12

Динамические расширения языка

В ЭТОЙ ГЛАВЕ...

- Среда Dynamic Language Runtime
- Тип `dynamic`
- Хостинг DLR ScriptRuntime
- Класс `DynamicObject`
- Класс `ExpandoObject`

Развитие таких языков, как Ruby, Python, и растущее использование JavaScript повлекло за собой усиленный интерес к динамическому программированию. В предыдущих версиях .NET Framework был открыт путь к такому виду программирования в C#, благодаря появлению ключевого слова `var` и анонимных методов. В версию .NET 4 был добавлен тип `dynamic`. Хотя C# остается статически типизированным языком, эти дополнения предоставили динамические возможности, которых недоставало некоторым разработчикам.

В этой главе вы ознакомитесь с типом `dynamic` и правилами его использования. Также будет показано, как выглядит реализация класса `DynamicObject` и как он может быть использован.

Среда Dynamic Language Runtime

Динамические возможности C# 4 — это часть исполняющей среды динамического языка (Dynamic Language Runtime — DLR). Система DLR представляет собой набор служб, добавленных к CLR, которые позволяют обрабатывать код динамических языков, подобных Ruby и Python. Система DLR также позволяет языку C# обрести некоторые динамические возможности, вроде тех, что имеют динамические языки.

Доступна версия DLR с открытым исходным кодом, которая находится на веб-сайте CodePlex. Эта же версия входит в состав .NET 4 Framework, но имеет дополнительную поддержку для реализаторов языков.

В .NET Framework система DLR находится в пространстве имен `System.Dynamic`, плюс с ней связано несколько дополнительных классов в пространстве имен `System.Runtime.CompilerServices`.

DLR используют IronRuby и IronPython — версии языков Ruby и Python с открытым кодом. Silverlight также использует DLR. Исполняющая среда сценариев позволяет передавать переменные в сценарии и из них.

Тип `dynamic`

Тип `dynamic` позволяет писать код, который обходит проверку времени компиляции. Компилятор предполагает, что любая операция, определенная для объекта типа `dynamic`, является допустимой. Если операция некорректна, ошибка не будет обнаружена вплоть до времени выполнения. Продемонстрируем это на следующем примере:

```
class Program
{
   static void Main(string[] args)
   {
      var staticPerson = new Person();
      dynamic dynamicPerson = new Person();
      staticPerson.GetFullName("John", "Smith");
      dynamicPerson.GetFullName("John", "Smith");
   }
}
class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string GetFullName()
   {
      return string.Concat(FirstName, " ", LastName);
   }
}
```

Этот пример не скомпилируется из-за вызова `staticPerson.GetFullName()`. В классе `Person` не существует метода, который принимал бы два параметра, поэтому компилятор

сообщит об ошибке. Если закомментировать эту строку кода, пример будет скомпилирован. После запуска кода должна возникнуть ошибка времени выполнения. Она проявится в виде исключения RuntimeBinderException. При этом RuntimeBinder — объект времени выполнения, который проверит, действительно ли Person поддерживает вызываемый метод. Позднее в настоящей главе мы еще вернемся к этому.

В отличие от ключевого слова var, объект, объявленный как dynamic, может менять тип во время выполнения. Напомним, что при использовании ключевого слова var определение типа объекта откладывается. Но как только он определен компилятором, изменять его уже нельзя. Что касается объекта dynamic, то можно не просто изменить его тип, но делать это многократно. Это отличается от приведения объекта от одного типа к другому. При приведении объекта создается новый объект с другим, но совместимым типом. Например, привести int к Person нельзя. В следующем примере показано, что если объект является dynamic, его тип можно изменить с int на Person:

```
dynamic dyn;
dyn = 100;
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);
dyn = "Какая-то строка";
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);
dyn = new Person() { FirstName = "Bugs", LastName = "Bunny" };
Console.WriteLine(dyn.GetType());
Console.WriteLine("{0} {1}", dyn.FirstName, dyn.LastName);
```

*Фрагмент кода **Dynamic\Program.cs***

Выполнение этого кода покажет, что объект dyn действительно меняет свой тип с System.Int32 на System.String и затем на Person. Если бы dyn был объявлен как int или string, скомпилировать данный код не удалось бы.

