30 Возможности языка C# 3.0

C# 3.0, следующая версия ведущего .NET-языка программирования от Microsoft, представляет множество новых синтаксических конструкций. В этой главе описывается неявная типизация данных, расширяющие методы, инициализаторы объектов, анонимные типы и лямбда-выражения.

Хотя многие из этих новых элементов языка могут непосредственно использоваться для создания надежного и функционального программного обеспечения для .NET, стоит отметить, что многие из новых конструкций более полезны при взаимодействии с группой технологий LINQ, которые мы рассмотрим в главе 31. Учитывая этот факт, не беспокойтесь, если полезность некоторых из этих конструкций вначале покажется вам не очевидной. После того как вы поймете роль LINQ, многое прояснится.

ПРИМЕЧАНИЕ

На момент написания этой книги C# 3.0 находился в состоянии бета-версии. Поэтому имейте в виду, что материал, рассматриваемый в этой главе, в финальном выпуске языка может оказаться неактуальным.

Компилятор командной строки C# 3.0

Среда Visual Studio 2005 «жестко закодирована» (если хотите) на использование компилятора C# 2.0 для компиляции исходного кода в .NET-сборки. Однако при установке системы Community Technology Preview для LINQ (см. главу 26) устанавливаются несколько новых шаблонов проектов, которые используют предварительную версию компилятора C# 3.0 (рис. 30.1).

Эти шаблоны автоматически устанавливают ссылки на несколько LINQ-сборок, используют несколько LINQ-пространств имен и обычно предоставляют исходный LINQ-код. В этой главе мы будем создавать все наши примеры, применяя шаблон LINQ Console Application проекта, чтобы оценить преимущества интеграции с компилятором C# 3.0. Так что пока просто проигнорируйте все детали, специфичные для LINQ.

В качестве альтернативы можете явно компилировать примеры этой главы из командной строки с помощью компилятора C# 3.0. По умолчанию утилита csc.exe (версии 3.0) устанавливается в папку C:\Program Files\LINQ Preview\Bin. То есть

для получения доступа к этой версии компилятора откройте командную строку и перейдите в папку Bin, используя команду cd:

```
cd C:\Program Files\LINQ Preview\Bin
```

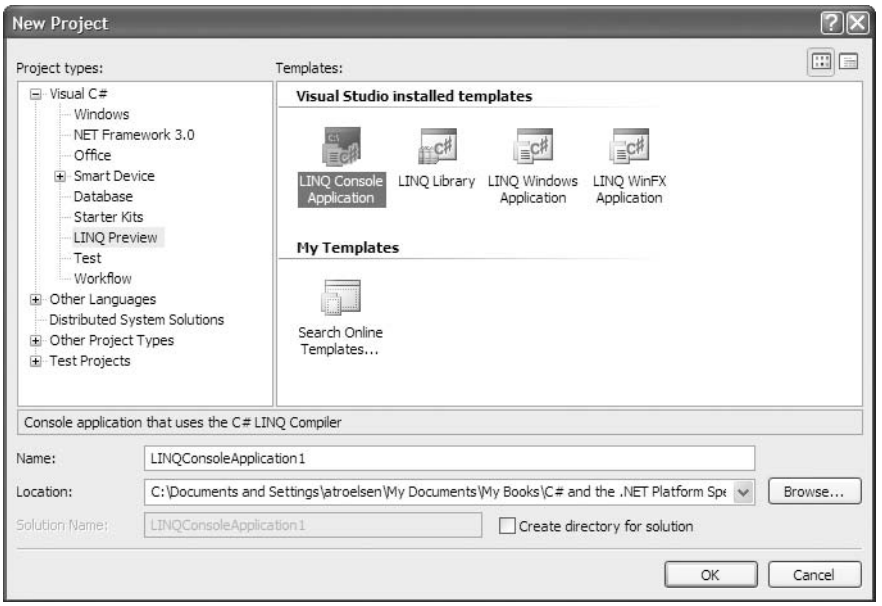

Рис. 30.1. Шаблоны LINQ-проектов в Visual Studio 2005 используют компилятор C# 3.0

Кроме того, чтобы упростить ввод аргументов командной строки, я предлагаю сохранять все файлы исходного кода в папке под названием CSharp3 на диске C. Таким образом, вы сможете компилировать код следующим образом (без какого бы то ни было изменения переменных окружения машины):

```
csc /out:C:\CSharp3\MyApp.exe C:\CSharp3\MyApp.cs
```

Давайте углубимся в новые возможности очередной версии языка программирования C#, а начнем мы с концепции неявно типизированных данных.

Неявно типизированные локальные переменные

Как вы узнали в самом начале этой книги, локальные переменные (например, объявляемые в области видимости метода) объявляются довольно предсказуемым образом:

```
static void Main()
{
    // Локальные переменные объявляются следующим образом:
    // dataType variableName = initialValue;
```

```
    int myInt = 0;
    bool myBool = true;
    string myString = "Time, marches on...";
    Console.ReadLine();
}
```

Теперь C# 3.0 предоставляет новое ключевое слово `var`, которое можно использовать вместо ввода формального типа данных (например, `int`, `bool` или `string`). В этом случае компилятор определяет нижележащий тип данных автоматически на основании значения, используемого для инициализации элемента данных. Например, предыдущий метод `Main()` можно написать следующим образом:

```
static void Main()
{
    // Неявно типизированные локальные переменные.
    var myInt = 0;
    var myBool = true;
    var myString = "Time, marches on...";
    Console.ReadLine();
}
```

В данном случае компилятор, учитывая присвоенное значение, может определить, что переменная `myInt` фактически имеет тип `System.Int32`, `myBool` — тип `System.Boolean`, а `myString` — тип `System.String`. Это можно проверить, если вывести имя типа посредством отражения:

```
static void Main()
{
    Console.WriteLine(«***** Fun with Implicit Typing *****\n»);

    // Неявно типизированные локальные переменные.
    var myInt = 0;
    var myBool = true;
    var myString = «Time, marches on...»;

    // Вывести нижележащие типы.
    Console.WriteLine(«myInt is a: {0}», myInt.GetType().Name);
    Console.WriteLine(«myBool is a: {0}», myBool.GetType().Name);
    Console.WriteLine(«myString is a: {0}», myString.GetType().Name);
    Console.ReadLine();
}
```

Более того, знайте, что неявную типизацию можно задействовать для любых типов из библиотек базовых классов, включая массивы, обобщения и пользовательские классы:

```
var evenNumbers = new int[] {2, 4, 6, 8};
var myMinivans = new List<MiniVan>();
var myCar = new SportsCar();
```

Если использовать отражение для этих неявно типизированных локальных переменных, вы увидите результат, показанный на рис. 30.2.

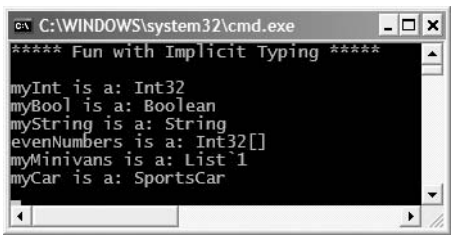

Рис. 30.2. Отражение неявно определенных локальных переменных

Ключевое слово `var` можно также использовать в конструкции цикла `foreach`. Как можно ожидать, компилятор корректно определит правильный тип:

```
// Использование ключевого слова «var» в стандартном цикле foreach.
var evenNumbers = new int[] { 2, 4, 6, 8 };

// Здесь var на самом деле имеет тип System.Int32.
foreach (var item in evenNumbers)
{
    Console.WriteLine("Item value: {0}", item);
}
```

Однако знайте, что при обработке неявно определенных локальных переменных циклы `foreach` могут использовать строго типизированные итераторы. То есть следующий фрагмент кода также синтаксически корректен:

```
// Использование ключевого слова «var» для объявления массива данных.
var evenNumbers = new int[] { 2, 4, 6, 8 };

// Используем для перебора содержимого строго
// типизированную переменную типа System.Int32.
foreach (int item in evenNumbers)
{
    Console.WriteLine("Item value: {0}", item);
}
```

Ограничения неявно типизированных переменных

Конечно, существуют различные ограничения, касающиеся использования ключевого слова `var`. Первое и основное — неявная типизация применяется *только* к локальным переменным в области видимости метода или свойства. Ключевое

слово var нельзя указывать с возвращаемыми значениями методов, параметрами и полями данных:

```
class ThisWillNeverCompile
{
    // Ошибка! Ключевое слово var нельзя использовать для полей данных!
    private var myInt = 10;

    // Ошибка! Ключевое слово var нельзя использовать
    // в качестве возвращаемых значений или параметров!
    public var MyMethod(var x, var y){}
}
```

К тому же локальным переменным, объявленным с помощью ключевого слова var, *необходимо* присвоить начальное значение при объявлении, в то же время значение null присвоено быть *не может*. Первое ограничение делает объявление неявно типизированных переменных похожим на объявление константных элементов данных. Последнее ограничение должно быть понятно, учитывая, что компилятор, основываясь лишь на значении null, не сможет определить, на какой тип в памяти должна указывать переменная:

```
// Ошибка! Необходимо присвоить значение!
var myData;

// Ошибка! Необходимо присвоить значение во время объявления!
var myInt;
myInt = 0;

// Ошибка! Нельзя присвоить значение null!
var myObj = null;
```

