal с клавиатуры
      string input = Console.ReadLine();
      decimal value = Convert.ToDecimal(input);

      // Выход из цикла при вводе корректного значения
      if (value >= 0)
      {
         // Возврат введенного значения
         return value;
      }

      // В противном случае генерируется и выводится
      // сообщение об ошибке
      Console.WriteLine(prompt +
           " не может иметь отрицательное значение");
      Console.WriteLine("Попробуйте еще раз");
      Console.WriteLine();
   }
}

// OutputInterestTable для заданных значений вклада,
// процентной ставки и срока генерирует и выводит на
// экран таблицу роста вклада
// Реализация раздела 3 основной программы
public static
     void OutputInterestTable(decimal principal,
                              decimal interest,
                              decimal duration)
{
   for (int year = 1; year <= duration; year++)
```


Book 1.indb 165 25.09.2008 20:53:26

```
        {
            // Вычисление начисленных процентов
            decimal interestPaid;
            interestPaid = principal * (interest / 100);

            // Вычисление значения нового вклада путем
            // добавления начисленных процентов к основному
            // вкладу
            principal = principal + interestPaid;

            // Округление вклада до копеек
            principal = decimal.Round(principal, 2);

            // Вывод результата
            Console.WriteLine(year + "-" + principal);
        }
    }
  }
}
```

Раздел `Main()` состоит из трех очевидных частей, каждая из которых снабжена комментарием, выделенным полужирным шрифтом. Кроме того, раздел 1, в свою очередь, поделен на три подраздела — 1а, 1б и 1в.

Вам не следует пытаться выделять в своих исходных текстах комментарии полужирным шрифтом или указывать номера разделов. Исходный текст реальной программы — и без того сложная и запутанная штука, чтобы вносить в него искусственные усложнения. На практике для понимания достаточно ясных и информативных имен методов, указывающих их назначение.

В разделе 1 для ввода значений трех переменных, необходимых для работы программы (`principal`, `interest` им `duration`), вызывается метод `InputInterestData()`. В разделе 2 полученные значения выводятся на экран так же, как и в предыдущих версиях программы. В разделе 3 строится и выводится на экран таблица вкладов с помощью метода `OutputInterestTable()`.

Начнем с конца, с метода `OutputInterestTable()`. В нем содержится цикл, в котором выполняется вычисление начисленных процентов, точно так, как в программе `CalculateInterestTable` из главы 4, “Управление потоком выполнения”. Преимущество данной версии заключается в том, что при разработке этой части кода не нужно сосредоточиваться на деталях ввода и верификации данных. При написании этого метода следует просто думать о том, как вычислить и вывести таблицу для уже полученных значений. После выполнения метода управление вернется в строку, следующую за вызовом метода `OutputInterestTable()`.

`OutputInterestTable()` — хороший повод для того, чтобы воспользоваться новым меню Refactoring в Visual Studio 2008. Для этого выполните следующие действия.

1. **Воспользуйтесь в качестве отправной точки примером `CalculateInterestTableMoreForgiving` из главы 4, “Управление потоком выполнения”, выбрав исходный текст от объявления переменной `year` до конца цикла `while`:**

Book 1.indb 166 25.09.2008 20:53:26

```
int year = 0;                // Переменная цикла
while(year <= duration) // и весь цикл while
{
     //...
}
```

2. **Выберите команду меню Refactor⇨Extract Method.**
3. **В диалоговом окне Extract Method введите `OutputInterestTable`, просмотрите поле Preview Method Signature и щелкните на кнопке OK.**

   Обратите внимание на то, что предложенная сигнатура нового метода начинается с ключевых слов `private static` и включает `principal`, `interest` и `duration` в круглых скобках. О ключевом слове `private` как альтернативе `public` будет рассказано в главе 11, "Класс — каждый сам за себя". Пока же вы можете при желании сделать этот метод `public`.

```
private static decimal
     OutputInterestTable(decimal principal,
                         decimal interest, int duration)
```

4. **Щелкните сначала на кнопке OK, а затем — на Apply для завершения рефакторинга.**

   Результат такого рефакторинга заключается в следующем.

   - Ниже метода `Main()` добавляется новый `private static` метод `OutputInterestTable()`.
   - Там, где в `Main()` находился выбранный код, появляется следующая строка:

```
principal = OutputInterestTable(principal,
                                 interest, duration);
```

Точно такой же подход "разделяй и властвуй" применим и для метода `InputInterestData()`. Однако в этом случае требуется более сложный рефакторинг, так что я выполнил его вручную и все его этапы здесь не показаны. Все же искусство рефакторинга выходит за рамки настоящей книги.

В методе `InputInterestData()` вы сосредоточиваетесь только на вводе трех значений типа `decimal`. В данном случае, несмотря на три различные переменные, действия по их вводу идентичны и могут быть размещены в методе `InputPositiveDecimal()`, который одинаково применим как для ввода вклада, так и для ввода процентной ставки и срока, для которого выполняется расчет. Заметьте, что три цикла `while` в исходной программе превратились в один в теле метода `InputPositiveDecimal()`. Тем самым устранено дублирование кода, которое всегда нежелательно.

Метод `InputPositiveDecimal()` выводит приглашение и ожидает ввода пользователя. Если введенное пользователем значение неотрицательно, он возвращает его вызвавшему его методу. Если же введенное значение отрицательно, метод выводит сообщение об ошибке и повторяет цикл ввода.

С точки зрения пользователя, получается та же программа, что и раньше (в главе 4, "Управление потоком выполнения"), поскольку работает она точно так же:

```
Введите вклад: 100
Введите процентную ставку: -10
Процентная ставка не может быть отрицательной
Попробуйте еще раз
```


Book 1.indb 167 25.09.2008 20:53:26

```
Введите процентную ставку: 10
Введите срок: 10

Вклад             = 100
Процентная ставка = 10%
Срок              = 10 лет

1-110.0
2-121.00
3-133.10
4-146.41
5-161.05
```

Зачем беспокоиться о функциях?

Когда концепция функции появилась в 1950-е годы в Фортране, ее единственной целью было избежать дублирования кода. Предположим, вы пишете программу, которая должна вычислять и выводить на экран некоторое отношение во многих местах. В этом случае программа может просто вызывать в этих местах функцию `DisplayRatio()`, позволяющую избежать дублирования кода. Такая экономия может показаться не слишком большой, если функция состоит всего из пары строк, но функции бывают разные; они могут быть очень сложными и большими. Кроме того, распространенные функции наподобие `WriteLine()` могут использоваться в сотнях различных мест.

Второе преимущество применения функций также очевидно: проще корректно написать и отладить одну функцию, чем десяток фрагментов кода, и вдвойне проще сделать это, если функция невелика. Функция `DisplayRatio()` включает проверку того, что знаменатель в отношении не равен нулю. Если у вас имеется множество фрагментов кода, а не одна функция, то, скорее всего, в некоторых местах программы вы просто забудете вставить эту проверку.

Менее очевидно третье преимущество: хорошо спроектированные функции снижают сложность программы. Каждая функция должна соответствовать некоторой концепции. Вы должны быть способны указать назначение каждой функции без использования слов *и* и *или*. Вы должны следовать принципу “одна функция — одна задача”.

Функция наподобие `calculateSin()` служит идеальным примером. Программист, реализуя сложные вычисления, совершенно не должен беспокоиться, как именно будут применены их результаты. Прикладной программист может использовать функцию `calculateSin()`, не интересуясь, как именно она устроена и работает. Этот подход существенно снижает количество вещей, о которых должен помнить прикладной программист. Большую работу гораздо проще сделать, если разделить ее на части.

Большие программы, как, например, текстовый редактор, строятся из множества функций разного уровня абстракции. Например, функция `RedisplayDocument()` должна вызывать функцию `Reparagraph()` для вывода абзацев документа. Эта функция, в свою очередь, должна вызывать функцию `CalculateWordWrap()` для вычисления длин отдельных строк абзаца. Функция `CalculateWordWrap()` может вызывать функцию `LookUpWordBreak()`, определяющую, как должно быть разбито для переноса слово, стоящее в конце строки. Каждая из перечисленных функций решает одну задачу, которую можно сформулировать простым предложением (кстати, обратите внимание и на информативность названий функций).

Без возможности *абстрагирования* сложных концепций написание программы даже средней сложности становится практически нереализуемым, не говоря уже о создании операционных систем, игр, офисного программного обеспечения и тому подобных больших и сложных программ.

Само собой разумеется, все сказанное здесь о функциях в C# относится и к методам.

Book 1.indb 168 25.09.2008 20:53:26

```
6-177.16
7-194.88
8-214.37
9-235.81
10-259.39
Нажмите <Enter> для завершения программы...
```

Итак, взяв длинную, запутанную программу и применив рефакторинг, можно получить программу меньшего размера, со сниженным дублированием кода, а главное — более понятную.

Аргументы метода

Метод, подобный приведенному ниже, полезен примерно так же, как и зубная щетка, которой может пользоваться только один человек. Это связано с тем, что никакие данные приведенному методу не передаются и им не возвращаются:

```
public static void Output()
{
     Console.WriteLine("Это метод");
}
```

Сравним этот пример с реальными методами. Например, метод вычисления синуса требует определенных входных данных (в конце концов, вы ведь вычисляете синус чего-то?). Аналогично при конкатенации двух строк нужно передать методу две строки (не так ли?) и получить от метода результаты его работы. Следовательно, возникает крайняя необходимость в механизме обмена информацией с методом.

Передача аргументов методу

Значения, передаваемые методу, называются *аргументами метода* (другое часто используемое название — *параметры*). Большинство методов требуют для работы аргументов определенного типа. Вы передаете аргументы методу, перечисляя их в скобках после его имени. Проанализируем следующее небольшое дополнение к рассматривавшемуся ранее классу `Example`:

```
public class Example
{
  public static void Output(string funcString)
  {
    Console.WriteLine("Метод Output() получил аргумент: "
                      + funcString);
  }
}
```

Этот метод можно вызвать в самом классе следующим образом:

```
Output("Hello");
```

В результате можно получить вывод на экран:

```
Метод Output() получил аргумент: Hello
```

Программа передает методу `Output()` ссылку на строку `"Hello"`. Метод получает эту строку и присваивает ей имя `funcString`. В теле метода `Output()` переменная `funcString` может использоваться точно так, как любая другая переменная типа `string`.

Book 1.indb 169 25.09.2008 20:53:26

Можно немного изменить пример:

```
string myString = "Hello";
Output(myString);
```

В этом фрагменте переменной `myString` присваивается ссылка на строку `"Hello"`. Вызов `Output(myString)` передает методу объект, на который ссылается переменная `myString`, т.е. ту же строку `"Hello"`, что и ранее. Этот процесс изображен на рис. 8.1. Результат работы фрагмента исходного текста тот же, что и до внесения в него изменений.

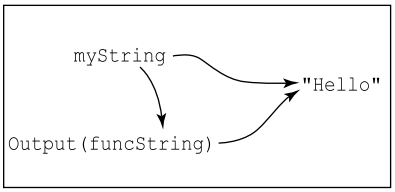

Рис. 8.1. Вызов `Output(myString)` копирует значение `myString` в переменную `funcString`

Передача методу нескольких аргументов

Когда я прошу сына помыть машину, он приводит сразу несколько аргументов, почему он не может это сделать. Но речь сейчас пойдет не о детях, а о нескольких аргументах, которые могут использоваться при вызове метода.

Вы можете определить метод с несколькими аргументами различных типов. Рассмотрим в качестве примера метод `AverageAndDisplay()`:

```
// AverageAndDisplay
using System;

namespace Example
{
   public class Program
   {
      public static void Main(string[] args)
      {
         // Обращение к методу-члену
         AverageAndDisplay("оценки 1", 3.5, "оценки 2", 4.0);

         // Ожидаем подтверждения пользователя
         Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
         Console.Read();
      }

      // AverageAndDisplay усредняет два числа и выводит
      // результат с использованием переданных меток
      public static
           void AverageAndDisplay(string s1, double d1,
                                  string s2, double d2)
```


Book 1.indb 170 25.09.2008 20:53:26

```
    {
      double average = (d1 + d2) / 2;
      Console.WriteLine("Среднее "   + s1
                        + ", равной " + d1
                        + ", и "        + s2
                        + ", равной " + d2
                        + ", равно "   + average);
    }
  }
}
```

Вот как выглядит вывод этой программы на экран:

```
Среднее оценки 1, равной 3.5, и оценки 2, равной 4, равно 3.75
Нажмите <Enter> для завершения программы...
```

Метод `AverageAndDisplay()` объявлен с несколькими аргументами в том порядке, в котором они в нее передаются.

Как обычно, выполнение программы начинается с первой команды после `Main()`. Первая строка после `Main()`, не являющаяся комментарием, вызывает метод `AverageAndDisplay()`, передавая ему две строки и два значения типа `double`.

Метод `AverageAndDisplay()` вычисляет среднее переданных значений типа `double`, `d1` и `d2`, переданных в метод вместе с их именами (содержащимися в переменных `s1` и `s2`), и сохраняет полученное значение в переменной `average`.

Изменение значений аргументов внутри метода может привести к ошибкам. Разумнее присвоить эти значения временным переменным и модифицировать уже их.

Соответствие определений аргументов их использованию

Каждый аргумент в вызове метода должен соответствовать определению метода как в смысле типа, так и в смысле порядка. Приведенный далее исходный текст некорректен и вызывает ошибку в процессе компиляции.

```
// AverageWithCompilerError - эта версия не компилируется!
using System;

namespace Example
{
  public class Program
  {
    public static void Main(string[] args)
    {
      // Обращение к методу-члену
      AverageAndDisplay("оценки 1", "оценки 2", 3.5, 4.0);

      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
    }

    // AverageAndDisplay усредняет два числа и выводит
    // результат с использованием переданных меток
```


Book 1.indb 171 25.09.2008 20:53:26

```
    public static
         void AverageAndDisplay(string s1, double d1,
                                string s2, double d2)
    {
      double average = (d1 + d2) / 2;
      Console.WriteLine("Среднее "   + s1
                        + ", равной " + d1
                        + ", и "        + s2
                        + ", равной " + d2
                        + ", равно "   + average);
    }
  }
}
```

Язык C# обнаруживает несоответствие типов передаваемых методу аргументов с аргументами в определении метода. Строка `"оценки 1"` соответствует типу `string` в определении метода; однако согласно определению метода вторым аргументом должно быть число типа `double`, в то время как при вызове вторым аргументом метода оказывается строка `string`.

Это случилось потому, что я просто переставил местами второй и третий аргументы метода. И это как раз то, за что я не люблю компьютеры (именно за то, что они понимают все совершенно буквально).

Чтобы исправить ошибку, достаточно переставить местами второй и третий аргументы в вызове метода `AverageAndDisplay()`.

Перегрузка метода

В одном классе может быть два метода с одним и тем же именем — *при условии различия их аргументов*. Это явление называется *перегрузкой* (overloading) имени метода.

Вот пример программы, демонстрирующей перегрузку:

```
// AverageAndDisplayOverloaded демонстрирует возможность
// перегрузки метода вычисления и вывода среднего значения
using System;

namespace AverageAndDisplayOverloaded
{
  public class Program
  {
    public static void Main(string[] args)
    {
      // Вызов первого метода-члена
      AverageAndDisplay("моей оценки", 3.5,
                        "твоей оценки", 4.0);
      Console.WriteLine();

      // Вызов второго метода-члена
      AverageAndDisplay(3.5, 4.0);

      // Ожидаем подтверждения пользователя
      Console.WriteLine("Нажмите <Enter> для " +
                        "завершения программы...");
      Console.Read();
```


Book 1.indb 172 25.09.2008 20:53:27

```
    }

    // AverageAndDisplay - вычисление среднего значения двух
    // чисел и его вывод на экран с переданными методу
    // метками этих чисел
    public static
         void AverageAndDisplay(string s1, double d1,
                                string s2, double d2)
    {
       double average = (d1 + d2) / 2;
       Console.WriteLine("Среднее "   + s1
                         + ", равной " + d1);
       Console.WriteLine("и "          + s2
                         + ", равной " + d2
                         + ", равно "   + average);
    }
    public static void AverageAndDisplay(double d1,
                                          double d2)
    {
       double average = (d1 + d2) / 2;
       Console.WriteLine("среднее " + d1
                         + " и "        + d2
                         + " равно "   + average);
    }
  }
}
```

В программе определены две версии метода `AverageAndDisplay()`. Программа вызывает одну из них после другой, передавая им соответствующие аргументы. C# в состоянии определить по переданным методу аргументам, какую из версий следует вызвать, сравнивая типы передаваемых значений с определениями методов. Программа корректно компилируется и выполняется, выводя на экран следующие строки:

```
Среднее моей оценки, равной 3.5,
и твоей оценки, равной 4, равно 3.75

Среднее 3.5 и 4 равно 3.75
Нажмите <Enter> для завершения программы...
```

Вообще говоря, C# не позволяет иметь в одной программе два метода с одинаковыми именами. В конце концов, как тогда он сможет разобраться, какой из методов следует вызывать? Но дело в том, что C# в имя метода во внутреннем представлении компилятора включает не только имя метода, но и количество и типы его аргументов. Поэтому C# в состоянии отличить метод `AverageAndDisplay(string,double,string,double)` от метода `AverageAndDisplay(double,double)`. Если рассматривать эти методы с их аргументами, становится очевидным, что они разные.

Реализация аргументов по умолчанию

Зачастую желательно иметь две (или более) версии метода, имеющие следующие отличия.

- ✓ Один из методов представляет собой более сложную версию, обеспечивающую б*о*льшую гибкость, но требующую большого количества аргументов от вызывающей программы, причем некоторые из них могут быть просто непонятны пользователю.

Book 1.indb 173 25.09.2008 20:53:27

Под пользователем метода зачастую подразумевается программист, применяющий ее в своих программах, так что пользователь метода и пользователь готовой программы — это разные люди. Еще один термин, применяемый для обозначения такого рода пользователя — клиент.

- ✓ Вторая версия метода гораздо проще для применения. Она работает так же, как и первая, в которой часть аргументов принимает некоторые предопределенные значения по умолчанию.

Такое поведение легко реализуется с использованием перегрузки методов.

Рассмотрим следующую пару методов `DisplayRoundedDecimal()`:

```
// MethodsWithDefaultArguments - две версии одного и того же
// метода, причем одна из них представляет версию второй с
// использованием значений аргументов по умолчанию

using System;

namespace MethodsWithDefaultArguments
{
   public class Program
   {
      public static void Main(string[] args)
      {
         // Вызов метода-члена
         Console.WriteLine("{0}",
                   DisplayRoundedDecimal(12.345678M, 3));

         // Ожидаем подтверждения пользователя
         Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
         Console.Read();
      }

      // DisplayRoundedDecimal преобразует значение типа
      // decimal в строку с определенным количеством значащих
      // цифр
      public static
           string DisplayRoundedDecimal(decimal value,
                    int nNumberOfSignificantDigits)
      {
         // Сначала округляем число до указанного количества
         // значащих цифр
         decimal mRoundedValue =
                      decimal.Round(value,
                                    nNumberOfSignificantDigits);
         // и преобразуем его в строку
         string s = Convert.ToString(mRoundedValue);

         return s;
      }
      public static
           string DisplayRoundedDecimal(decimal value)
```


Book 1.indb 174 25.09.2008 20:53:27

```
    {
      // Вызываем DisplayRoundedDecimal(decimal, int) с
      // указанием количества значащих цифр по умолчанию
      string s = DisplayRoundedDecimal(value, 2);
      return s;
    }
  }
}
```

Метод `DisplayRoundedDecimal()` преобразует значение типа `decimal` в значение типа `string` с определенным количеством значащих цифр после десятичной точки. Поскольку числа типа `decimal` часто применяются в финансовых расчетах, наиболее распространенными будут вызовы этого метода со вторым аргументом, равным 2. В анализируемой программе это предусмотрено, и вызов `DisplayRoundedDecimal()` с одним аргументом округляет значение этого аргумента до двух цифр после десятичной точки, позволяя пользователю не беспокоиться о смысле и числовом значении второго аргумента метода.

Обратите внимание на то, что версия метода `DisplayRoundedDecimal(decimal)` в действительности вызывает метод `DisplayRoundedDecimal(decimal,int)`. Такая практика позволяет избежать ненужного дублирования кода. Обобщенная версия метода может использовать существенно большее количество аргументов, которые ее разработчик может даже не включить в документацию.

Аргументы по умолчанию не просто сберегают силы ленивого программиста. Программирование — работа, требующая высочайшей степени концентрации, и излишние аргументы метода, для выяснения назначения и рекомендуемых значений которых необходимо обращаться к документации, затрудняют программирование, приводят к перерасходу времени и повышают вероятность внесения ошибок в код. Автор метода хорошо понимает взаимосвязи между аргументами метода и способен обеспечить несколько корректных перегруженных версий, более дружественных к клиенту.

Примечание для программистов на Visual Basic и C/C++. В C# единственный способ реализации аргументов по умолчанию — перегрузка метода. C# не позволяет иметь необязательные аргументы.

Передача в метод типов-значений

Базовые типы переменных, такие как `int`, `double` и `decimal`, известны как *типы-значения*. Переменные таких типов могут быть переданы в метод одним из двух способов. По умолчанию эти переменные передаются в метод *по значению* (by value); альтернативный способ передачи — передача по ссылке.

Программисты часто не совсем точны в употреблении терминов. Если речь идет о типах-значениях, то когда программист говорит о “передаче переменной в метод”, это обычно означает “передача значения переменной в метод”.

Передача по значению

В отличие от ссылок на объекты, переменные с типами-значениями наподобие `int` или `double` обычно передаются в метод *по значению*, т.е. методу передается значение, содержащееся в этой переменной, но не сама переменная.

Book 1.indb 175 25.09.2008 20:53:27

При такой передаче изменение значения соответствующей переменной внутри метода не вызовет изменения значения переданной переменной в вызывающей программе, что и демонстрируется в следующей программе:

```
// PassByValue - программа для демонстрации семантики
// передачи аргумента по значению
using System;

namespace PassByValue
{
   public class Program
   {
      // Update - метод пытается модифицировать значения
      // аргументов, переданные ему; обратите внимание на то,
      // что методы в классе могут быть объявлены в любом
      // порядке
      public static void Update(int i, double d)
      {
         i = 10;
         d = 20.0;
      }

      public static void Main(string[] args)
      {
         // Объявляем и инициализируем две переменные
         int i = 1;
         double d = 2.0;
         Console.WriteLine("Перед вызовом Update(int,double):");
         Console.WriteLine("i = " + i + ", d = " + d);

         // Вызываем метод
         Update(i, d);

         // Обратите внимание на то, что значения 1 и 2.0 не
         // изменились
         Console.WriteLine("После вызова Update(int,double):");
         Console.WriteLine("i = " + i + ", d = " + d);

         // Ожидаем подтверждения пользователя
         Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
         Console.Read();
      }
   }
}
```

В результате выполнения программы на экран выводится следующее:

```
Перед вызовом Update(int,double):
i = 1, d = 2
После вызова Update(int,double):
i = 1, d = 2
Нажмите <Enter> для завершения программы...
```

Вызов `Update()` передает методу *значения* 1 и 2.0, а не ссылки на переменные `i` и `d`. Таким образом, изменение их значений в методе никак не влияет на значения исходных переменных в вызывающей программе.

Book 1.indb 176 25.09.2008 20:53:27

Передача по ссылке

Передача методу переменных с типами-значениями по ссылке имеет ряд преимуществ — этот метод передачи используется, в частности, когда вызывающая программа решает предоставить методу возможность изменять значение передаваемой в качестве аргумента переменной. Приведенная далее программа `PassByReference` демонстрирует эту возможность.

Язык C# позволяет программисту передавать аргументы по ссылке при помощи ключевых слов `ref` и `out`. В несколько измененной программе `PassByValue` демонстрируется, как это можно делать:

```
// PassByReference - программа для демонстрации семантики
// передачи аргумента по ссылке
using System;

namespace PassByValue
{
   public class Program
   {
      // Update - метод пытается модифицировать значения
      // аргументов, переданные ему; обратите внимание на
      // передачу аргументов как ref и out
      public static void Update(ref int i, out double d)
      {
         i = 10;
         d = 20.0;
      }

      public static void Main(string[] args)
      {
         // Объявляем две переменные и одну инициализируем
         int i = 1;
         double d;
         Console.WriteLine("Перед вызовом " +
                           "Update(ref int, out double):");
         Console.WriteLine("i = " + i +
                           ", d неинициализирована");

         // Вызываем метод
         Update(ref i, out d);

         // Обратите внимание на то, что значение i равно 10,
         // значение d равно 20.0
         Console.WriteLine("После вызова "
                           "Update(ref int, out double):");
         Console.WriteLine("i = " + i + ", d = " + d);

         // Ожидаем подтверждения пользователя
         Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
         Console.Read();
      }
   }
}
```


Book 1.indb 177 25.09.2008 20:53:27

Ключевое слово `ref` указывает C#, что в метод следует передать ссылку на `i`, а не просто значение этой переменной. Следовательно, изменения, выполненные с этой переменной, оказываются экспортированы обратно вызывающей программе.

Аналогично ключевое слово `out` говорит: "Передай ссылку на эту переменную, но можешь никак не заботиться о ее значении, поскольку оно все равно не будет использоваться и будет перезаписано в процессе работы метода". Это ключевое слово годится тогда, когда переменная применяется исключительно для того, чтобы вернуть значение вызывающей программе.

В результате выполнения рассмотренной программы `PassByReference` на экран выводится следующее:

```
Перед вызовом Update(ref int, out double):
i = 1, d неинициализирована
После вызова Update(ref int, out double):
i = 10, d = 20
Нажмите <Enter> для завершения программы...
```

Аргумент, передаваемый как `out`, всегда считается передаваемым так же, как `ref`, т.е. `ref out` — это тавтология. Кроме того, при передаче аргументов по ссылке всегда следует передавать только *переменные* — передача литеральных значений, например просто числа 2 вместо переменной типа `int`, в этом случае приводит к ошибке.

Обратите внимание на то, что начальные значения `i` и `d` переписываются в методе `Update()`. После возврата в метод `Main()` эти переменные остаются с измененными в методе `Update()` значениями. Сравните это поведение с поведением программы `PassByValue`, в которой внесенные изменения не сохраняются при выходе из метода.

Не передавайте переменную по ссылке дважды

Никогда не передавайте по ссылке одну и ту же переменную дважды в один метод, поскольку это может привести к неожиданным побочным эффектам. Описать эту ситуацию труднее, чем просто продемонстрировать на примере программы. Внимательно взгляните на метод `Update()` в приведенном листинге:

```
// PassByReferenceError - демонстрация потенциально
// ошибочной ситуации при вызове метода с передачей
// аргументов по ссылке
using System;

namespace PassByReferenceError
{
   public class Program
   {
      // Update пытается изменить значения
      // переданных ей аргументов
      public static void DisplayAndUpdate(ref int var1,
                                          ref int var2)
      {
         Console.WriteLine("Начальное значение var1 - " +
                           var1);
         var1 = 10;
```


Book 1.indb 178 25.09.2008 20:53:27