С типом dynamic связано несколько ограничений. Объект dynamic не поддерживает расширяющие методы. Анонимные функции (лямбда-выражения) также не могут использоваться в качестве параметров вызова динамического метода, а потому LINQ не может нормально работать с динамическими объектами. Большинство вызовов LINQ являются вызовами расширяющих методов, и лямбда-выражения используются в качестве аргументов этих расширяющих методов.

Особенности типа dynamic

Но что же происходит “за кулисами”, позволяющее реализовать описанное выше поведение? C# по-прежнему является статически типизированным языком. Это не изменилось. Давайте взглянем на код IL, который генерируется при использовании типа dynamic.

Сначала возьмем пример кода C#, который выглядит следующим образом:

```
using System;
namespace DeCompile
{
   class Program
   {
      static void Main(string[] args)
      {
         StaticClass staticObject = new StaticClass();
         DynamicClass dynamicObject = new DynamicClass();
         Console.WriteLine(staticObject.IntValue);
         Console.WriteLine(dynamicObject.DynValue);
         Console.ReadLine();
      }
   }
```

```
    class StaticClass
    {
       public int IntValue = 100;
    }
    class DynamicClass
    {
       public dynamic DynValue = 100;
    }
}
```

Здесь определены два класса — StaticClass и DynamicClass. Класс StaticClass имеет единственное поле, которое возвращает int. Класс DynamicClass имеет поле, возвращающее объект dynamic. Метод Main просто создает эти объекты и печатает значения, возвращаемые методами. Достаточно просто.

Теперь закомментируем ссылки на DynamicClass в Main, как показано ниже:

```
static void Main(string[] args)
{
    StaticClass staticObject = new StaticClass();
    //DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

С помощью инструмента ildasm (см. главу 18) можно просмотреть код IL, сгенерированный для метода Main:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size        26 (0x1a)
  .maxstack 1
  .locals init ([0] class DeCompile.StaticClass staticObject)
  IL_0000: nop
  IL_0001: newobj     instance void DeCompile.StaticClass::.ctor()
  IL_0006: stloc.0
  IL_0007: ldloc.0
  IL_0008: ldfld      int32 DeCompile.StaticClass::IntValue
  IL_000d: call       void [mscorlib]System.Console::WriteLine(int32)
  IL_0012: nop
  IL_0013: call       string [mscorlib]System.Console::ReadLine()
  IL_0018: pop
  IL_0019: ret
} // end of method Program::Main
```

Не углубляясь в детали IL, и просто взглянув на этот раздел кода, можно довольно много сказать о нем. В строке IL_0001 вызывается конструктор StaticClass. В строке IL_0008 производится обращение к полю IntValue в StaticClass. В следующей строке выводится значение.

Теперь закомментируйте ссылки на StaticClass и уберите комментарии со ссылок DynamicClass:

```
static void Main(string[] args)
{
    //StaticClass staticObject = new StaticClass();
    DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

После компиляции приложения будет сгенерирован следующий код IL:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size        121 (0x79)
  .maxstack 9
  .locals init ([0] class DeCompile.DynamicClass dynamicObject,
               [1] class
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo[] CS$0$0000)
  IL_0000: nop
  IL_0001: newobj     instance void DeCompile.DynamicClass::.ctor()
  IL_0006: stloc.0
  IL_0007: ldsfld     class [System.Core]System.Runtime.CompilerServices.CallSite`1
                      <class [mscorlib]
System.Action`3<class

[System.Core]System.Runtime.CompilerServices.CallSite,class [mscorlib]
System.Type,object>> DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_000c: brtrue.s   IL_004d
  IL_000e: ldc.i4.0
  IL_000f: ldstr "WriteLine"
  IL_0014: ldtoken    DeCompile.Program
  IL_0019: call       class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_001e: ldnull
  IL_001f: ldc.i4.2
  IL_0020: newarr     [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo
  IL_0025: stloc.1
  IL_0026: ldloc.1
  IL_0027: ldc.i4.0
  IL_0028: ldc.i4.s   33
  IL_002a: ldnull
  IL_002b: newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfoFlags,
string)
  IL_0030:  stelem.ref
  IL_0031: ldloc.1
  IL_0032: ldc.i4.1
  IL_0033: ldc.i4.0
  IL_0034: ldnull
  IL_0035: newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfoFlags,
string)
  IL_003a: stelem.ref
  IL_003b: ldloc.1
  IL_003c: newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder
.CSharpInvokeMemberBinder::.ctor(valuetype Microsoft.CSharp]Microsoft.CSharp
.RuntimeBinder.CSharpCallFlags,
string,

class [mscorlib]System.Type,
class [mscorlib]System.Collections.Generic.IEnumerable`1

<class [mscorlib]System.Type>,
class [mscorlib]System.Collections.Generic.IEnumerable`1
```

```
<class [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo>)
  IL_0041: call       class [System.Core]System.Runtime.CompilerServices.CallSite`1
<!0> class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Create(
     class [System.Core]System.Runtime.CompilerServices.CallSiteBinder)
  IL_0046: stsfld     class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
     DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_004b: br.s       IL_004d
  IL_004d: ldsfld     class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
     DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_0052: ldfld      !0 class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Target
  IL_0057: ldsfld     class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
     DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_005c: ldtoken    [mscorlib]System.Console
  IL_0061: call
     class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_0066: ldloc.0
  IL_0067: ldfld      object DeCompile.DynamicClass::DynValue
  IL_006c: callvirt   instance void class [mscorlib]System.Action`3
        <class [System.Core]System.Runtime.CompilerServices.CallSite, class
        [mscorlib]System.Type,object>::Invoke(!0,!1,!2)
  IL_0071: nop
  IL_0072: call       string [mscorlib]System.Console::ReadLine()
  IL_0077: pop
  IL_0078: ret
} // end of method Program::Main
```