Однако уже после присваивания объектной локальной переменной начального значения ей можно присвоить значение null:

```
// Так можно!
var myCar = new SportsCar();
myCar = null;
```

Более того, разрешается присваивать значение одной неявно типизированной локальной переменной другим переменным, типизированным как явно, так и не явно:

```
// Тоже можно!
var myInt = 0;
var anotherInt = myInt;
string myString = «Wake up!»;
var myData = myString;
```

И наконец, знайте, что, так как var — это новое ключевое слово, его нельзя использовать в качестве названий переменных и членов типов:

```
// Ошибка компиляции!
int var = 10;
```

Неявно типизированные локальные массивы

С неявно типизированными локальными переменными тесно связана тема неявно типизированных локальных массивов. То есть можно создать новый массив, используя следующий синтаксис:

```
// a - на самом деле int[].
var a = new[] { 1, 2, 3, 4000 };

// b - на самом деле double[].
var b = new[] { 1, 1.5, 2, 2.5 };

// c - на самом деле string[].
var c = new[] { «we», null, «are», null, «family» };

// myCars - на самом деле SportsCar[].
var myCars = new[] { new SportsCar(), new SportsCar()};
```

Конечно, точно так же, как и при создании массива с помощью традиционного синтаксиса C#, элементы в списке инициализации массива должны иметь одинаковый тип (все int, все string, все SportsCar и т. д.). Таким образом, следующий код приведет к ошибке времени компиляции:

```
// Ошибка! Смешение типов!
var d = new[] { 1, «one», 2, «two» };
```

Последние штрихи в вопросе неявного назначения типов

Чтобы закончить наш обзор неявно типизированных локальных переменных, я хочу отметить, что само по себе использование ключевого слова var для объявления типов данных мало что дает. Фактически, злоупотребление этим может запутать тех, кто будет читать ваш код, потому что усложняет быстрое определение нижележащих типов данных (в результате оказывается намного сложнее разобраться в назначении переменной).

Однако, как вы увидите в главе 31, в технологиях LINQ используются *выражения запросов*, позволяющие на основе формата самого запроса получить динамически созданные результирующие множества. В таких ситуациях неявное назначение типов чрезвычайно полезно, так как избавляет от необходимости явно определять тип, который может вернуть запрос, что экономит массу времени на ввод.

Более того, процедура выяснения типов локальных переменных в данном случае совсем *не* та, что используется в языках сценариев (например, в VBScript или Perl) или в COM-типе данных `Variant`, где переменная может принимать значения разных типов в течение своей жизни в программе. Вместо этого процедура выяснения типов опирается на строгую типизацию языка C# и выполняется только при объявлении переменной во время компиляции. После этого элемент данных рассматривается так, как если бы он был явно объявлен с полученным типом, а присваивание переменной значения другого типа приведет к ошибке компиляции:

```
// Ошибка! Компилятор знает, что myVar имеет тип string, а не int!
var s = «This variable can only hold string data!»;
s = «This is fine...»;
s = 44; // Так нельзя!
```

ПРИМЕЧАНИЕ

Проект ImplicitlyTypedLocalVars находится в папке Chapter 30.

Расширяющие методы

Как вы знаете, после того как тип (класс, интерфейс, структура, перечисление или делегат) определен и откомпилирован в .NET-сборку, процедура его определения более-менее заканчивается. Единственный способ добавить новые члены, а также изменить или удалить существующие состоит в изменении и повторной компиляции кода в модернизированную сборку (можно прибегнуть и к более радикальным мерам, например динамически изменить откомпилированный тип с помощью пространства имен `System.Reflection.Emit`).

В C# 3.0 появилась возможность определять методы как *расширяющие*. В двух словах, расширяющие методы позволяют без непосредственного изменения расширяемых типов добавлять новую функциональность в существующие откомпилированные типы (например, в классы во внешней библиотеке классов платформы .NET), а также в типы, которые предстоит откомпилировать (например, типы в проекте, содержащем расширяющие методы).

Как уже отмечалось, эта техника может быть довольно полезна, если необходимо добавить новую функциональность в типы, для которых у вас нет исходного кода. Также это может быть полезно, если требуется, чтобы тип поддерживал некоторые члены (в интересах полиморфизма), но первоначальное объявление типа менять нельзя. Используя расширяющие методы, можно добавлять функциональность в уже скомпилированные типы, создавая иллюзию того, что эти методы уже там были.

При объявлении расширяющих методов первое ограничение состоит в том, что они должны определяться в *статическом классе* (см. главу 3), а значит, все расширяющие методы также должны быть объявлены как статические. Второй момент состоит в том, что методы помечаются как расширяющие за счет использования ключевого слова `this` в качестве модификатора первого (и только первого) параметра рассматриваемого метода. И третий момент — расширяющие методы

могут вызываться как через корректный экземпляр в памяти, так и *статически* посредством определяющего статического класса. Звучит странно? Давайте рассмотрим готовый пример, чтобы все стало понятно.

Определение расширяющих методов

Предположим, что вы создаете вспомогательный класс под названием `MyExtensions`, определяющий два расширяющих метода. Первый метод позволяет любому объекту в библиотеках базовых классов платформы .NET иметь новый метод под названием `DisplayDefiningAssembly()`, использующий типы из пространства имен `System.Reflection`. Второй расширяющий метод под названием `ReverseDigits()` позволяет любому объекту типа `System.Int32` получить свою новую версию с порядком следования цифр, измененным на обратный. Например, если объект целого типа, содержащий значение 1234, вызывает метод `ReverseDigits()`, возвращаемое значение будет равно 4321:

```
static class MyExtensions
{
    // Этот метод позволяет любому объекту вывести
    // на экран сборку, в которой он определен.
    public static void DisplayDefiningAssembly(this object obj)
    {
        Console.WriteLine("{0} lives here:\n\t->{1}\n", obj.GetType().Name,
            Assembly.GetAssembly(obj.GetType()));
    }

    // Этот метод позволяет любому целому числу изменять порядок своих цифр.
    // Например, в случае значения 56 будет получено 65.
    public static int ReverseDigits(this int i)
    {
        // Преобразовать int в string, а затем
        // получить все символы.
        char[] digits = i.ToString().ToCharArray();

        // Теперь изменить порядок следования элементов в массиве.
        Array.Reverse(digits);

        // Поместить назад в строку.
        string newDigits = new string(digits);

        // И, наконец, вернуть модифицированную строку назад в виде int.
        return int.Parse(newDigits);
    }
}
```

И снова обратите внимание на то, что первый параметр каждого расширяющего метода квалифицирован ключевым словом this перед определением типа параметра. Учитывая, что метод DisplayDefiningAssembly() расширяет тип System.Object, любой тип в любой сборке теперь будет иметь этот новый член. Однако метод ReverseDigits() расширяет только целочисленные типы, поэтому если какой-нибудь тип, помимо целочисленного, попытается вызвать этот метод, вы получите ошибку времени компиляции.

Знайте, что расширяющий метод может иметь множество параметров, но *только* первый параметр может быть квалифицирован ключевым словом this. В качестве иллюстрации рассмотрим перегруженный расширяющий метод, определенный в другом вспомогательном классе под названием TesterUtilClass:

```
static class TesterUtilClass
{
    // Каждый тип Int32 теперь имеет метод Foo()...
    public static void Foo(this int i)
    { Console.WriteLine("{0} called the Foo() method.", i); }

    // ...который был перегружен и принимает строку!
    public static void Foo(this int i, string msg)
    { Console.WriteLine("{0} called Foo() and told me: {1}", i, msg); }
}
```

Вызов расширяющих методов на уровне экземпляра

Теперь, когда у нас есть эти расширяющие методы, давайте убедимся, что все объекты (в библиотеках базовых классов платформы .NET это, конечно же, означает все что угодно) имеют новый метод под названием DisplayDefiningAssembly(), а типы System.Int32 (и только целые значения) имеют методы с названиями ReverseDigits() и Foo():

```
static void Main(string[] args)
{
    Console.WriteLine(«***** Fun with Extension Methods *****\n»);

    // int имеет новую функциональность!
    int myInt = 12345678;
    myInt.DisplayDefiningAssembly();

    // А еще DataSet!
    System.Data.DataSet d = new System.Data.DataSet();
    d.DisplayDefiningAssembly();

    // И SoundPlayer!
```

```
    System.Media.SoundPlayer sp = new System.Media.SoundPlayer();
    sp.DisplayDefiningAssembly();

    // Используем новую функциональность целых чисел.
    Console.WriteLine(«Value of myInt: {0}», myInt);
    Console.WriteLine(«Reversed digits of myInt: {0}»,
        myInt.ReverseDigits());
    myInt.Foo();
    myInt.Foo("Ints that Foo? Who would have thought it!");

    // Ошибка! Булевы типы не имеют метода Foo()!
    bool b2 = true;
    // b2.Foo();

    Console.ReadLine();
}
```

Результат показан на рис. 30.3.

