За одним важным исключением, тип dynamic очень похож на тип object, поскольку

его можно использовать для ссылки на объект любого типа. А отличается он

от типа object тем, что вся проверка объектов типа dynamic на соответствие типов

откладывает до времени выполнения, тогда как объекты типа object подлежат этой

проверке во время компиляции. Преимущество откладывания подобной проверки до

времени выполнения состоит в том, что во время компиляции предполагается, что

объект типа dynamic поддерживает любые операции, включая применение операторов,

вызовы методов, доступ к полям и т.д. Это дает возможность скомпилировать код

без ошибок. Конечно, если во время выполнения фактический тип, присваиваемый

объекту, не поддерживает ту или иную операцию, то возникнет исключительная ситуация

во время выполнения.

В приведенном ниже примере программы применение типа dynamic демонстрируется

на практике.

(***glava20\_14***)

using System.Globalization;

class DynDemo

{

static void Main()

{

//two dynamic variable

dynamic str;

dynamic val;

//supports implicit change to dynamic types

//thats why next assigns is valid

str = "This is a string";

val = 10;

Console.WriteLine("str: " + str);

Console.WriteLine("val: " + val + '\n');

str = str.ToUpper(CultureInfo.CurrentCulture);

Console.WriteLine("str: " + str);

val = val - 2;

Console.WriteLine("val: " + val + '\n');

string str2 = str.ToLower(CultureInfo.CurrentCulture);

Console.WriteLine("str2: " + str2);

//suports implicit assign from dynamic types

int x = val \* 2;

Console.WriteLine("x: " + x);

}

}

Обратите внимание в этой программе на две переменные str и val, объявляемые

с помощью типа dynamic. Это означает, что проверка на соответствие типов операций

с участием обеих переменных не будет произведена во время компиляции. В итоге

для них оказывается пригодной любая операция. В данном случае для переменной

str вызываются методы ToUpper() и ToLower() класса String, а переменная участвует

в операциях сложения и умножения. И хотя все перечисленные выше действия

совместимы с типами объектов, присваиваемых обеим переменным в рассматриваемом

здесь примере, компилятору об этом ничего не известно — он просто принимает.

И это, конечно, упрощает программирование динамических процедур, хотя и допускает

возможность появления ошибок в подобных действиях во время выполнения.

В разбираемом здесь примере программа ведет себя "правильно" во время выполнения,

поскольку объекты, присваиваемые упомянутым выше переменным, поддерживают

действия, выполняемые в программе. В частности, переменной val присваивается

целое значение, и поэтому она поддерживает такие целочисленные операции,

как сложение. А переменной str присваивается символьная строка, и поэтому она

поддерживает строковые операции. Следует, однако, иметь в виду, что ответственность

за фактическую поддержку типом объекта, на который делается ссылка, всех операций

над данными типа dynamic возлагается на самого программирующего. В противном

случае выполнение программы завершится аварийным сбоем.

В приведенном выше примере обращает на себя внимание еще одно обстоятельство:

переменной типа dynamic может быть присвоен любой тип ссылки на объект

благодаря неявному преобразованию любого типа в тип dynamic. Кроме того, тип

dynamic автоматически преобразуется в любой другой тип. Разумеется, если во время

выполнения такое преобразование окажется неправильным, то произойдет ошибка

при выполнении. Так, если добавить в конце рассматриваемой здесь программы следующую

строку кода:

bool b = val;

то возникнет ошибка при выполнении из-за отсутствия неявного преобразования типа

int (который оказывается типом переменной val во время выполнения) в тип bool.

Поэтому данная строка кода приведет к ошибке при выполнении, хотя она и будет

скомпилирована безошибочно.

В этом, собственно, и заключается основное отличие типов object и dynamic. Несмотря на то что оба типа могут использоваться для ссылки на объект любого другого типа, над переменной

типа object можно производить только те действия, которые поддерживаются

типом object. Если же вы используете тип dynamic, то можете указать какое угодно

действие, при условии что это действие поддерживается конкретным объектом, на который

делается ссылка во время выполнения.