```
        Console.WriteLine("Начальное значение var2 - " +
                          var2);
        var2 = 20;
      }

      public static void Main(string[] args)
      {
        // Объявляем и инициализируем переменную
        int n = 1;
        Console.WriteLine("Перед вызовом " +
                          "Update(ref n, ref n):");
        Console.WriteLine("n = " + n);
        Console.WriteLine();

        // Вызываем метод
        DisplayAndUpdate(ref n, ref n);

        // Обратите внимание на то, как изменяется значение n
        // - не так, как ожидалось от этой переменной и
        // метода, в которую она была передана
        Console.WriteLine();
        Console.WriteLine("После вызова " +
                          "Update(ref n, ref n):");
        Console.WriteLine("n = " + n);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
      }
    }
}
```

В этом примере метод `Update(ref int, ref int)` объявлен как метод с двумя целыми аргументами, передаваемыми по ссылке — и в этом нет ничего некорректного или необычного. Проблема возникает тогда, когда в метод передается *одна и та же переменная* в качестве как первого, так и второго аргументов. Внутри метода происходит изменение переменной `var1`, которая ссылается на переменную `n`, от ее начального значения 1 до значения 10. Затем происходит изменение переменной `var2`, которая также ссылается на переменную `n`, от ее начального значения 1 до значения 20, и побочное действие заключается в том, что переменная `n` теряет ранее присвоенное ей значение 10 и получает новое значение 20.

Это видно из приведенного ниже вывода программы на экран:

```
Перед вызовом Update(ref n, ref n):
n = 1

Начальное значение var1 - 1
Начальное значение var2 - 10

После вызова Update(ref n, ref n):
n = 20
Нажмите <Enter> для завершения программы...
```


Book 1.indb 179 25.09.2008 20:53:27

Понятно, что ни программист, который писал метод `Update()`, ни программист, его использовавший, не рассчитывали на такой экзотический результат. Вся проблема в том, что одна и та же переменная была передана в один и тот же метод по ссылке больше одного раза. Никогда так не поступайте, если только вы не абсолютно точно знаете, чего именно вы хотите добиться таким действием.

Вы можете передавать одно и то же значение в метод сколько угодно раз, если передаете его по значению.

Возврат значений из метода

Многие реальные операции создают значения, которые должны быть возвращены тому, кто вызвал эти операции. Например, метод `sin()` получает аргумент и возвращает значение тригонометрической функции "синус" для данного аргумента. Метод может

Почему некоторые аргументы используются только для возврата значений

Язык C# по мере возможности старается защитить программиста от всех глупостей, которые он вольно или невольно может сделать. Одна из этих глупостей — забыть проинициализировать переменную перед ее первым применением (особенно часто это случается с переменными-счетчиками). C# генерирует ошибку, когда вы пытаетесь использовать объявленную, но не инициализированную переменную:

```
int variable;
Console.WriteLine("Это ошибка " + variable);
variable = 1;
Console.WriteLine("А это - нет " + variable);
```

Однако C# не в состоянии отслеживать переменные в методе:

```
void SomeMethod(ref int variable)
{
     Console.WriteLine("Ошибка или нет? " + variable);
}
```

Откуда метод `SomeMethod()` может знать, была ли переменная `variable` инициализирована перед вызовом метода? Это невозможно. Вместо этого C# отслеживает переменные при вызове метода; например, такой вызов метода приведет к сообщению об ошибке:

```
int uninitializedVariable;
SomeMethod(ref uninitializedVariable);
```

Если бы C# позволил осуществить такой вызов, то метод `SomeMethod()` получил бы ссылку на неинициализированную переменную (т.е. на *мусор* (garbage)) в памяти. Ключевое слово `out` позволяет методу и вызывающему его коду договориться о том, что передаваемая по ссылке переменная может быть не инициализирована — метод обещает не использовать ее значение до тех пор, пока оно не будет каким-либо образом присвоено самим методом. Следующий фрагмент исходного текста компилируется без каких-либо замечаний:

```
int uninitializedVariable;
SomeMethod(out uninitializedVariable);
```


Book 1.indb 180 25.09.2008 20:53:27

вернуть значение вызывающему методу двумя способами. Наиболее распространенный — с помощью команды `return`; при втором способе используются возможности передачи аргументов по ссылке.

Возврат значения оператором `return`

В приведенном далее фрагменте исходного текста демонстрируется небольшой метод, возвращающий среднее значение переданных ему аргументов:

```
public class Example
{
     public static double Average(double d1, double d2)
     {
          double average = (d1 + d2) / 2;
          return average;
     }
     public static void Test()
     {
          double v1 = 1.0;
          double v2 = 3.0;
          double averageValue = Average(v1, v2);
          Console.WriteLine("Среднее для " + v1
                            + " и " + v2 + " равно "
                            + averageValue);
          // Такой метод также вполне работоспособен
          Console.WriteLine("Среднее для " + v1
                            + " и " + v2 + " равно "
                            + Average(v1, v2));
     }
}
```

Прежде всего, обратите внимание на то, что метод объявлен как `public static d ouble Average()`: тип `double` перед именем метода указывает на тот факт, что метод `Average()` возвращает вызывающему методу значение типа `double`.

Метод `Average()` использует имена `d1` и `d2` для значений, переданных ему в качестве аргументов. Он создает переменную `average`, которой присваивает среднее значение этих переменных. Затем значение, содержащееся в переменной `average`, возвращается вызывающему методу.

Программисты иногда говорят, что “метод возвращает `average`”. Это некорректное выражение. Говорить, что передается или возвращается `average` или иная переменная, — неточно. В данном случае вызывающему методу возвращается значение, содержащееся в переменной `average`.

Вызов `Average()` из метода `Test()` выглядит так же, как и вызов любого другого метода; однако значение типа `double`, возвращаемое методом `Average()`, сохраняется в переменной `averageValue`.

Метод, который возвращает значение (как, например, `Average()`), не может завершиться просто по достижении закрывающей фигурной скобки, поскольку C# совершенно непонятно, какое же именно значение должен будет вернуть этот метод? Для этого обязательно наличие оператора `return`.

Book 1.indb 181 25.09.2008 20:53:27

Возврат значения посредством передачи по ссылке

Метод может также вернуть одно или несколько значений вызывающей программе с помощью ключевых слов `ref` и `out`. Рассмотрим пример `Update()` из раздела "Передача в метод типов-значений" данной главы:

```
// Update - метод пытается модифицировать значения
// аргументов, переданные ему; обратите внимание на
// передачу аргументов как ref и out
public static void Update(ref int i, out double d)
{
  i = 10;
  d = 20.0;
}
```

Этот метод объявлен как `void`, поскольку он не возвращает никакого значения вызывающему методу. Однако, так как переменная `i` объявлена как `ref`, а переменная `d` — как `out`, любые изменения значений этих переменных, выполненные в методе `Update()`, сохранятся при возврате в вызывающий метод. Другими словами, значения этих переменных вернутся вызывающему методу.

Когда какой способ передачи использовать

Вы можете подумать так: "Метод может возвращать значение с помощью как оператора `return`, так и переменных, переданных по ссылке. Так какой же способ мне лучше применять в своих программах?" В конце концов, тот же метод `Average()` вы могли написать и так:

```
public class Example
{
     // Примечание. Параметр, передаваемый как 'out', лучше
     // сделать последним в списке
     public static void Average(double 1, double d2,
                                     out double results)
     {
          results = (d1 + d2) / 2;
     }
     public static void Test()
     {
          double v1 = 1.0;
          double v2 = 3.0;
          double averageValue;
          Average(averageValue, v1, v2);
          Console.WriteLine("Среднее " + v1
                                + " и " + v2 + " равно "
                                + averageValue);
     }
}
```

Обычно значение вызывающему методу возвращается с помощью оператора `return`, а не `out`-аргумента, хотя обосновать преимущество такого подхода очень трудно.

Иногда для возврата значения посредством передачи аргумента по ссылке необходимо выполнить дополнительные действия. Однако обычно эффективность — не главный фактор при принятии решения о способе возврата значения из метода.

Book 1.indb 182 25.09.2008 20:53:27

Как правило, “способ `out`” используется, если требуется вернуть из метода несколько значений, например, как в следующем методе:

```
public class Example
{
   public static
      void AverageAnproduct(double d1, double d2,
                                out double average,
                                out double product)
   {
      average = (d1 + d2) / 2;
      product = d1 * d2;
   }
}
```

Возврат из метода нескольких значений встречается не так часто, как может показаться. Метод, который возвращает несколько значений, обычно делает это путем возврата одного объекта класса, который инкапсулирует несколько значений, или путем возврата массива значений. Оба способа приводят к более ясному и понятному исходному тексту.

Определение метода без возвращаемого значения

Выражение `public static double Average()` объявляет метод `Average()` как возвращающий значение типа `double`. Однако существуют методы, не возвращающие ничего. Ранее вы сталкивались с примером такого метода, `AverageAndDisplay()`, который выводил вычисленное среднее значение на экран, ничего не возвращая вызывающему методу. Вместо того чтобы опустить в объявлении такого метода тип возвращаемого значения, в C# указывается `void`:

```
public void AverageAndDisplay(double, double)
```

Ключевое слово `void`, употребленное вместо имени типа, по сути, означает *отсутствие типа*, т.е. указывает, что метод `AverageAndDisplay()` ничего не возвращает вызывающему методу. (В C# любое объявление метода обязано указывать возвращаемый тип, даже если это `void`.)

Метод, который не возвращает значения, программистами называется *`void`-методом*, по использованному ключевому слову в его описании.

Методы, не являющиеся `void`-методами, возвращают управление вызывающему методу при выполнении оператора `return`, за которым следует возвращаемое вызывающему методу значение. Поскольку `void`-метод не возвращает никакого значения, выход из него осуществляется посредством оператора `return` без какого бы то ни было значения либо (по умолчанию) при достижении закрывающей тело метода фигурной скобки.

Рассмотрим следующий метод `DisplayRatio()`:

```
public class Example
{
   public static void DisplayRatio(double numerator,
                                    double denominator)
   {
      // Если знаменатель равен 0...
      if (denominator == 0.0)
```


Book 1.indb 183 25.09.2008 20:53:27

```
    {
      // ...вывести сообщение об ошибке и вернуть
      // управление вызывающему методу...
      Console.WriteLine("Знаменатель не может быть нулем");
      // Выход из метода
      return;
    }
    // Эта часть метода выполняется только в том случае,
    // когда знаменатель не равен нулю
    double ratio = numerator / denominator;
    Console.WriteLine("Отношение " + numerator
                      + " к " + denominator
                      + " равно " + ratio);
  } // Если знаменатель не равен нулю, выход из метода
}   // выполняется здесь
```

Метод `DisplayRatio()` начинает работу с проверки, не равно ли значение `denominator` нулю.

- ✓ Если значение `denominator` равно нулю, программа выводит сообщение об ошибке и возвращает управление вызывающему методу, не пытаясь вычислить значение отношения. При попытке вычислить отношение произошла бы ошибка деления на нуль с аварийным остановом программы в результате.
- ✓ Если значение `denominator` не равно нулю, программа выводит на экран значение отношения. При этом закрывающая фигурная скобка после вызова метода `WriteLine()` является закрывающей скобкой метода `DisplayRatio()` и, таким образом, представляет собой точку возврата из метода в вызывающую программу.

Нулевая ссылка и ссылка на нуль

Ссылочные переменные, в отличие от переменных типов-значений, при создании инициализируются значением по умолчанию `null`. Однако нулевая ссылка, т.е. ссылка, инициализированная значением `null`, — это не то же самое, что ссылка на нуль. Например, две следующие ссылки совершенно различны:

```
class Example
{
     int value;
}
// Создание нулевой ссылки ref1
Example ref1;
// Создание ссылки на нулевой объект
Example ref2 = new Example();
ref2.value = 0;
```

Переменная `ref1` пуста, как мой бумажник. Она указывает “в никуда”, т.е. не указывает ни на какой реальный объект. Ссылка же `ref2` указывает на вполне конкретный объект, значение которого равно нулю.

Возможно, эта разница станет понятнее после рассмотрения следующего примера:

```
string s1;
string s2 = "";
```


Book 1.indb 184 25.09.2008 20:53:27

По сути, возникает аналогичная ситуация: `s1` указывает на *нулевой объект*, в то время как `s2` — на *пустую строку* (на сленге программистов пустая строка иногда называется нулевой строкой). Это очень существенное отличие, как становится ясно из следующего исходного текста:

```
// Test - тестовая программа
namespace Test
{
   using System;
   public class Program
   {
      public static void Main(string[] strings)
      {
         Console.WriteLine("Эта программа исследует " +
                           "метод TestString()");
         Console.WriteLine();
         Example exampleObject = new Example();
         Console.WriteLine("Передача нулевого объекта:");
         string s = null;
         exampleObject.TestString(s);
         Console.WriteLine();
         // Теперь передаем в метод нулевую (пустую) строку
         Console.WriteLine("Передача пустой строки:");
         exampleObject.TestString("");
         Console.WriteLine();
         // Наконец, передаем реальную строку
         Console.WriteLine("Передача реальной строки:");
         exampleObject.TestString("test string");
         Console.WriteLine();
         // Ожидаем подтверждения пользователя
         Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
         Console.Read();
      }
   }
   class Example
   {
      public void TestString(string sTest)
      {
         // Проверка, не нулевой ли объект (эта проверка должна
         // быть первой!)
         if (sTest == null)
         {
            Console.WriteLine("sTest == null");
            return;
         }
         // Мы знаем, что sTest не указывает на нулевой объект,
         // но все еще может указывать на пустую строку.
         // Проверяем, не указывает ли sTest на нулевую
         // (пустую) строку
         if (String.Compare(sTest, "") == 0)
         {
            Console.WriteLine("sTest - ссылка на пустую строку");
            return;
```


Book 1.indb 185 25.09.2008 20:53:27

```
        }
        // Строка не пустая, выводим ее
        Console.WriteLine("sTest указывает на: '" + sTest +
"'");
     }
   }
}
```

Метод `TestString()` использует сравнение `sTest==null` для проверки, не на нулевой ли объект указывает ссылка. Для проверки, не указывает ли ссылка на пустую строку, метод `TestString()` использует метод `Compare()`. (Метод `Compare()` возвращает `0`, если переданные ему строки равны. В главе 6, "Работа со строками в C#", сравнение строк рассматривается более детально.)

Вот как выглядит вывод этой программы на экран:

```
Эта программа исследует метод TestString()
Передача нулевого объекта:
sTest == null
Передача пустой строки:
sTest - ссылка на пустую строку
Передача реальной строки:
sTest указывает на: 'test string'
Нажмите <Enter> для завершения программы...
```

Метод `WriteLine()`

Вы могли заметить, что метод `WriteLine()`, использовавшийся в рассматриваемых программах, представляет собой не более чем вызов метода класса `Console`:

```
Console.WriteLine("Это — вызов метода");
```

Метод `WriteLine()` — один из множества предопределенных методов, предоставляемых библиотекой .NET. `Console` — предопределенный класс, предназначенный для использования в консольных приложениях.

Аргументом метода `WriteLine()`, применявшимся в рассмотренных выше примерах, является строка `string`. Оператор `+` позволяет программисту собрать эту строку из нескольких строк или строк и переменных встроенных типов, например, так:

```
string s = "Маша";
Console.WriteLine("Меня зовут " + s +
                  " и мне " + 3 + " года");
```

В результате вы увидите выведенную на экран строку "`Меня зовут Маша и мне 3 года`".

Второй вид метода `WriteLine()` допускает наличие более гибкого множества аргументов, например:

```
Console.WriteLine("Меня зовут {0} и мне {1} года",
                  "Маша", 3);
```

Первый аргумент такого вызова называется форматной строкой. В данном примере строка `"Маша"` вставляется вместо символов `{0}` — нуль указывает на первый аргумент после командной строки. Целое число 3 вставляется в позицию, помеченную как `{1}`. Этот вид метода более эффективен, поскольку конкатенация строк не так проста, как это звучит, и не столь эффективна.

Кроме того, в данном варианте в форматной строке может использоваться ряд управляющих элементов, которые указывают, как именно должны выводиться аргументы метода `WriteLine()`. Вы встречались с ними в главе 6, "Работа со строками в C#".

Book 1.indb 186 25.09.2008 20:53:27