Определенно можно сказать, что для поддержки динамического типа компилятор C# выполнил немного больше работы. В сгенерированном коде несложно заметить ссылки на `System.Runtime.CompilerServices.CallSite` и `System.Runtime.CompilerServices.CallSiteBinder`.

`CallSite` представляет собой тип, обрабатывающий поиск во время выполнения. Когда во время выполнения осуществляется вызов на динамическом объекте, кто-то должен просмотреть объект и проверить, существует ли указанный член. `CallSite` кэширует эту информацию, так что этот поиск не приходится выполнять повторно. Без этого производительность циклических структур была бы под вопросом.

После того, как `CallSite` выполнит поиск члена, вызывается `CallSiteBonder`. Он извлекает информацию у `CallSite` и генерирует дерево выражения, представляющее операцию, к которой нужно привязаться.

Очевидно, что здесь много чего происходит. Поэтому следовало очень постараться, чтобы оптимизировать эту очень сложную операцию. Должно быть очевидным, что хотя применение типа `dynamic` удобно, за это приходится платить свою цену.

Хостинг DLR ScriptRuntime

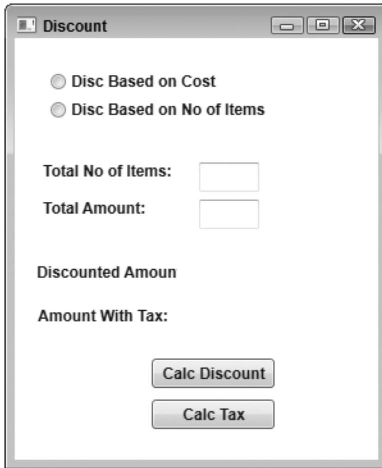

Рис. 12.1. *Пример экрана приложения*

Представьте себе, что к приложению можно добавить возможности обработки сценариев. При этом допускается передача значений в сценарий и из него, а приложение может пользоваться результатами работы сценария. Это именно то, что обеспечивает хостинг DLR ScriptRuntime. В настоящее время в качестве хостируемых языков сценариев поддерживаются IronPython, IronRuby и JavaScript.

Вместе со ScriptRuntime вы получаете возможность выполнения фрагментов кода или компиляции сценария, хранящихся в файле. Вы можете выбрать нужный языковый механизм и позволить DLR обнаружить его. Сценарий может быть создан в собственном или в текущем домене. Имеется возможность не только передавать значения и получать их из сценария, но также вызывать методы на динамических объектах, созданных в сценарии.

При такой степени гибкости у хостинга ScriptRuntime существуют практически бесчисленные возможности применения. В следующем примере демонстрируется один из способов использования ScriptRuntime. Предположим, что разрабатывается приложение с корзиной для покупок. Одним из обычных требований является вычисление скидки на основе определенного критерия. Эти скидки часто изменяются в связи с началом и окончанием различных торговых акций. Существует много способов удовлетворения такого требования; в данном примере показано, как это можно сделать с помощью ScriptRuntime и небольшого сценария Python.