Для того чтобы стало понятно, насколько тип dynamic способен упростить решение

некоторых задач, рассмотрим простой пример его применения вместе с рефлексией.

Как пояснялось в главе 17, чтобы вызвать метод для объекта класса, получаемого

во время выполнения с помощью рефлексии, можно, в частности, обратиться к методу

Invoke(). И хотя такой способ оказывается вполне работоспособным, нужный метод

намного удобнее вызвать по имени в тех случаях, когда его имя известно. Например,

вполне возможна такая ситуация, когда в некоторой сборке содержится конкретный

класс, поддерживающий методы, имена и действия которых заранее известны. Но поскольку

эта сборка подвержена изменениям, то приходится постоянно убеждаться в том,

что используется последняя ее версия. Для проверки текущей версии сборки можно,

например, воспользоваться рефлексией, сконструировать объект искомого класса, а затем

вызвать методы, определенные в этом классе. Теперь эти методы можно вызвать по

имени с помощью типа dynamic, а не метода Invoke(), поскольку их имена известны.

Разместите сначала приведенный ниже код в файле с именем MyClass.cs. Этот

код будет динамически загружаться посредством рефлексии.

(Программу я так и не запустил, чтобы создать библиотеку надо создавать новый проект в студии, но код оставляю.)

(***MyClass.dll***)

using System;

public class DivBy

{

public bool IsDivBy(int a, int b)

{

if ((a % b) == 0) return true;

return false;

}

public bool IsEven(int a)

{

if ((a % 2) == 0) return true;

return false;

}

}

Затем скомпилируйте этот файл в библиотеку DLL под именем MyClass.dll.

(***glava20\_15***)

using System.Reflection;

class DenRefDemo

{

static void Main()

{

Assembly asm = Assembly.LoadFrom("MyClass.dll");

Type[] all = asm.GetTypes();

//find class DivBy

int i;

for (i = 0; i < all.Length; i++)

if (all[i].Name == "DivBy") break;

if (i == all.Length)

{

Console.WriteLine("Class DivBy has not been found.");

return;

}

Type t = all[i];

//now find default constructor

ConstructorInfo[] ci = t.GetConstructors();

int j; ;

for (j = 0; j < ci.Length; j++)

if (ci[j].GetParameters().Length == 0) break;

if (j == ci.Length)

{

Console.WriteLine("Constructor has not been found.");

return;

}

//create object DivBy class dynamic

dynamic obj = ci[i].Invoke(null);

//then call method by name obj, this is valid

//because obj is dynamic, and call methods checks

//type match while executing, not compilation

if (obj.IsDivBy(15, 4))

Console.WriteLine("15 devide by 3.");

else

Console.WriteLine("15 NOT devide by 3.");

if (obj.IsEven(9))

Console.WriteLine("9 is Even.");

else

Console.WriteLine("9 is NOT Even.");}}

Как видите, в данной программе сначала динамически загружается библиотека

MyClass.dll, а затем используется рефлексия для построения объекта класса DivBy.

Построенный объект присваивается далее переменной obj типа dynamic. А раз так,

то методы IsDivBy() и IsEven() могут быть вызваны для переменной obj по имени,

а не с помощью метода Invoke(). В данном примере это вполне допустимо, поскольку

переменная obj на самом деле ссылается на объект класса DivBy. В противном случае

выполнение программы завершилось бы неудачно.

Приведенный выше пример сильно упрощен и несколько надуман. Тем не менее он

наглядно показывает главное преимущество, которое дает тип dynamic в тех случаях,

когда типы получаются во время выполнения. Когда характеристики искомого типа,

в том числе методы, операторы, поля и свойства, заранее известны, эти характеристики

могут быть получены по имени с помощью типа dynamic, как следует из приведенного

выше примера. Благодаря этому код становится проще, короче и понятнее.

