Chapter 19. Dynamic Programming

Chapter 4 explained how dynamic binding works in the C# language. In this chapter, we look briefly at the *Dynamic Language Runtime* (DLR) and then explore the following dynamic programming patterns:

- Dynamic member overload resolution
- Custom binding (implementing dynamic objects)
- Dynamic language interoperability

> **NOTE**
>
> In Chapter 24, we describe how `dynamic` can improve COM interoperability.

The types in this chapter reside in the `System.Dynamic` namespace, except for `CallSite<>`, which resides in `System.Runtime.CompilerServices`.

The Dynamic Language Runtime

C# relies on the DLR to perform dynamic binding.

Contrary to its name, the DLR is not a dynamic version of the CLR. Rather, it's a library that sits atop the CLR—just like any other library such as *System.Xml.dll*. Its primary role is to provide runtime services to *unify* dynamic programming—in both statically and dynamically typed languages. Hence, languages such as C#, Visual Basic, IronPython, and IronRuby all use the same protocol for calling functions dynamically. This allows them to share libraries and call code written in other languages.

The DLR also makes it relatively easy to write new dynamic languages in .NET. Instead of having to emit Intermediate Language (IL), dynamic language authors work at the level of *expression trees* (the same expression trees in `System.Linq.Expressions` that we talked about in Chapter 8).

The DLR further ensures that all consumers get the benefit of *call-site caching*, an optimization whereby the DLR prevents unnecessarily repeating the potentially expensive member resolution decisions made during dynamic binding.

WHAT ARE CALL SITES?

When the compiler encounters a dynamic expression, it has no idea who will evaluate that expression at runtime. For instance, consider the following method:

```
public dynamic Foo (dynamic x, dynamic y)
{
  return x / y;   // Dynamic expression
}
```

The `x` and `y` variables could be any CLR object, a COM object, or even an object hosted in a dynamic language. The compiler cannot, therefore, take its usual static approach of emitting a call to a known method of a known type. Instead, the compiler emits code that eventually results in an expression tree that describes the operation, managed by a *call site* that the DLR will bind at runtime. The call site essentially acts as an intermediary between caller and callee.

A call site is represented by the `CallSite<>` class in *System.Core.dll*. We can see this by disassembling the preceding method—the result is something like this:

```
static CallSite<Func<CallSite,object,object,object>> divideSite;

[return: Dynamic]
public object Foo ([Dynamic] object x, [Dynamic] object y)
{
  if (divideSite == null)
    divideSite =
      CallSite<Func<CallSite,object,object,object>>.Create (
        Microsoft.CSharp.RuntimeBinder.Binder.BinaryOperation (
          CSharpBinderFlags.None,
          ExpressionType.Divide,
          /* Remaining arguments omitted for brevity */ ));

  return divideSite.Target (divideSite, x, y);
}
```

As you can see, the call site is cached in a static field to avoid the cost of re-creating it on each call. The DLR further caches the result of the binding phase and the actual method targets. (There can be multiple targets depending on the types of `x` and `y`.)

The actual dynamic call then happens by calling the site's `Target` (a delegate), passing in the `x` and `y` operands.

Notice that the `Binder` class is specific to C#. Every language with support for dynamic binding provides a language-specific binder to help the DLR interpret expressions in a manner specific to that language, so as not to surprise the programmer. For instance, if we called `Foo` with integer values of 5 and 2, the C# binder would ensure that we got back 2. In contrast, a VB.NET binder would give us 2.5.

Dynamic Member Overload Resolution

Calling a statically known method with dynamically typed arguments defers member overload resolution from compile time to runtime. This is useful in simplifying certain programming tasks—such as simplifying the *Visitor* design pattern. It's also useful in working around limitations imposed by C#'s static typing.

Simplifying the Visitor Pattern

In essence, the *Visitor* pattern allows you to "add" a method to a class hierarchy without altering existing classes. Although useful, this pattern in its static incarnation is subtle and unintuitive compared to most other design patterns. It also requires that visited classes be made "visitor-friendly" by exposing an `Accept` method, which can be impossible if the classes are not under your control.