Для простоты пусть это будет клиентским приложением Windows, хотя оно может быть частью более крупного веб-приложения или любого другого приложения. На рис. 12.1 показан пример экрана приложения.

Приложение берет ряд наименований покупок, их общую стоимость и применяет скидку на основе выбранного переключателя. В реальном приложении применялся бы несколько более изощренный способ вычисления скидки, но для данного примера достаточно и переключателей. Ниже приведен код для вычисления скидки:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
   string scriptToUse;
   if (CostRadioButton.IsChecked.Value)
   {
      scriptToUse = "AmountDisc.py";
   }
   else
   {
      scriptToUse = "CountDisc.py";
   }
   ScriptRuntime scriptRuntime = ScriptRuntime.CreateFromConfiguration();
   ScriptEngine pythEng = scriptRuntime.GetEngine("Python");
   ScriptSource source = pythEng.CreateScriptSourceFromFile(scriptToUse);
   ScriptScope scope = pythEng.CreateScope();
   scope.SetVariable("prodCount", Convert.ToInt32(totalItems.Text));
   scope.SetVariable("amt", Convert.ToDecimal(totalAmt.Text));
   source.Execute(scope);
   label5.Content = scope.GetVariable("retAmt").ToString();
}
```

Фрагмент кода **`Window1.xaml.cs`**

В первой части кода просто определяется, какой сценарий должен быть применен — `AmountDisc.py` или `CountDisc.py`. Сценарий `AmountDisc.py` вычисляет скидку на основе объема покупки:

```
discAmt = .25
retAmt = amt
if amt > 25.00:
  retAmt = amt-(amt*discAmt)
```

Минимальная сумма покупки, позволяющая претендовать на скидку, составляет $25. Если сумма меньше, никакой скидки нет, иначе предоставляется скидка в 25%.

Сценарий `CountDisc.py` вычисляет скидку на основе количества наименований приобретенного товара:

```
discCount = 5
discAmt = .1
retAmt = amt
if prodCount > discCount:
  retAmt = amt-(amt*discAmt)
```

В данном сценарии Python количество приобретенных наименований должно быть более 5, тогда будет предоставлена скидка в 10% от общей суммы.

Следующая часть кода касается установки среды ScriptRuntime. Здесь предпринимаются четыре специфических шага: создание объекта `ScriptRuntime`, его соответствующая настройка, создание `ScriptSource` и создание `ScriptScope`.

Объект `ScriptRuntime` — начальная точка, или база, хостинга. Он хранит глобальное состояние среды хостинга. `ScriptRuntime` создается с помощью статического метода `CreateFromConfiguration`. Ниже показано содержимое файла `app.config`:

```
<configuration>
  <configSections>
    <section
      name="microsoft.scripting"
      type="Microsoft.Scripting.Hosting.Configuration.Section,
             Microsoft.Scripting,
             Version=0.9.6.10,
             Culture=neutral,
             PublicKeyToken=null"
             requirePermission="false" />
  </configSections>
  <microsoft.scripting>
    <languages>
      <language
        names="IronPython;Python;py"
        extensions=".py"
        displayName="IronPython 2.6 Alpha"
        type="IronPython.Runtime.PythonContext,
               IronPython,
               Version=2.6.0.1,
               Culture=neutral,
               PublicKeyToken=null" />
    </languages>
  </microsoft.scripting>
</configuration>
```

Код определяет раздел `microsoft.scripting` и устанавливает несколько свойств для механизма сценариев IronPython.

Затем получается ссылка на `ScriptEngine` из `ScriptRuntime`. В данном примере указано, что нужен механизм Python, но `ScriptRuntime` мог бы определить это и самостоятельно — по расширению `py` сценария.

ScriptEngine проводит работы по выполнению кода сценария. Существует несколько методов для запуска сценариев из файлов либо из фрагментов кода. ScriptEngine также предоставляет ScriptSource и ScriptScope.

Объект ScriptSource — это то, что открывает доступ к сценарию. Он представляет исходный код сценария. С его помощью можно манипулировать источником сценария: загружать его с диска, разбирать по строкам и даже компилировать сценарий в объект CompiledCode. Последнее удобно, если один и тот же сценарий должен быть выполнен многократно.

Объект ScriptScope — это, по сути, пространство имен. Чтобы передать значение в сценарий и из него, потребуется привязать переменную к ScriptScope. В данном примере с помощью метода SetVariable осуществляется передача в сценарий Python переменных prodCount и amt. Эти значения берутся из текстовых полей totalItems и totalAmt. Вычисленная скидка извлекается из сценария посредством метода GetVariable. В данном примере переменная retAmt содержит искомое значение.

В обработчике кнопки Calc Tax (Вычислить налог) определяется, каким образом вызвать метод на объекте Python. Сценарий CalcTax.py — это очень простой метод, принимающий входную переменную, добавляющий 7,5% налога и возвращающий новое значение. Ниже показан его код:

```
def CalcTax(amount):
   return amount*1.075
```

А вот код C#, вызывающий метод CalcTax:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
   ScriptRuntime scriptRuntime = ScriptRuntime.CreateFromConfiguration();
   dynamic calcRate = scriptRuntime.UseFile("CalcTax.py");
   label6.Content = calcRate.CalcTax(Convert.ToDecimal(label5.Content)).ToString();
}
```

Как видите, процесс не особо сложен. Снова создается объект ScriptRuntime с использованием тех же конфигурационных настроек, что и прежде. CalcRate — объект ScriptScope. Он определен как dynamic, поэтому на нем можно вызывать метод CalcTax. Это пример того, как тип dynamic может несколько облегчить решение.

DynamicObject и ExpandoObject

Что, если необходимо создать собственный динамический объект? Для этого на выбор доступно два варианта: наследование от DynamicObject либо использование ExpandoObject. Использование DynamicObject требует немного больше работы, поскольку придется переопределить несколько методов. ExpandoObject представляет собой запечатанный (sealed) класс, готовый к применению.

DynamicObject

Рассмотрим объект, представляющий персону. Обычно при этом должны определяться свойства для имени, отчества и фамилии. Теперь предположит, что необходимо иметь возможность строить такой объект во время выполнения, на основе системы, не имеющей предварительного представления о том, какие свойства могут быть у объекта, и какие методы он может поддерживать. Это именно то, что может обеспечить объект, основанный на DynamicObject. Такая функциональность требуется очень редко, но до настоящего времени язык C# не позволял ее реализовать.

Для начала посмотрим, как выглядит DynamicObject:

```
class WroxDynamicObject: DynamicObject
{
    Dictionary<string, object> _dynamicData = new Dictionary<string, object>();
    public override bool TryGetMember(GetMemberBinder binder, out object result)
    {
        bool success = false;
        result = null;
        if (_dynamicData.ContainsKey(binder.Name))
        {
            result = _dynamicData[binder.Name];
            success = true;
        }
        else
        {
            result = "Property Not Found!";
            success = false;
        }
        return success;
    }
    public override bool TrySetMember(SetMemberBinder binder, object value)
    {
        _dynamicData[binder.Name] = value;
        return true;
    }
    public override bool TryInvokeMember(InvokeMemberBinder binder,
                                         object[] args,
                                         out object result)
    {
        dynamic method = _dynamicData[binder.Name];
        result = method((DateTime)args[0]);
        return result != null;
    }
}
```

Фрагмент кода **Dynamic\Program.cs**

В этом примере переопределяются три метода: `TrySetMember`, `TryGetMember` и `TryInvokeMember`.

`TrySetMember` добавляет новый метод, свойство или поле к объекту. В данном примере информация о членах сохраняется в объекте `Dictionary`. Объект `SetMemberBinder`, переданный в `TrySetMember`, содержит свойство `Name`, которое используется для идентификации элемента в `Dictionary`.

`TryGetMember` извлекает объект, хранящийся в `Dictionary`, на основе свойства `Name` из `GetMemberBinder`.

И что со всем этим делать? Ниже приведен код, в котором используется только что созданный динамический объект:

```
dynamic wroxDyn = new WroxDynamicObject();
wroxDyn.FirstName = "Bugs";
wroxDyn.LastName = "Bunny";

Console.WriteLine(wroxDyn.GetType());
Console.WriteLine("{0} {1}", wroxDyn.FirstName, wroxDyn.LastName);
```

Выглядит достаточно просто, но где же вызовы методов, которые вы переопределили? Здесь нам помогает .NET Framework. `DynamicObject` обрабатывает привязку, и все, что остается сделать — это сослаться на свойства `FirstName` и `LastName`, как если бы они существовали изначально.