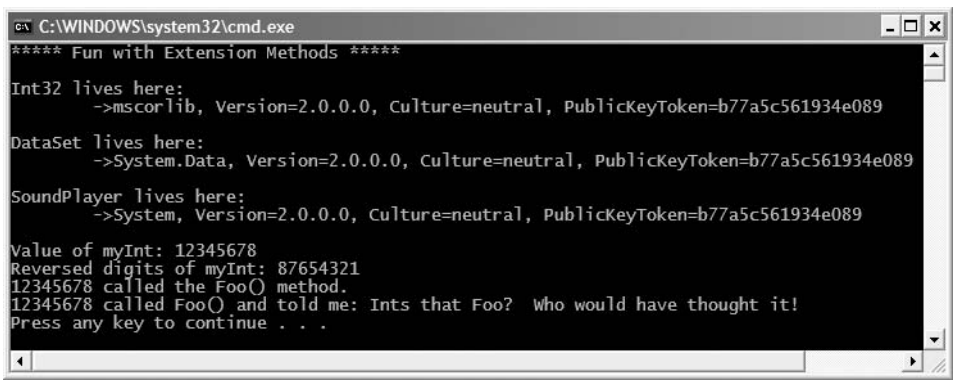

Рис. 30.3. Расширяющие методы в действии

Статический вызов расширяющих методов

Вспомните, что первый параметр расширяющего метода помечается ключевым словом `this`, после чего указывается тип элемента, к которому будет применяться этот метод. Если взглянуть, что происходит за кулисами (с помощью таких инструментов, как ildasm.exe или Lutz Roeder's Reflector), мы увидим, что компилятор просто вызывает «обычный» статический метод, передавая ему в качестве параметра переменную, вызывающую этот метод (то есть `this`). Обратите внимание на следующий фрагмент кода на C#, который моделирует выполняющуюся подстановку кода:

```
private static void Main(string[] args)
{
    Console.WriteLine(«***** Fun with Extension Methods *****\n»);
    int myInt = 12345678;
```

```
    MyExtensions.DisplayDefiningAssembly(myInt);

    DataSet d = new DataSet();
    MyExtensions.DisplayDefiningAssembly(d);

    SoundPlayer sp = new SoundPlayer();
    MyExtensions.DisplayDefiningAssembly(sp);

    Console.WriteLine("Value of myInt: {0}", myInt);
    Console.WriteLine("Reversed digits of myInt: {0}",
        MyExtensions.ReverseDigits(myInt));
    TesterUtilClass.Foo(myInt);
    TesterUtilClass.Foo(myInt, "Ints that Foo? Who would have thought it!");
}
```

Учитывая, что вызов расширяющего метода из объекта (то есть так, чтобы казалось, что метод на самом деле является методом уровня экземпляра) — это просто фокус компилятора, вы всегда можете вызывать расширяющие методы как обычные статические, используя соответствующий синтаксис C# (как было только что показано).

Импорт типов, определяющих расширяющие методы

При выделении множества статических типов с расширяющими методами в отдельное пространство имен (например, под названием `MyUtilities`) другие пространства имен в этой сборке будут использовать стандартное ключевое слово `using` языка C# для импорта не только самих статических классов, но и всех поддерживаемых расширяющих методов. Это важно запомнить, так как если вы явно не импортируете корректное пространство имен, расширяющие методы для этого файла исходного кода на C# окажутся недоступными.

Хотя может показаться, что расширяющие методы «глобальны», фактически они ограничены пространством имен, которое их импортирует. Попробуем поместить типы `MyExtensions` и `TesterUtilClass` в пространство имен под названием `MyUtilities` следующим образом:

```
namespace MyUtilities
{
    static class MyExtensions
    {
        ...
    }

    static class TesterUtilClass
```

```
    {
        ...
    }
}
```

В результате единственными целыми числами, способными получить доступ к методам `Foo()` и `ReverseDigits()` (и единственными объектами, способными вызвать метод `DisplayDefiningAssembly()`), окажутся те, которые явно импортировали пространство имен `MyUtilities`:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace TestNamespace
{
    class JustATest
    {
        void SomeMethod()
        {
            // Ошибка! Необходимо использовать пространство имен
            // MyUtilities для расширения int методом Foo()!
            int i = 0;
            i.Foo();
        }
    }
}
```

ПРИМЕЧАНИЕ

Проект ExtensionMethods находится в папке Chapter 30.

Создание и использование библиотек расширений

Последний аспект применения расширяющих методов, которые мы здесь рассмотрим, касается создания библиотек расширений. Наш предыдущий пример расширял функциональность различных типов (например, типа `System.Int32`) для использования текущим консольным приложением. Однако я уверен, что вы можете представить себе, насколько полезной была бы библиотека кода платформы .NET, определяющая множество расширений, которые могли бы использоваться несколькими приложениями. К счастью, сделать это очень просто.

В качестве иллюстрации создадим в Visual Studio 2005 «Orcas» новый проект LINQ-библиотеки (под названием MyExtensionsLibrary), выбрав в диалоговом окне **New Project** узел **LINQ Preview** и проект **LINQ Library**. Далее, переименуйте исходный файл на C# в **MyExtensions.cs** и скопируйте методы `DisplayDefiningAssembly()`

и ReverseDigits() в новое определение класса. В результате ваше пространство имен будет выглядеть следующим образом:

```
namespace MyExtensionsLibrary
{
    public static class MyExtensions
    {
        // Такая же реализация, как и ранее.
        public static void DisplayDefiningAssembly(this object obj)
        {...}

        // Такая же реализация, как и ранее.
        public static int ReverseDigits(this int i)
        {...}
    }
}
```

ПРИМЕЧАНИЕ

Если вы захотите экспортировать расширяющие методы из библиотеки кода платформы .NET, определяющий тип должен быть объявлен как public (вспомните, что модификатором доступа для типов по умолчанию является internal).

Стоит отметить, что сборка System.Query.dll (которая является ключевой LINQ-сборкой) определяет атрибут [Extension] с пространством имен System.Runtime.CompilerServices. Этот атрибут, который может применяться на уровне сборки, класса или метода, позволяет явно пометить член (а также класс или сборку) как объект, содержащий расширяющие методы.

Хотя текущая версия системы Community Technology Preview для LINQ не требует применения этого атрибута для экспорта расширяющих методов через границы сборок, с выходом финальной версии языка C# 3.0 все может измениться. Вот как выглядит измененное определение типа MyExtensions, в котором явно помечены расширяющие классы и их члены:

```
namespace MyExtensionsLibrary
{
    [Extension]
    public static class MyExtensions
    {
        [Extension]
        public static void DisplayDefiningAssembly(this object obj)
        {...}

        [Extension]
        public static int ReverseDigits(this int i)
        {...}
    }
}
```

В любом случае теперь вы можете откомпилировать свою библиотеку и использовать сборку MyExtensionsLibrary.dll в новых .NET-проектах. В результате в любом приложении вы сможете задействовать новую функциональность, добавленную типам System.Object и System.Int32. Для тестирования добавьте в текущее решение новый проект консольного LINQ-приложения (под названием MyExtensionsLibraryClient).

Далее добавьте ссылку на сборку MyExtensionsLibrary.dll. В исходном файле кода укажите, что используется пространство имен MyExtensionsLibrary, и напишите некоторый простой код, вызывающий эти новые методы для локального целого значения:

```
// Получить наше пространство имен.
using MyExtensionsLibrary;

namespace MyExtnesionsLibraryClient
{
    class Program
    {
        static void Main(string[] args)
        {
            // В этот раз данные расширяющие методы
            // были определены во внешней
            // библиотеке классов платформы .NET.
            int myInt = 987;
            myInt.DisplayDefiningAssembly();
            Console.WriteLine(«{0} is reversed to {1}»,
                myInt, myInt.ReverseDigits());
            Console.ReadLine();
        }
    }
}
```

ПРИМЕЧАНИЕ

Проекты MyExtensionsLibrary и MyExtensionsLibraryClient находятся в папке Chapter 30.

Инициализаторы объектов

Как помните из главы 12, в которой мы исследовали .NET-атрибуты, при применении атрибутов используется *синтаксис именованных свойств* для неявного присваивания значений этим свойствам:

```
// Применить атрибут, используя синтаксис именованных свойств.
[AttributeUsage(AttributeTargets.Class,
    Inherited=false, AllowMultiple = false)]
public sealed class SomeInterestingAttribute: Attribute
{ ... }
```

Очевидно, эта возможность очень полезна для синтаксиса атрибутов, так как нижележащий объект (`AttributeUsageAttribute` в данном случае) создается не при применении атрибута, а во время выполнения.

В C# 3.0 аналогичную задачу решает *синтаксис инициализаторов объектов*. Этот подход применяется для объектов, которые вы создаете непосредственно сами. В результате можно создать новую объектную переменную и несколькими строчками кода назначить ей различные свойства и/или открытые поля. Попросту говоря, инициализатор объекта состоит из списка разделенных запятыми значений, вложенного в маркеры { и }. Каждый член списка инициализации отображается на имя открытого поля или свойства инициализируемого объекта.

Чтобы увидеть этот новый синтаксис в действии, рассмотрим различные геометрические типы, созданные нами ранее (`Point`, `Rectangle`, `Hexagon` и т. д.). Например, вспомните наш простой тип `Point`, часто используемый в этой книге:

```
public struct Point
{
    private int xPos, yPos;

    public Point(int x, int y)
    { xPos = x; yPos = y; }

    public int X
    {
        get { return xPos; }
        set { xPos = value; }
    }
    public int Y
    {
        get { return yPos; }
        set { yPos = value; }
    }

    public override string ToString()
    { return string.Format(«[{0}, {1}]», xPos, yPos); }
}
```

В C# 3.0 теперь можно создавать объекты типа `Point`, применяя следующие подходы:

```
static void Main(string[] args)
{
    // Создать объект Point, вручную установив все свойства.
    Point firstPoint = new Point();
    firstPoint.X = 10;
```

```
    firstPoint.Y = 10;

    // Создать объект Point с помощью пользовательского конструктора.
    Point anotherPoint = new Point(20, 20);

    // Создать несколько объектов типа Point, используя новый
    // синтаксис инициализации объектов.
    var yetAnotherPoint = new Point { X = 30, Y = 30 };
    Point finalPoint = new Point { X = 30, Y = 30 };
}
```

Последние два объекта типа `Point` (один из которых неявно типизированный) созданы без участия пользовательского конструктора типа (как это традиционно делается), вместо этого значения присваиваются открытым свойствам `X` и `Y`. За кулисами вызывается конструктор типа по умолчанию, после чего следует присваивание значений указанным свойствам. Таким образом, `yetAnotherPoint` и `finalPoint` — это просто краткая запись синтаксиса создания переменной `firstPoint` (присваивание свойства за свойством).

Теперь вспомните, что такой же синтаксис позволяет присваивать значения открытым полям типа, которые тип `Point` в данный момент не содержит. Однако для аргументации предположим, что переменные-члены `xPos` и `yPos` были объявлены как открытые. Тогда значения этих полей можно установить следующим образом:

```
var p = new Point {xPos = 2, yPos = 3};
```

Учитывая, что `Point` теперь имеет четыре открытых члена, следующий синтаксис также допустим. Однако попробуйте определить реальные итоговые значения `xPos` и `yPos`:

```
var p = new Point {xPos = 2, yPos = 3, X = 900};
```

Как можно догадаться, `xPos` устанавливается в 900, тогда как `yPos` — в 3. Таким образом, можно правильно предположить, что инициализация объектов выполняется слева направо. Чтобы внести ясность, предыдущая инициализация `p` с помощью стандартного синтаксиса конструкторов объектов выглядела бы так:

```
Point p = new Point();
p.xPos = 2;
p.yPos = 3;
p.X = 900;
```

Вызов пользовательских конструкторов с использованием синтаксиса инициализации

В предыдущих примерах объекты типа `Point` инициализировались путем неявного вызова конструктора по умолчанию:

```
// Здесь конструктор по умолчанию вызывается неявно.
Point finalPoint = new Point { X = 30, Y = 30 };
```

Если вы хотите полностью в этом разобраться, конструктор по умолчанию можно явно вызвать следующим образом:

```
// Здесь конструктор по умолчанию вызывается явно.
Point finalPoint = new Point() { X = 30, Y = 30 };
```

Имейте в виду, что при создании объекта типа с помощью нового синтаксиса инициализации вы можете вызывать *любой* конструктор, определенный классом или структурой. Наш тип `Point` в данный момент определяет конструктор с двумя аргументами для установки позиции (x, y). Поэтому следующее объявление `Point` приводит к тому, что `X` содержит значение 100, `Y` — 100, хотя аргументы конструктора определяли значения 10 и 16:

```
// Вызов пользовательского конструктора.
Point pt = new Point(10, 16) { X = 100, Y = 100 };
```

Учитывая текущее определение типа `Point`, вызывать пользовательский конструктор при помощи синтаксиса инициализации не очень хорошо (и весьма многословно). Однако если наш тип `Point` предоставляет новый конструктор, позволяющий вызывающему устанавливать цвет (посредством пользовательского перечисления под названием `PointColor`), комбинация пользовательских конструкторов и синтаксиса инициализации объектов станет понятной. Предположим, что мы изменили тип `Point` следующим образом:

```
public enum PointColor
{ LightBlue, BloodRed, Gold }
public struct Point
{
    public int xPos, yPos;
    private PointColor c;

    public Point(PointColor color)
    {
        xPos = 0; yPos = 0;
        c = color;
    }

    public Point(int x, int y)
    {
        xPos = x; yPos = y;
        c = PointColor.Gold;
    }
    ...

    public override string ToString()
    { return string.Format(«[{0}, {1}, Color = {2}]», xPos, yPos, c); }
}
```

Теперь с помощью этого нового конструктора мы можем создать точку золотого цвета (расположенную в координатах 90, 20):

```
// Вызов более интересного пользовательского конструктора
// с использованием синтаксиса инициализации.
Point goldPoint = new Point(PointColor.Gold){ X = 90, Y = 20 };
Console.WriteLine(«Value of Point is: {0}», goldPoint);
```

Инициализация внутренних типов

Вспомните из главы 4, что отношение «имеет» (has-a) позволяет создавать новые типы, определяя переменные-члены существующих типов. Для примера предположим, что у нас есть класс `Rectangle`, который использует тип `Point` для представления координат верхнего левого и нижнего правого углов:

```
public class Rectangle
{
    private Point topLeft = new Point();
    private Point bottomRight = new Point();

    public Point TopLeft
    {
        get { return topLeft; }
        set { topLeft = value; }
    }
    public Point BottomRight
    {
        get { return bottomRight; }
        set { bottomRight = value; }
    }

    public override string ToString()
    {
        return string.Format(
           «[TopLeft: {0}, {1}, BottomRight: {2}, {3}]», topLeft.X,
            topLeft.Y, bottomRight.X, bottomRight.Y);
    }
}
```

Используя синтаксис инициализации объектов, мы можем создать новый объект типа `Rectangle` и установить внутренние переменные-члены типа `Point` следующим образом:

```
// Создать и инициализировать Rectangle.
Rectangle myRect = new Rectangle
```

```
{
    TopLeft = new Point { X = 10, Y = 10 },
    BottomRight = new Point { X = 200, Y = 200}
};
```

И снова, преимущество этого нового синтаксиса в том, что он просто сокращает количество нажатий клавиш. Вот как выглядит код, реализующий традиционный подход к созданию аналогичного объекта типа `Rectangle`:

```
// Старый подход.
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point();
p2.X = 200;
p2.Y = 200;
r.BottomRight = p2;
```

Инициализация коллекций

Тесно связана с концепцией инициализации объектов *инициализация коллекций*. Синтаксис, моделирующий простой массив, позволяет заполнять обобщенные контейнеры (например, `List<T>`) элементами. Конкретнее, этот синтаксис подходит только для типов, реализующих интерфейс `ICollection<T>` (включая пользовательские обобщенные контейнеры, которые вы можете реализовать самостоятельно), так как метод `Add()` служит для добавления элементов в коллекцию. Учитывая это ограничение, контейнеры из пространства имен `System.Collection` (например, `ArrayList`) не могут задействовать этот новый синтаксис, так как они не реализуют требуемый интерфейс. Взгляните на следующие примеры:

```
// Инициализировать стандартный массив.
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализировать обобщенный список List<> целых значений.
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Ошибка! ArrayList не реализует ICollection<T>!
ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Если ваш контейнер — это управляемая коллекция объектных типов, можете объединить синтаксис инициализации объектов с синтаксисом инициализации коллекций:

```
List<Point> myListOfPoints = new List<Point>
{
```

```
    new Point { X = 2, Y = 2},
    new Point { X = 3, Y = 3},
    new Point { X = 4, Y = 4}
};

foreach (var pt in myListOfPoints)
{
    Console.WriteLine(pt);
}
```

И снова, преимущество этого синтаксиса в том, что он позволяет экономить на нажатиях клавиш. Хотя вложенные фигурные скобки не всегда удобно читать, если не заботиться о форматировании, представьте себе объем кода, который потребовался бы для заполнения следующего типа `List<>` объектами типа `Rectangle`, если бы не было синтаксиса инициализации:

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
    new Rectangle { TopLeft = new Point { X = 10, Y = 10 },
        BottomRight = new Point { X = 200, Y = 200}},
    new Rectangle { TopLeft = new Point { X = 2, Y = 2 },
        BottomRight = new Point { X = 100, Y = 100}},
    new Rectangle { TopLeft = new Point { X = 5, Y = 5 },
        BottomRight = new Point { X = 90, Y = 75}}
};

foreach (var r in myListOfRects)
{
    Console.WriteLine(r);
}
```

ПРИМЕЧАНИЕ

Проект ObjectInitializers находится в папке Chapter 30.

Анонимные типы

Как программист, использующий объектно-ориентированный подход, вы знаете о преимуществах классов, представляющих состояние и функциональность некоторого программного объекта. Несомненно, всякий раз, когда требуется определить класс, предназначенный сразу для нескольких проектов и предоставляющий различную функциональность посредством множества методов, событий, свойств и пользовательских конструкторов, обычной практикой является создание класса на C#.

Однако в программировании существуют и другие ситуации, когда вам может потребоваться определить класс просто для моделирования множества инкапсулированных (и каким-то образом связанных) элементов данных без связанных с ними методов, событий или другой пользовательской функциональности. Более того, что если этот тип нужен только внутри вашего текущего приложения и не предназначен для многократного использования? Чтобы получить такой «временный» тип, в C# 2.0 вам потребовалось бы вручную писать новое определение класса:

```
internal class SomeClass
{
    // Определить множество закрытых переменных-членов...