With dynamic binding, you can achieve the same goal more easily—and without needing to modify existing classes. To illustrate, consider the following class hierarchy:

```
class Person
{
  public string FirstName { get; set; }
  public string LastName  { get; set; }

  // The Friends collection may contain Customers & Employees:
  public readonly IList<Person> Friends = new Collection<Person> ();
}

class Customer : Person { public decimal CreditLimit { get; set; } }
class Employee : Person { public decimal Salary      { get; set; } }
```

Suppose that we want to write a method that programmatically exports a `Person`'s details to an XML `XElement`. The most obvious solution is to write a virtual method called `ToXElement()` in the `Person` class that returns an `XElement` populated with a `Person`'s properties. We would then override it in `Customer` and `Employee` classes such that the `XElement` was also populated with `CreditLimit` and `Salary`. This pattern can be problematic, however, for two reasons:

- You might not own the `Person`, `Customer`, and `Employee` classes, making it impossible to add methods to them. (And extension methods wouldn't give polymorphic behavior.)
- The `Person`, `Customer`, and `Employee` classes might already be quite big. A frequent antipattern is the "God Object," in which a class such as `Person` attracts so much functionality that it becomes a nightmare to maintain. A good antidote is to avoid adding functions to `Person` that don't need to access `Person`'s private state. A `ToXElement` method might be an excellent candidate.

With dynamic member overload resolution, we can write the `ToXElement` functionality in a separate class, without resorting to ugly switches based on type:

```
class ToXElementPersonVisitor
{
  public XElement DynamicVisit (Person p) => Visit ((dynamic)p);
```

```
    XElement Visit (Person p)
    {
      return new XElement ("Person",
        new XAttribute ("Type", p.GetType().Name),
        new XElement ("FirstName", p.FirstName),
        new XElement ("LastName", p.LastName),
        p.Friends.Select (f => DynamicVisit (f))
      );
    }

    XElement Visit (Customer c)   // Specialized logic for customers
    {
      XElement xe = Visit ((Person)c);   // Call "base" method
      xe.Add (new XElement ("CreditLimit", c.CreditLimit));
      return xe;
    }

    XElement Visit (Employee e)   // Specialized logic for employees
    {
      XElement xe = Visit ((Person)e);   // Call "base" method
      xe.Add (new XElement ("Salary", e.Salary));
      return xe;
    }
  }
```

The `DynamicVisit` method performs a dynamic dispatch—calling the most specific version of `Visit` as determined at runtime. Notice the line in boldface, in which we call `DynamicVisit` on each person in the `Friends` collection. This ensures that if a friend is a `Customer` or `Employee`, the correct overload is called.

We can demonstrate this class, as follows:

```
var cust = new Customer
{
  FirstName = "Joe", LastName = "Bloggs", CreditLimit = 123
};
cust.Friends.Add (
  new Employee { FirstName = "Sue", LastName = "Brown", Salary = 50000 }
);

Console.WriteLine (new ToXElementPersonVisitor().DynamicVisit (cust));
```

Here’s the result:

```
<Person Type="Customer">
  <FirstName>Joe</FirstName>
  <LastName>Bloggs</LastName>
  <Person Type="Employee">
    <FirstName>Sue</FirstName>
    <LastName>Brown</LastName>
    <Salary>50000</Salary>
  </Person>
  <CreditLimit>123</CreditLimit>
</Person>
```

Variations

If you plan more than one visitor class, a useful variation is to define an abstract base class for visitors:

```
abstract class PersonVisitor<T>
{
  public T DynamicVisit (Person p) { return Visit ((dynamic)p); }

  protected abstract T Visit (Person p);
  protected virtual T Visit (Customer c) { return Visit ((Person) c); }
  protected virtual T Visit (Employee e) { return Visit ((Person) e); }
}
```

Subclasses then don't need to define their own `DynamicVisit` method: all they do is override the versions of `Visit` whose behavior they want to specialize. This also has the advantages of centralizing the methods that encompass the `Person` hierarchy and allowing implementers to call base methods more naturally:

```
class ToXElementPersonVisitor : PersonVisitor<XElement>
{
  protected override XElement Visit (Person p)
  {
    return new XElement ("Person",
      new XAttribute ("Type", p.GetType().Name),
      new XElement ("FirstName", p.FirstName),
      new XElement ("LastName", p.LastName),
      p.Friends.Select (f => DynamicVisit (f))
    );
  }
```

```
    protected override XElement Visit (Customer c)
    {
      XElement xe = base.Visit (c);
      xe.Add (new XElement ("CreditLimit", c.CreditLimit));
      return xe;
    }

    protected override XElement Visit (Employee e)
    {
      XElement xe = base.Visit (e);
      xe.Add (new XElement ("Salary", e.Salary));
      return xe;
    }
  }
```

You then can even subclass `ToXElementPersonVisitor` itself.

Anonymously Calling Members of a Generic Type

The strictness of C#'s static typing is a double-edged sword. On the one hand, it enforces a degree of correctness at compile time. On the other hand, it occasionally makes certain kinds of code difficult or impossible to express, at which point you must resort to reflection. In these situations, dynamic binding is a cleaner and faster alternative to reflection.