А как насчет добавления метода? Это также несложно. Можно воспользоваться тем же `WroxDynamicObject` и добавить к нему метод `GetTomorrowDate`. Этот метод получает объект `DateTime` и возвращает строку даты следующего дня.

Ниже показан код:

```
dynamic wroxDyn = new WroxDynamicObject();
Func<DateTime, string> GetTomorrow = today => today.AddDays(1).ToShortDateString();

wroxDyn.GetTomorrowDate = GetTomorrow;
Console.WriteLine("Завтрашняя дата: {0}", wroxDyn.GetTomorrowDate(DateTime.Now));
```

Делегат `GetTomorrow` создается с использованием `Func<T, TResult>`. Метод, представленный делегатом — это вызов `AddDays`. К передаваемому значению `Date` прибавляется 1 день и возвращается строка полученной даты. Делегат затем устанавливается в `GetTomorrowDate` на объекте `wroxDyn`. В последней строке новый метод вызывается с передачей ему текущей даты.

Опять в дело вступает динамическая "магия", и получается объект с допустимым методом.

ExpandoObject

Объект `ExpandoObject` работает аналогично `WroxDynamicObject`, который был создан в предыдущем разделе. Отличие в том, что никаких методов переопределять не понадобится, что показано в следующем примере кода:

```
static void DoExpando()
{
   dynamic expObj = new ExpandoObject();
   expObj.FirstName = "Daffy";
   expObj.LastName = "Duck";
   Console.WriteLine(expObj.FirstName + " " + expObj.LastName);
   Func<DateTime, string> GetTomorrow = today => today.AddDays(1).ToShortDateString();
   expObj.GetTomorrowDate = GetTomorrow;
   Console.WriteLine("Tomorrow is {0}", expObj.GetTomorrowDate(DateTime.Now));
   expObj.Friends = new List<Person>();
   expObj.Friends.Add(new Person() { FirstName = "Bob", LastName = "Jones" });
   expObj.Friends.Add(new Person() { FirstName = "Robert", LastName = "Jones" });
   expObj.Friends.Add(new Person() { FirstName = "Bobby", LastName = "Jones" });
   foreach (Person friend in expObj.Friends)
   {
      Console.WriteLine(friend.FirstName + " " + friend.LastName);
   }
}
```

Обратите внимание, что этот код в основном идентичен тому, что делалось ранее. Вы добавляете свойства `FirstName` и `LastName`, затем добавляете функцию `GetTomorrow` и предпринимаете одно дополнительное действие — добавляете коллекцию объектов `Person` в качестве свойства объекта.

На первый взгляд может показаться, что это ничем не отличается от применения типа `dynamic`. На самом деле есть пара тонких отличий, которые важны. Во-первых, нельзя просто создать пустой объект типа `dynamic`. Объекту `dynamic` нужно обязательно что-то присвоить. Например, следующий код работать не будет:

```
dynamic dynObj;
dynObj.FirstName = "Joe";
```

Но для объекта `ExpandoObject`, как вы видели в предыдущем примере, это возможно.

Во-вторых, поскольку типу dynamic нужно что-нибудь присвоить, он вернет тип присвоенного ему объекта, если вы вызовете GetType. То есть, если ему присвоено значение int, он сообщит, что имеет тип int. Подобное не происходит с ExpandoObject или объектом-наследником DynamicObject.

Если нужно управлять добавлением и обращением к свойствам динамического объекта, то наилучшим выбором будет наследование от DynamicObject. Вместе с DynamicObject можно использовать несколько методов для переопределения и точного контроля того, как объект взаимодействует с исполняющей средой. Для прочих случаев подойдет тип dynamic или ExpandoObject.

Резюме

В этой главе было продемонстрировано использование нового типа dynamic. Также было показано, что делает компилятор, когда встречает тип dynamic. Мы обсудили исполняющую среду динамического языка (Dynamic Language Runtime), и вы научились встраивать ее в простое приложение. Чтобы задействовать DLR, использовались сценарии Python, с передачей и получаем от них значений. И, наконец, вы создали собственный динамический тип, унаследовав его класс от DynamicObject.

Динамическая разработка набирает популярность. Она позволяет делать такие вещи, которые очень трудно реализовать в статически типизированном языке. Тип dynamic и DLR позволяют программистам C# использовать некоторые динамические возможности.