    // Создать свойство для каждой переменной...

    // Перекрыть ToString() для учета всех переменных-членов...
}
```

Хотя создание такого класса — задача не слишком сложная, может потребоваться довольно много труда, если инкапсулировать множество членов. В C# 3.0 имеется более простой механизм решения этой задачи под названием анонимного типа. *Анонимные типы* являются естественным расширением синтаксиса анонимных методов языка C# (см. главу 8).

Для определения анонимного типа используется новое ключевое слово `var` в связке с только рассмотренным синтаксисом инициализации объектов. Рассмотрим следующий анонимный класс, моделирующий простой автомобиль:

```
// Создать анонимный объект, представляющий автомобиль.
var myCar = new {Color = «Bright Pink», Make = «Saab», CurrentSpeed = 55};

// Теперь можно получать и устанавливать значения,
// используя синтаксис свойств.
myCar.Color = «Black»; // Ах! Изменим цвет!
Console.WriteLine("My car is the color {0}.", myCar.Color);
```

И снова обратите внимание на то, что переменная `myCar` неявно типизирована (это обязательно), что имеет смысл, так как мы не моделируем концепцию автомобиля в строго типизированном определении класса. Также обратите внимание на необходимость указать (с помощью синтаксиса инициализации объектов) множество свойств, моделирующих инкапсулируемые данные. После определения эти свойства можно получать или изменять, используя стандартный синтаксис вызова свойств в C# (так же, как мы изменяли свойство `Color` с `Bright Pink` на `Black` в предыдущем фрагменте кода).

Внутреннее представление анонимных типов

Все анонимные типы автоматически наследуются от типа `System.Object` и поэтому поддерживают все члены, предоставляемые этим базовым классом. Учитывая это, мы можем вызвать метод `ToString()`, `GetHashCode()`, `Equals()` или `GetType()`

для неявно типизированного объекта myCar. Предположим, что наш класс Program определяет следующую статическую вспомогательную функцию:

```
static void ReflectOverAnonymousType(object obj)
{
    Console.WriteLine(«obj is an instance of: {0}», obj.GetType().FullName);
    Console.WriteLine(«Base class of {0} is {1}»,
        obj.GetType().Name,
        obj.GetType().BaseType);
    Console.WriteLine(«obj.ToString() = {0}», obj.ToString());
    Console.WriteLine(«obj.GetHashCode() = {0}», obj.GetHashCode());
    Console.WriteLine();
}
```

Теперь предположим, что мы вызываем этот метод из метода Main(), передавая объект myCar в качестве параметра:

```
static void Main(string[] args)
{
    Console.WriteLine(«***** Fun with Anonymous types *****\n»);

    // Создать анонимный объект, представляющий автомобиль.
    var myCar = new {Color = «Bright Pink»,
        Make = «Saab», CurrentSpeed = 55};

    // Определить, что было сгенерировано компилятором.
    ReflectOverAnonymousType(myCar);
    Console.ReadLine();
}
```

Взгляните на результат, показанный на рис. 30.4.

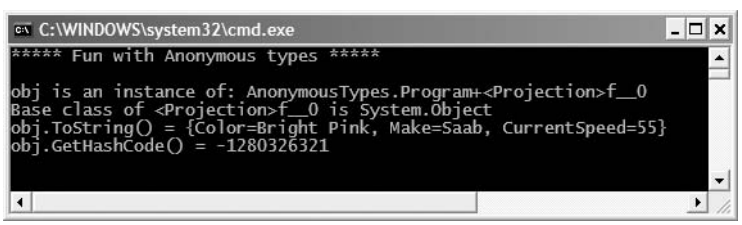

Рис. 30.4. Анонимные типы представляются классами, сгенерированными компилятором

В первую очередь обратите внимание на то, что в этом примере объект myCar имеет тип <Projection>f__0, который, по сути, *вложен* в класс Program (вспомните, что символ плюс используется прикладным программным интерфейсом отражения для обозначения вложенных типов). Знайте, что имя, назначаемое типу, полностью определяется компилятором и из вашего кода на C# недоступно.

И, возможно, самое главное. Обратите внимание, что все пары имя/значение, определенные с использованием синтаксиса инициализации объектов, отображаются на свойства с идентичными названиями и на соответствующие закрытые поля. В следующем фрагменте кода на C# приблизительно показан класс, сгенерированный компилятором для представления объекта `myCar` (что опять-таки можно проверить с помощью таких инструментов, как Lutz Roeder's Reflector и ildasm.exe):

```
[CompilerGenerated]
public sealed class <Projection>f__0
{
    // Создается конструктор по умолчанию.
    public <Projection>f__0();

    // А также готовые реализации всех виртуальных
    // членов System.Object.
    public override bool Equals(object);
    public override int GetHashCode();
    public override string ToString();

    // Кроме того, свойства, использующиеся в качестве оболочек для...
    public string Color { get; set; }
    public int CurrentSpeed { get; set; }
    public string Make { get; set; }

    // ...всех закрытых переменных-членов.
    private string _Color;
    private int _CurrentSpeed;
    private string _Make;
}
```

Реализация методов ToString() и GetHashCode()

И снова обратите внимание на то, что тип, сгенерированный компилятором, непосредственно наследуется от `System.Object` и имеет перекрытые версии методов `Equals()`, `GetHashCode()` и `ToString()`. Реализация `ToString()` просто создает строку из всех пар имя/значение:

```
public override string ToString()
{
    StringBuilder builder1 = new StringBuilder();
    builder1.Append(«{«);
    builder1.Append(«Color»);
    builder1.Append(«=»);
    builder1.Append(this.Color);
    builder1.Append(«, «);
```

```
    builder1.Append(«Make»);
    builder1.Append(«=»);
    builder1.Append(this.Make);
    builder1.Append(«, «);
    builder1.Append(«CurrentSpeed»);
    builder1.Append(«=»);
    builder1.Append(this.CurrentSpeed);
    builder1.Append(«}»);
    return builder1.ToString();
}
```

Реализация `GetHashCode()` вычисляет значение хеша, вызывая метод `GetHashCode()` для всех переменных-членов типа:

```
public override int GetHashCode()
{
    int num1 = 0;
    if (this.Color != null)
    {
        num1 ^= this.Color.GetHashCode();
    }
    if (this.Make != null)
    {
        num1 ^= this.Make.GetHashCode();
    }
    return (num1 ^ this.CurrentSpeed.GetHashCode());
}
```

Используя такую реализацию метода `GetHashCode()`, два анонимных типа получат одинаковое хеш-значение тогда (и только тогда), когда они имеют одинаковые множества свойств, которым назначены одни и те же значения.

Семантика равенства для анонимных типов

Хотя реализации перекрытых методов `ToString()` и `GetHashCode()` довольно простые, вы можете задаться вопросом, как реализован метод `Equals()`. Например, если бы мы определили две переменные для «анонимных автомобилей», содержащие одинаковые пары имя/значение, считались бы эти переменные равными или нет? Чтобы увидеть результаты, измените метод `Main()` следующим образом:

```
// Создать два анонимных класса с одинаковыми парами имя/значение.
var firstCar = new { Color = «Bright Pink»,
    Make = «Saab», CurrentSpeed = 55 };
var secondCar = new { Color = «Bright Pink»,
    Make = «Saab», CurrentSpeed = 55 };

// Считаются ли они равными при использовании метода Equals()?
```

```
if (firstCar.Equals(secondCar))
    Console.WriteLine(«Same anonymous object!»);
else
    Console.WriteLine(«Not the same anonymous object!»);

// Считаются ли они равными при использовании оператора ==?
if (firstCar == secondCar)
    Console.WriteLine(«Same anonymous object!»);
else
    Console.WriteLine(«Not the same anonymous object!»);

// Имеют ли эти объекты одинаковые нижележащие типы?
if (firstCar.GetType().Name == secondCar.GetType().Name)
    Console.WriteLine(«We are both the same type!»);
else
    Console.WriteLine(«We are different types!»);