An example is when you need to work with an object of type `G<T>` where `T` is unknown. We can illustrate this by defining the following class:

```
public class Foo<T> { public T Value; }
```

Suppose that we then write a method as follows:

```
static void Write (object obj)
{
  if (obj is Foo<>)                           // Illegal
    Console.WriteLine ((Foo<>) obj).Value);   // Illegal
}
```

This method won't compile: you can't invoke members of *unbound* generic types.

Dynamic binding offers two means by which we can work around this. The first is to access the `Value` member dynamically as follows:

```
static void Write (dynamic obj)
{
  try { Console.WriteLine (obj.Value); }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
}
```

MULTIPLE DISPATCH

C# and the CLR have always supported a limited form of dynamism in the form of virtual method calls. This differs from C#'s `dynamic` binding in that for virtual method calls, the compiler must commit to a particular virtual member at compile time—based on the name and signature of a member you called. This means that:

- The calling expression must be fully understood by the compiler (e.g., it must decide at compile time whether a target member is a field or property).
- Overload resolution must be completed entirely by the compiler, based on the compile-time argument types.

A consequence of that last point is that the ability to perform virtual method calls is known as *single dispatch*. To see why, consider the following method call (in which `Walk` is a virtual method):

```
animal.Walk (owner);
```

The runtime decision of whether to invoke a dog's `Walk` method or a cat's `Walk` method depends only on the type of the *receiver*, `animal` (hence, "single"). If many overloads of `Walk` accept different kinds of owner, an overload will be selected at compile time without regard to the actual runtime type of the `owner` object. In other words, only the runtime type of the *receiver* can vary which method gets called.

In contrast, a dynamic call defers overload resolution until runtime:

```
animal.Walk ((dynamic) owner);
```

The final choice of which `Walk` method to call now depends on the types of both `animal` and `owner`—this is called *multiple dispatch* because the runtime types of arguments, in addition to the receiver type, contribute to the determination of which `Walk` method to call.

This has the (potential) advantage of working with any object that defines a `Value` field or property. However, there are a couple of problems. First, catching an exception in this manner is somewhat messy and inefficient (and there's no way to ask the DLR in advance, "Will this operation succeed?"). Second, this approach wouldn't work if `Foo` were an interface (say, `IFoo<T>`) and either of the following conditions were true:

- `Value` was implemented explicitly
- The type that implemented `IFoo<T>` was inaccessible (more on this soon)

A better solution is to write an overloaded helper method called `GetFooValue` and to call it using *dynamic member overload resolution*:

```
static void Write (dynamic obj)
{
  object result = GetFooValue (obj);
  if (result != null) Console.WriteLine (result);
}

static T GetFooValue<T> (Foo<T> foo) => foo.Value;
static object GetFooValue (object foo) => null;
```

Notice that we overloaded `GetFooValue` to accept an `object` parameter, which acts as a fallback for any type. At runtime, the C# dynamic binder will pick the best overload when calling `GetFooValue` with a dynamic argument. If the object in question is not based on `Foo<T>`, it will choose the object-parameter overload instead of throwing an exception.

NOTE

An alternative is to write just the first `GetFooValue` overload and then catch the `RuntimeBinderException`. The advantage is that it distinguishes the case of `foo.Value` being null. The disadvantage is that it incurs the performance overhead of throwing and catching an exception.

In Chapter 18, we solved the same problem with an interface using reflection—with a lot more effort (see "Anonymously Calling Members of a Generic Interface"). The example we used was to design a more powerful version of `ToString()` that could understand objects such as `IEnumerable` and `IGrouping<,>`. Here's the same example solved more elegantly using dynamic binding:

```
static string GetGroupKey<TKey,TElement> (IGrouping<TKey,TElement> group)
  => "Group with key=" + group.Key + ": ";

static string GetGroupKey (object source) => null;

public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  if (value is string s) return s;
  if (value.GetType().IsPrimitive) return value.ToString();

  StringBuilder sb = new StringBuilder();

  string groupKey = GetGroupKey ((dynamic)value);   // Dynamic dispatch
  if (groupKey != null) sb.Append (groupKey);

  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append (ToStringEx (element) + " ");

  if (sb.Length == 0) sb.Append (value.ToString());

  return "\r\n" + sb.ToString();
}
```

Here it is in action:

```
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c) ));

Group with key=x: x
Group with key=y: y y
Group with key=z: z z z
```

Notice that we used dynamic *member overload resolution* to solve this problem. If we instead did the following:

```
dynamic d = value;
try { groupKey = d.Value); }
catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
```

it would fail because LINQ's `GroupBy` operator returns a type implementing `IGrouping<,>`, which itself is internal and therefore inaccessible:

```
internal class Grouping : IGrouping<TKey,TElement>, ...
{
  public TKey Key;
  ...
}
```

Even though the `Key` property is declared `public`, its containing class caps it at `internal`, making it accessible only via the `IGrouping<,>` interface. And as is explained in Chapter 4, there's no way to instruct the DLR to bind to that interface when invoking the `Value` member dynamically.