Применяя тип dynamic, следует также иметь в виду, что при компиляции программы

тип dynamic фактически заменяется объектом, а для описания его применения во

время выполнения предоставляется соответствующая информация. И поскольку тип

dynamic компилируется в тип object для целей перегрузки, то оба типа dynamic

и object расцениваются как одно и то же. Поэтому при компиляции двух следующих

перегружаемых методов возникнет ошибка.

static void f(object v){ // ... }

static void f(dynamic v) { // ... } // Ошибка!

И последнее замечание: тип dynamic поддерживается компонентом DLR (Dynamic

Language Runtime — Средство создания динамических языков во время выполнения),

внедренным в .NET 4.0.

**Возможность взаимодействия с моделью СОМ**

В версии C# 4.0 внедрены средства, упрощающие возможность взаимодействия

с неуправляемым кодом, определяемым моделью компонентных объектов (СОМ)

и применяемым, в частности, в COM-объекте Office Automation. Некоторые из этих

средств, в том числе тип dynamic, именованные и необязательные свойства, пригодны

для применения помимо возможности взаимодействия с моделью СОМ. Тема модели

СОМ вообще и COM-объекта Office Automation в частности весьма обширна, а порой и

довольно сложна, чтобы обсуждать ее в этой книге. Поэтому возможность взаимодействия

с моделью СОМ выходит за рамки данной книги.

Тем не менее две особенности, имеющие отношение к возможности взаимодействия

с моделью СОМ, заслуживают краткого рассмотрения в этом разделе. Первая из

них состоит в применении индексированных свойств, а вторая — в возможности передавать

аргументы значения тем COM-методам, которым требуется ссылка.

Как вам должно быть уже известно, в C# свойство обычно связывается только с одним

значением с помощью одного из аксессоров get или set. Но совсем иначе дело обстоит

со свойствами модели СОМ. Поэтому, начиная с версии C# 4.0, в качестве выхода из

этого затруднительного положения во время работы с COM-объектом появилась возможность

пользоваться *индексированным свойством* для доступа к COM-свойству, имеющему

несколько параметров. С этой целью имя свойства индексируется, почти так

же, как это делается с помощью индексатора. Допустим, что имеется объект myXLApp,

который относится к типу Microsoft.Office.Interop.Execl.Application.

В прошлом для установки строкового значения "OК" в ячейках С1-СЗ электронной

таблицы Excel можно было бы воспользоваться оператором, аналогичным следующему.

***myXLapp.get\_Range("C1", "С3").set\_Value(Type.Missing, "OK");***

В этой строке кода интервал ячеек электронной таблицы получается при вызове

метода get\_Range(), для чего достаточно указать начало и конец интервала. А значения

задаются при вызове метода set\_Value(), для чего достаточно указать тип (что

не обязательно) и конкретное значение. В этих методах используются свойства Range и

Value, поскольку у обоих свойств имеются два параметра. Поэтому в прошлом к ним

нельзя было обращаться как к свойствам, но приходилось пользоваться упомянутыми

выше методами. Кроме того, аргумент Type.Missing служил в качестве обычного заполнителя,

который передавался для указания на тип, используемый по умолчанию.

Но, начиная с версии C# 4.0, появилась возможно переписать приведенный выше оператор,

приведя его к следующей более удобной форме.

***myXLapp.Range["C1", "С3"].Value = "OK";***

В этом случае значения интервала ячеек электронной таблицы передаются с использованием

синтаксиса индексаторов, а заполнитель Туре.Missing уже не нужен,

поскольку данный параметр теперь задается по умолчанию.

Как правило, при определении в методе параметра ref приходится передавать

ссылку на этот параметр. Но, работая с моделью СОМ, можно передавать параметру

ref значение, не заключая его предварительно в оболочку объекта. Дело в том, что

компилятор будет автоматически создавать временный аргумент, который уже заключен

в оболочку объекта, и поэтому указывать параметр ref в списке аргументов уже

не нужно.