// Показать все детали.
Console.WriteLine();
ReflectOverAnonymousType(firstCar);
ReflectOverAnonymousType(secondCar);
```

На рис. 30.5 показан вывод.

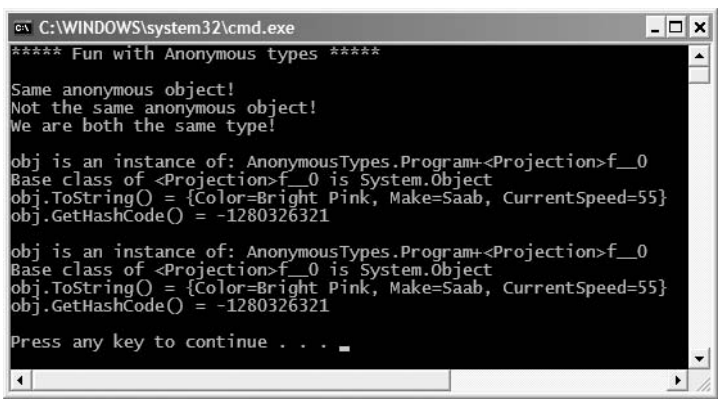

Рис. 30.5. Равенство анонимных типов

При запуске этого тестового типа вы увидите, что первая проверка с использованием метода `Equals()` возвращает `true`, поэтому на экран выводится сообщение «Same anonymous object!» Это происходит потому, что метод `Equals()`, сгенерированный компилятором, при сравнении на равенство задействует *значимую семантику* (то есть проверяются все поля сравниваемых объектов). Вот как выглядит (в первом приближении) реализация метода `Equals()`, сгенерированного для анонимного типа:

```
public override bool Equals(object obj1)
{
    Program.<Projection>f__0 f__1 = obj1 as Program.<Projection>f__0;
    if (f__1 == null)
    {
        return false;
    }
    if (!(((this.Color == null) && (f__1.Color == null))
        || this.Color.Equals(f__1.Color)))
    {
        return false;
    }
    if (!(((this.Make == null) && (f__1.Make == null))
        || this.Make.Equals(f__1.Make)))
    {
        return false;
    }
    if (this.CurrentSpeed != f__1.CurrentSpeed)
    {
        return false;
    }
    return true;
}
```

Однако вторая проверка (в которой используется оператор равенства языка C#) приводит к выводу сообщения «Not the same anonymous object!», что сначала может показаться отчасти противоестественным. Это происходит из-за того, что в C# анонимные типы *не имеют* перегруженных версий операторов сравнения (== и !=). В результате при проверке анонимных типов на равенство с помощью операторов равенства языка C# (вместо метода `Equals()`) сравниваются *ссылки*, а не значения объектов. Вспомните из главы 9, что этот режим по умолчанию присущ всем классам, пока не перегрузит указанные операторы непосредственно в коде (что невозможно для анонимных типов).

И последнее, но не менее важное. Наша итоговая проверка (в которой мы проверяем имя нижележащего типа) показывает, что анонимные типы являются экземплярами одного и того же класса, сгенерированного компилятором (в этом примере, `Program+<Projection>f__0`), поскольку объекты `firstCar` и `secondCar` имеют одинаковые свойства (`Color`, `Make` и `CurrentSpeed`).

Анонимные типы, содержащие анонимные типы

В языке C# 3.0 можно создать анонимный тип, содержащий дополнительные анонимные типы. Для примера предположим, что вы хотите смоделировать заказ на покупку, состоящий из даты, цены и покупаемого автомобиля. Вот

как выглядит новый (чуть более сложный) анонимный тип, предоставляющий такую возможность:

```
// Создать анонимный тип, содержащий другой тип.
var purchaseItem = new {
    TimeBought = DateTime.Now,
    ItemBought = new {Color = «Red», Make = «Saab», CurrentSpeed = 55},
    Price = 34.000};

ReflectOverAnonymousType(purchaseItem);
```

Здесь объект `purchaseItem` (внутренне в этом примере представленный классом под названием `<Projection>f__1`) содержит три свойства, одно из которых — это другой анонимный класс, обозначающий автомобиль. Это представляется переменной-членом типа `.<Projection>f__0` (другими словами, `<Projection>f__1` *имеет* `<Projection>f__0`).

Если применить утилиту `ildasm.exe` (а лучше программу Lutz Roeder's Reflector) для просмотра класса, сгенерированного компилятором, мы увидим, что новый тип более-менее напоминает следующий код на C#:

```
[CompilerGenerated]
public sealed class <Projection>f__1
{
    public <Projection>f__1();
    public override bool Equals(object);
    public override int GetHashCode();
    public override string ToString();

    public Program.<Projection>f__0 ItemBought { get; set; }
    public double Price { get; set; }
    public DateTime TimeBought { get; set; }

    private Program.<Projection>f__0 _ItemBought;
    private double _Price;
    private DateTime _TimeBought;
}
```

Теперь вы должны понимать синтаксис определения анонимных типов, но вы все еще можете задаваться вопросом, где же (и когда) использовать этот новый инструмент языка. Если говорить прямо, анонимные типы не следует применять повсеместно, обычно они полезны только для группы технологий LINQ (см. главу 31). Никогда не стоит отказываться от строго типизированных классов или структур просто так, особенно если вспомнить о некоторых ограничениях анонимных типов, включающих следующие:

- вы не контролируете имена анонимных типов;
- анонимные типы всегда расширяют тип `System.Object`;

- анонимные типы не поддерживают события, пользовательские методы и перекрытие методов;
- анонимные типы всегда неявно запечатаны;
- анонимные типы всегда создаются с использованием конструкторов по умолчанию.

Тем не менее, когда вы лучше изучите технологии LINQ, вы увидите, что во многих случаях синтаксис анонимных типов может быть очень полезен, если требуется быстро смоделировать общий вид, а не функциональность сущности.

ПРИМЕЧАНИЕ

Проект AnonymousTypes находится в папке Chapter 30.

Роль лямбда-выражений

Изучение новых возможностей C# 3.0 мы закончим исследованием *лямбда-выражений*. Вопросы их поддержки горячо обсуждалась в разных кругах «мира C#». И причина не в том, что лямбда-выражения не слишком полезны, а в том, что их синтаксис сначала может показаться довольно неудобным (если только у вас нет опыта программирования на каком-нибудь функциональном языке, например Lisp или Haskell).

Некоторые считают, что на практике лямбда-выражения будут игнорироваться большей частью сообщества C#. Так что если ваш день в основном занят заполнением сеток реляционными данными или созданием мультимедийного программного обеспечения, запросто может получиться, что этот инструмент языка окажется вне поля вашего зрения.

Другие же считают, что как только разработчики разберутся в базовом синтаксисе, они быстро оценят привлекательность этого инструмента. Это особенно справедливо, если вы изучали компьютерные науки (или связанные дисциплины, такие как формальная лингвистика, когнивистика или математика), где исчисление лямбда-выражений — обычная практика.

В любом случае, если вы поймете основы использования лямбда-выражений (и сможете работать со столь экономичным синтаксисом), эти небольшие программные конструкции помогут вам сберечь *очень много* времени. При исследовании технологии LINQ в следующей главе вы увидите, что эти конструкции могут также значительно упростить написание кода.

ПРИМЕЧАНИЕ

При работе над следующими примерами вы можете обнаружить, что в окне редактора кода Visual Studio 2005 «Orcas» определяет лямбда-выражения как ошибки (все они подчеркиваются красными линиями). Это ограничение текущей версии системы Community Technology Preview для LINQ. При компиляции вы не увидите ошибок компиляции (если, конечно, не наделаете синтаксических ошибок в коде!).

Лямбда-выражения как альтернатива анонимным методам

Вспомните из главы 8, что C# 2.0 предоставляет возможность обрабатывать события «прямо в коде», назначая блок инструкций кода непосредственно событию (анонимные методы). При обработке событий различных элементов пользовательского интерфейса (Button, GridView, мышь, клавиатура и т. д.) эта возможность языка вряд ли будет очень полезной. В таких случаях традиционный синтаксис обработки событий является оптимальным для отделения кода обработки событий от другого кода обработки — это справедливо для самых разных действий пользователя (нажатиях клавиш, щелчков на кнопках панелей инструментов, выборе пунктов меню). Рассмотрим следующий пример Windows Forms:

```
class MainForm : Form
{
    public MainForm()
    {
        // Обработать событие MouseMove, используя
        // стандартный синтаксис событий C#.
        this.MouseMove += new MouseEventHandler(MainForm_MouseMove);
    }
    void MainForm_MouseMove(object sender, MouseEventArgs e)
    {
        // Сделать что-нибудь с данными от мыши.
    }
}
```

Однако многие классы в библиотеках базовых классов платформы .NET определяют методы, которым в качестве параметров требуются делегаты. При использовании «традиционного» синтаксиса делегатов код может получаться довольно неуклюжим. Для примера рассмотрим метод FindAll() обобщенного типа List<T>. Этот метод ожидает обобщенный тип-делегат System.Predicate<T>, который является оболочкой для методов, возвращающих булево значение и принимающих T в качестве единственного входного параметра:

```
class Program
{
    static void Main(string[] args)
    {
        // Создать список целых значений, используя
        // синтаксис инициализации коллекций языка C# 3.0.
        List<int> list = new List<int>() {20, 1, 4, 8, 9, 44};

        // Вызвать метод FindAll(), используя традиционный
        // синтаксис делегатов.
        Predicate<int> callback = new Predicate<int>(CallMeHere);
```