Implementing Dynamic Objects

An object can provide its binding semantics by implementing `IDynamicMetaObjectProvider`—or more easily by subclassing `DynamicObject`, which provides a default implementation of this interface. This is demonstrated briefly in Chapter 4 via the following example:

```
dynamic d = new Duck();
d.Quack();                  // Quack method was called
d.Waddle();                 // Waddle method was called

public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

DynamicObject

In the preceding example, we overrode `TryInvokeMember`, which allows the consumer to invoke a method on the dynamic object—such as a `Quack` or `Waddle`. `DynamicObject` exposes other virtual methods that enable consumers to use other programming constructs as well. The following correspond to constructs that have representations in C#:

Method	Programming construct
`TryInvokeMember`	Method
`TryGetMember`, `TrySetMember`	Property or field
`TryGetIndex`, `TrySetIndex`	Indexer
`TryUnaryOperation`	Unary operator such as `!`
`TryBinaryOperation`	Binary operator such as `==`
`TryConvert`	Conversion (cast) to another type
`TryInvoke`	Invocation on the object itself—e.g., `d("foo")`

These methods should return `true` if successful. If they return `false`, the DLR will fall back to the language binder, looking for a matching member on the `DynamicObject` (subclass) itself. If this fails, a `RuntimeBinderException` is thrown.

We can illustrate `TryGetMember` and `TrySetMember` with a class that lets us dynamically access an attribute in an `XElement` (`System.Xml.Linq`):

```
static class XExtensions
{
  public static dynamic DynamicAttributes (this XElement e)
    => new XWrapper (e);
  class XWrapper : DynamicObject
  {
    XElement _element;
    public XWrapper (XElement e) { _element = e; }

    public override bool TryGetMember (GetMemberBinder binder,
                                       out object result)
    {
      result = _element.Attribute (binder.Name).Value;
      return true;
    }

    public override bool TrySetMember (SetMemberBinder binder,
                                       object value)
    {
      _element.SetAttributeValue (binder.Name, value);
      return true;
    }
  }
}
```

Here's how to use it:

```
XElement x = XElement.Parse (@"<Label Text=""Hello"" Id=""5""/>");
dynamic da = x.DynamicAttributes();
Console.WriteLine (da.Id);          // 5
da.Text = "Foo";
Console.WriteLine (x.ToString());   // <Label Text="Foo" Id="5" />
```

The following does a similar thing for `System.Data.IDataRecord`, making it easier to use data readers:

```
public class DynamicReader : DynamicObject
{
  readonly IDataRecord _dataRecord;
  public DynamicReader (IDataRecord dr) { _dataRecord = dr; }

  public override bool TryGetMember (GetMemberBinder binder,
                                     out object result)
  {
    result = _dataRecord [binder.Name];
    return true;
  }
}
...
```

```
using (IDataReader reader = someDbCommand.ExecuteReader())
{
  dynamic dr = new DynamicReader (reader);
  while (reader.Read())
  {
    int id = dr.ID;
    string firstName = dr.FirstName;
    DateTime dob = dr.DateOfBirth;
    ...
  }
}
```

The following demonstrates `TryBinaryOperation` and `TryInvoke`:

```
dynamic d = new Duck();
Console.WriteLine (d + d);          // foo
Console.WriteLine (d (78, 'x'));    // 123

public class Duck : DynamicObject
{
  public override bool TryBinaryOperation (BinaryOperationBinder binder,
                                           object arg, out object result)
  {
    Console.WriteLine (binder.Operation);   // Add
    result = "foo";
    return true;
  }

  public override bool TryInvoke (InvokeBinder binder,
                                  object[] args, out object result)
  {
    Console.WriteLine (args[0]);    // 78
    result = 123;
    return true;
  }
}
```

`DynamicObject` also exposes some virtual methods for the benefit of dynamic languages. In particular, overriding `GetDynamicMemberNames` allows you to return a list of all member names that your dynamic object provides.

NOTE

Another reason to implement `GetDynamicMemberNames` is that Visual Studio's debugger makes use of this method to display a view of a dynamic object.

ExpandoObject

Another simple application of `DynamicObject` would be to write a dynamic class that stored and retrieved objects in a dictionary, keyed by string. However, this functionality is already provided via the `ExpandoObject` class:

```
dynamic x = new ExpandoObject();
x.FavoriteColor = ConsoleColor.Green;
x.FavoriteNumber = 7;
Console.WriteLine (x.FavoriteColor);    // Green
Console.WriteLine (x.FavoriteNumber);   // 7
```

`ExpandoObject` implements `IDictionary<string,object>`—so we can continue our example and do this:

```
var dict = (IDictionary<string,object>) x;
Console.WriteLine (dict ["FavoriteColor"]);    // Green
Console.WriteLine (dict ["FavoriteNumber"]);   // 7
Console.WriteLine (dict.Count);                // 2
```

Interoperating with Dynamic Languages

Although C# supports dynamic binding via the `dynamic` keyword, it doesn't go as far as allowing you to execute an expression described in a string at runtime:

```
string expr = "2 * 3";
// We can't "execute" expr
```

This is because the code to translate a string into an expression tree requires a lexical and semantic parser. These features are built into the C# compiler and are not available as a runtime service. At runtime, C# merely provides a *binder*, which instructs the DLR how to interpret an already-built expression tree.

True dynamic languages such as IronPython and IronRuby do allow you to execute an arbitrary string, and this is useful in tasks such as scripting, writing dynamic configuration systems, and implementing dynamic rules engines. So, although you can write most of your application in C#, it can be useful to call out to a dynamic language for such tasks. In addition, you might want to use an API that is written in a dynamic language where no equivalent functionality is available in a .NET library.

NOTE

The Roslyn scripting NuGet package *Microsoft.CodeAnalysis.CSharp.Scripting* provides an API that lets you execute a C# string, although it does so by first compiling your code into a program. The compilation overhead makes it slower than Python interop, unless you intend to execute the same expression repeatedly.

In the following example, we use IronPython to evaluate an expression created at runtime from within C#. You could use the following script to write a calculator.

NOTE

To run this code, add the NuGet packages *DynamicLanguageRuntime* (not to be confused with the *System.Dynamic.Runtime* package) and *IronPython* to your application.

```
using System;
using IronPython.Hosting;
using Microsoft.Scripting;
using Microsoft.Scripting.Hosting;
```

```
int result = (int) Calculate ("2 * 3");
Console.WriteLine (result);             // 6

object Calculate (string expression)
{
  ScriptEngine engine = Python.CreateEngine();
  return engine.Execute (expression);
}
```

Because we're passing a string into Python, the expression will be evaluated according to Python's rules and not C#'s. It also means that we can use Python's language features, such as lists:

```
var list = (IEnumerable) Calculate ("[1, 2, 3] + [4, 5]");
foreach (int n in list) Console.Write (n);  // 12345
```

Passing State Between C# and a Script

To pass variables from C# to Python, a few more steps are required. The following example illustrates those steps and could be the basis of a rules engine:

```
// The following string could come from a file or database:
string auditRule = "taxPaidLastYear / taxPaidThisYear > 2";

ScriptEngine engine = Python.CreateEngine ();

ScriptScope scope = engine.CreateScope ();
scope.SetVariable ("taxPaidLastYear", 20000m);
scope.SetVariable ("taxPaidThisYear", 8000m);

ScriptSource source = engine.CreateScriptSourceFromString (
                      auditRule, SourceCodeKind.Expression);

bool auditRequired = (bool) source.Execute (scope);
Console.WriteLine (auditRequired);   // True
```

You can also get variables back by calling `GetVariable`:

```
string code = "result = input * 3";
```

```
ScriptEngine engine = Python.CreateEngine();

ScriptScope scope = engine.CreateScope();
scope.SetVariable ("input", 2);

ScriptSource source = engine.CreateScriptSourceFromString (code,
                                  SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (scope.GetVariable ("result"));   // 6
```

Notice that we specified `SourceCodeKind.SingleStatement` in the second example (rather than `Expression`) to inform the engine that we want to execute a statement.

Types are automatically marshaled between the .NET and Python worlds. You can even access members of .NET objects from the scripting side:

```
string code = @"sb.Append (""World"")";

ScriptEngine engine = Python.CreateEngine ();

ScriptScope scope = engine.CreateScope ();
var sb = new StringBuilder ("Hello");
scope.SetVariable ("sb", sb);

ScriptSource source = engine.CreateScriptSourceFromString (
                      code, SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (sb.ToString());   // HelloWorld
```