```
        List<int> evenNumbers = list.FindAll(callback);

        foreach (int evenNumber in evenNumbers)
        {
            Console.WriteLine(evenNumber);
        }
        Console.ReadLine();
    }

    // Это четное значение?
    static bool CallMeHere(int i)
    {
        return (i % 2) == 0;
    }
}
```

Здесь мы имеем метод (`CallMeHere`), отвечающий за проверку четности входного целочисленного параметра с помощью оператора взятия остатка от деления `%` языка C#. Хотя код и компилируется, как ожидается, этот метод вызывается в очень ограниченных обстоятельствах, а конкретнее, когда мы вызываем метод `FindAll()`, что заставляет нас создавать полное определение метода. Если бы мы использовали анонимный метод, наш код значительно упростился бы:

```
static void Main(string[] args)
{
    // Создать список целых значений, используя
    // синтаксис инициализации коллекций языка C# 3.0.
    List<int> list = new List<int>() {20, 1, 4, 8, 9, 44};

    // Теперь используем анонимный метод.
    List<int> evenNumbers =
        list.FindAll(delegate(int i) { return (i % 2) == 0; } );

    foreach (int evenNumber in evenNumbers)
    {
        Console.WriteLine(evenNumber);
    }
    Console.ReadLine();
}
```

В данном случае вместо непосредственного создания типа-делегата `Predicate<T>` и реализации автономного метода мы использовали анонимный метод. Хотя это и шаг в правильном направлении, нам все равно требуется ключевое слово `delegate` (или строго типизированный параметр `Predicate<T>`) и мы должны

гарантировать, что список параметров полностью подходит. К тому же, согласитесь, синтаксис определения анонимного метода все еще несколько неуклюжий, что более чем очевидно:

```
List<int> evenNumbers = list.FindAll(
    delegate(int i)
    {
        return (i % 2) == 0;
    }
);
```

Для дальнейшего упрощения вызова метода `FindAll()` можно задействовать лямбда-выражения. При применении этого нового синтаксиса в коде не остается и следа от нижележащего делегата. Рассмотрим следующее изменение того же самого кода:

```
static void Main(string[] args)
{
    // Создать список целых значений, используя
    // синтаксис инициализации коллекций языка C# 3.0.
    List<int> list = new List<int>() {20, 1, 4, 8, 9, 44};

    // Теперь используем лямбда-выражение C# 3.0.
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

    foreach (int evenNumber in evenNumbers)
    {
        Console.WriteLine(evenNumber);
    }
    Console.ReadLine();
}
```

Чтобы еще больше упростить код, можно попытаться получать типы с помощью ключевого слова `var`. Как выясняется, это ключевое слово хорошо сочетается с лямбда-выражениями, поскольку они обычно могут возвращать различные значимые типы. Например, в данном случае скрытый делегат `Predicate<T>` возвращает `List<int>`; однако другие лямбда-выражения могут возвращать что-то совершенно иное. Вот последнее изменение:

```
static void Main(string[] args)
{
    // Теперь используем неявную типизацию.
    var list = new List<int>() {20, 1, 4, 8, 9, 44};
    var evenNumbers = list.FindAll(i => (i % 2) == 0);
    foreach (var evenNumber in evenNumbers)
```

```
    {
        Console.WriteLine(evenNumber);
    }
    Console.ReadLine();
}
```

В любом случае обратите внимание на то, что в метод `FindAll()`, который в действительности является лямбда-выражением, передается какой-то странный код. В этой итерации примера нет и следа делегата `Predicate<T>` (или ключевого слова `delegate`). Все, что мы указали, — это лямбда-выражение:

```
i => (i % 2) == 0
```

Прежде чем разобрать этот синтаксис, просто поймите, что лямбда-выражения могут применяться везде, где пригодились бы анонимные методы, просто на это уходит меньше нажатий клавиш. Внутренне компилятор C# преобразует наше выражение в обычный анонимный метод, использующий тип-делегат `Predicate<T>` (что можно проверить с помощью утилиты ildasm.exe).

Анализ лямбда-выражений

При создании лямбда-выражения сначала определяется список параметров, за которым следует маркер => (это оператор лямбда языка C# для исчисления лямбда-выражений), за которым следует выражение. В нашем первом примере результат выглядит следующим образом:

```
// «i» - это список параметров.
// (i % 2) == 0 - это выражение.
var evenNumbers = list.FindAll(i => (i % 2) == 0);
```

Параметры лямбда-выражений могут быть явно или неявно типизированы. В данном примере нижележащий тип данных параметра `i` (целочисленный тип) определяется неявно. Компилятор может определить, что `i` имеет целый тип на основании контекста всего лямбда-выражения. Однако можно также явно определить типы всех параметров выражения, обернув тип данных и имя переменной в пару скобок следующим образом:

```
// Теперь явно определим тип параметра.
var evenNumbers = list.FindAll((int i) => (i % 2) == 0);
```

Как видите, если лямбда-выражение имеет единственный неявно типизированный параметр, скобки в списке параметров можно опустить. Если вы хотите быть последовательным при использовании параметров лямбда-выражений, *всегда* можете помещать список параметров в скобки, что в результате приведет к следующему выражению:

```
var evenNumbers = list.FindAll((i) => (i % 2) == 0);
```

И наконец, обратите внимание, что наше текущее выражение не было помещено в скобки (естественно, мы поместили в скобки оператор взятия остатка от

деления, чтобы гарантировать, что он выполнится до проверки на равенство). Лямбда-выражения позволяют написать оператор следующим образом:

```
// Теперь также обернем выражение.
var evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Теперь, когда вы узнали разные способы создания лямбда-выражений, как прочитать эту инструкцию с лямбда-выражением на человеческом языке? Оставив чистую математику, следующее определение все объясняет:

```
// Мой список параметров (в данном случае один целочисленный параметр
// под названием i) будет обрабатываться выражением (i % 2) == 0.
var evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Два вида лямбда-выражений

Наше первое лямбда-выражение представляло собой единственную инструкцию, которая возвращала булево значение. Однако как вы, конечно же, знаете, многие целевые методы делегатов содержат несколько инструкций. По этой причине C# 3.0 позволяет определять лямбда-выражения с множеством инструкций. Если выражение должно обрабатывать несколько параметров в нескольких строках кода, область этих инструкций можно очертить привычными фигурными скобками. Рассмотрим следующий пример измененного вызова метода `FindAll()`:

```
// В этот раз лямбда-выражение содержит блок инструкций.
var justATest = list.FindAll(i => {
        Console.WriteLine(«Called by FindAll()!»);
        Console.WriteLine(«value of i is currently: {0}», i);
        bool isEven = ((i % 2) == 0);
        return isEven;
    } );
foreach (var evenNumber in justATest)
{
    Console.WriteLine(evenNumber);
}
```

В данном случае наш список параметров (и снова это единственный целочисленный параметр под названием `i`) обрабатывается множеством инструкций. Если не учитывать вызовы метода `Console.WriteLine()`, наша инструкция взятия остатка от деления была разбита на две инструкции, что сделало код более понятным.

Далее приведена окончательная версия нашего первого лямбда-выражения, результат вычисления которого показан на рис. 30.6:

```
static void Main(string[] args)
{
    Console.WriteLine(«***** Fun with Lambda Expressions *****\n»);

    // Создать список целых значений, используя синтаксис C# 3.0
```

```
    // для инициализации коллекций.
    var list = new List<int>() {20, 1, 4, 8, 9, 44};

    // В этот раз лямбда-выражение содержит блок инструкций.
    var justATest = list.FindAll(i =>
        {
            Console.WriteLine(«Called by FindAll()!»);
            Console.WriteLine(«value of i is currently: {0}», i);
            bool isEven = ((i % 2) == 0);
            return isEven;
        }
    );

    Console.WriteLine(«\nHere are the even numbers from your list:»);
    foreach (var evenNumber in justATest)
    {
        Console.WriteLine(evenNumber);
    }

    Console.ReadLine();
}
```

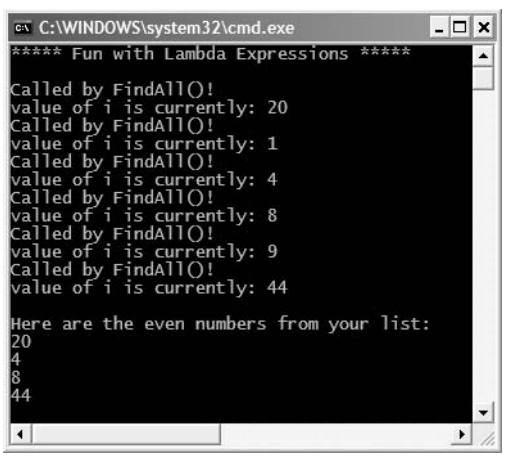

Рис. 30.6. Результат вычисления нашего первого лямбда-выражения

ПРИМЕЧАНИЕ

Проект SimpleLambdaExpressions находится в папке Chapter 30.

Изменение проекта CarDelegate с помощью лямбда-выражений

Учитывая, что весь смысл лямбда-выражений состоит в том, чтобы предоставить ясный и лаконичный способ определения анонимных методов (и поэтому неявно упростить работу с делегатами), давайте переделаем проект CarDelegate, созданный в главе 8. Вспомните, что в этом примере определялся тип `Car` с двумя пользовательскими делегатами (`AboutToBlow` и `Exploded`), а также набор методов (`OnAboutToBlow()`, `RemoveAboutToBlow()`, `OnExploded()` и `RemoveExploded()`), что позволяло вызывающей стороне передавать объекты-делегаты в качестве параметров для подключения к источнику событий или отключения от него. Рисунок 30.7 призван освежить вашу память.

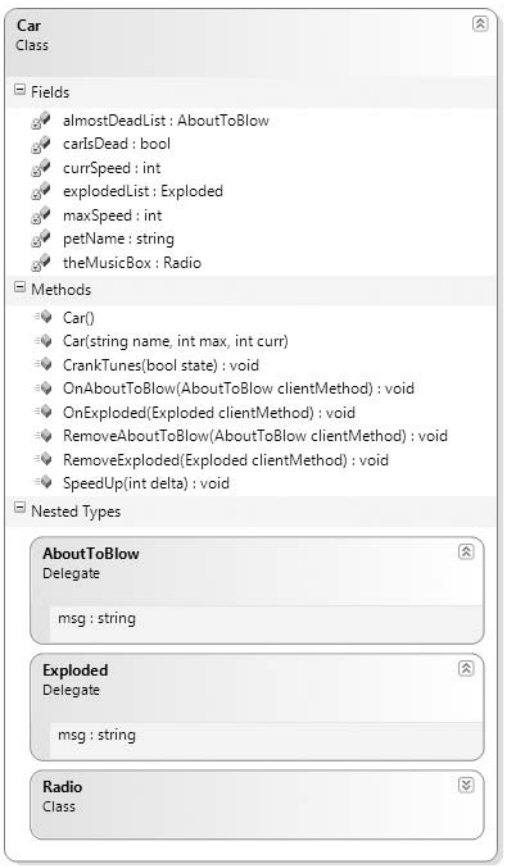

Рис. 30.7. Тип Car из проекта CarDelegate в главе 8

Вот упрощенная версия класса `Program` этого проекта, в которой для обработки обратных вызовов используется традиционный синтаксис делегатов:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(«***** More Fun with Lambdas *****\n»);

        // Создать автомобиль, как обычно.
        Car c1 = new Car(«SlugBug», 100, 10);

        // Традиционный синтаксис делегатов.
        c1.OnAboutToBlow(new Car.AboutToBlow(CarAboutToBlow));
        c1.OnExploded(new Car.Exploded(CarExploded));

        // Ускориться (это приведет к генерации событий).
        Console.WriteLine(«\n***** Speeding up *****»);
        for (int i = 0; i < 6; i++)
            c1.SpeedUp(20);
        Console.ReadLine();
    }

    // Цели делегатов.
    public static void CarAboutToBlow(string msg)
    { Console.WriteLine(msg); }

    public static void CarExploded(string msg)
    { Console.WriteLine(msg); }
}
```

Теперь посмотрим на измененную версию метода `Main()`, в которой используется синтаксис анонимных методов:

```
static void Main(string[] args)
{
    Console.WriteLine(«***** More Fun with Lambdas *****\n»);

    // Создать автомобиль, как обычно.
    Car c1 = new Car(«SlugBug», 100, 10);

    // Теперь используем анонимные методы.
    c1.OnAboutToBlow(delegate(string msg) { Console.WriteLine(msg); });
    c1.OnExploded(delegate(string msg) { Console.WriteLine(msg); });

    // Ускориться (это приведет к генерации событий).
```

```
    Console.WriteLine(«\n***** Speeding up *****»);
    for (int i = 0; i < 6; i++)
        c1.SpeedUp(20);

    Console.ReadLine();
}
```

И, наконец, посмотрим на версию метода `Main()` с лямбда-выражениями:

```
static void Main(string[] args)
{
    Console.WriteLine(«***** More Fun with Lambdas *****\n»);

    // Создать автомобиль, как обычно.
    Car c1 = new Car(«SlugBug», 100, 10);

    // Теперь с лямбда-выражениями!
    c1.OnAboutToBlow(msg => { Console.WriteLine(msg); });
    c1.OnExploded(msg => { Console.WriteLine(msg); });

    // Ускориться (это приведет к генерации событий).
    Console.WriteLine(«\n***** Speeding up *****»);
    for (int i = 0; i < 6; i++)
        c1.SpeedUp(20);

    Console.ReadLine();
}
```

Результат во всех случаях будет одинаков (рис. 30.8). Как видите, синтаксис лямбда-выражений наиболее компактный и функциональный, тем не менее помните, что в любом случае применять лямбда-выражения вовсе не обязательно.

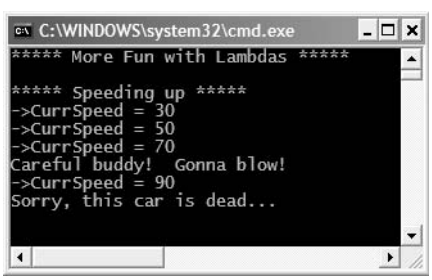

Рис. 30.8. Результат применения традиционного синтаксиса делегатов, синтаксиса анонимных методов и синтаксиса лямбда-выражений один и тот же

ПРИМЕЧАНИЕ

Проект CarDelegateWithLambdas находится в папке Chapter 30.

Лямбда-выражения с множеством параметров (или без параметров)

Все лямбда-выражения, которые мы здесь видели, имели единственный параметр. Однако это не обязательное требование: лямбда-выражения могут иметь множество аргументов или вообще их не иметь. Для иллюстрации первого сценария предположим, что у нас есть следующая версия типа SimpleMath, который мы впервые увидели в главе 8:

```
public class SimpleMath
{
    public delegate void MathMessage(string msg, int result);
    private MathMessage mmDelegate;

    public void SetMathHandler(MathMessage target)
    { mmDelegate = target; }

    public void Add(int x, int y)
    {
        if (mmDelegate != null)
            mmDelegate.Invoke(«Adding has completed!», x + y);
    }
}
```

Обратите внимание, что делегат MathMessage ожидает два параметра. Для представления их в виде лямбда-выражений наш метод Main() можно переписать следующим образом:

```
static void Main(string[] args)
{
    // Регистрация делегата в качестве лямбда-выражения.
    SimpleMath m = new SimpleMath();
    m.SetMathHandler((msg, result) =>
        {Console.WriteLine(«Message: {0}, Result: {1}», msg, result);});

    // Это запустит лямбда-выражение.
    m.Add(10, 10);

    Console.ReadLine();
}
```

Здесь мы используем механизм получения типов, так как наши два параметра для простоты не были строго типизированы. Однако мы можем вызвать метод SetMathHandler():

```
m.SetMathHandler((string msg, int result) =>
    {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});
```

И наконец, при использовании лямбда-выражений для взаимодействия с делегатом, совсем не принимающим параметры, нужно указать пустые скобки в качестве параметра. Для примера предположим, что мы определили следующий делегат:

```
public delegate string VerySimpleDelegate();
```

Результат вызова можно обработать следующим образом:

```
// Выводит на консоль сообщение «Enjoy your string!».
VerySimpleDelegate d = new VerySimpleDelegate( () =>
    {return «Enjoy your string!»;} );
Console.WriteLine(d.Invoke());
```

Теперь я надеюсь, вы видите общую роль лямбда-выражений и понимаете, что они обеспечивают «функциональный образ» для анонимных методов и делегатов. Более того, при исследовании LINQ вы также увидите, что лямбда-выражения довольно часто помогают упростить работу с реляционными данными и XML-документами.

ПРИМЕЧАНИЕ

C# 3.0 позволяет также представлять лямбда-выражения в виде объектов в памяти с помощью деревьев выражений. Это может быть очень полезно для сторонних разработчиков, создающих программное обеспечение, призванное расширить функциональность существующих лямбда-выражений. Я подозреваю, что большинству проектов на C# это не потребуется, так что если вам интересно, обратитесь к документации по .NET Framework SDK.

ПРИМЕЧАНИЕ

Проект LambdaExpressionsMultipleParams можно найти в папке Chapter 30.

Заключение

C# 3.0 предоставляет несколько очень интересных инструментов, переносящих этот язык в семейство *функциональных языков*. В данной главе дается обзор всех ключевых изменений, начиная с неявно типизированных локальных переменных. Хотя большинство локальных переменных объявлять с ключевым словом `var` вряд ли потребуется, как вы увидите в следующей главе, это может значительно упростить взаимодействие с группой технологий LINQ.

В этой главе также затронуты темы, касающиеся роли расширяющих методов (позволяющих добавлять новую функциональность в откомпилированные типы) и синтаксиса инициализации объектов (который можно использовать для присваивания значений свойствам во время создания). Заканчивается глава исследованием анонимных типов и множеством примеров лямбда-выражений. По существу, новый оператор лямбда (`=>`) позволяет значительно упростить работу с типами-делегатами и представляет собой более элегантное решение по сравнению с синтаксисом анонимных методов в C# 2.0.