18

Reflection, Attributes, and Dynamic Programming

Attributes are a means of inserting additional metadata into an assembly and associating the metadata with a programming construct such as a class, method, or property. This chapter investigates the details surrounding attributes that are built into the framework and describes how to define custom attributes. To take advantage of custom attributes, it is necessary to identify them. This is handled through reflection. This chapter begins with a look at reflection, including how you can use it to dynamically bind at execution time based on member invocation by name (or metadata) at compile time. Reflection is frequently leveraged within tools such as a code generator. In addition, reflection is used at execution time when the call target is unknown.

The chapter ends with a discussion of dynamic programming, a feature added in C# 4.0 that greatly simplifies working with data that is dynamic and requires execution-time rather than compile-time binding.

\*\*\*COMP: Insert 18mindmap pick up from previous edition 978-1-5093-0358-8 p. 721

|  |
| --- |
| Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_Mindmaps:18mindmap.pdf |

Reflection

Using reflection, it is possible to do the following.

* Access the metadata for types within an assembly. This includes constructs such as the full type name, member names, and any attributes decorating the construct.

Dynamically invoke a type’s members at runtime using the metadata, rather than a compile-time–defined binding.

Reflection is the process of examining the metadata within an assembly. Traditionally, when code compiles down to a machine language, all the metadata (such as type and method names) about the code is discarded. In contrast, when C# compiles into the Common Intermediate Language (CIL), it maintains most of the metadata about the code. Furthermore, using reflection, it is possible to enumerate through all the types within an assembly and search for those that match certain criteria. You access a type’s metadata through instances of System.Type, and this object includes methods for enumerating the type instance’s members. Additionally, it is possible to invoke those members on objects that are of the examined type.

The facility for reflection enables a host of new paradigms that otherwise are unavailable. For example, reflection enables you to enumerate over all the types within an assembly, along with their members, and in the process create stubs for documentation of the assembly API. You can then combine the metadata retrieved from reflection with the XML document created from XML comments (using the /doc switch) to create the API documentation. Similarly, programmers use reflection metadata to generate code for persisting (serializing) business objects into a database. It could also be used in a list control that displays a collection of objects. Given the collection, a list control could use reflection to iterate over all the properties of an object in the collection, defining a column within the list for each property. Furthermore, by invoking each property on each object, the list control could populate each row and column with the data contained in the object, even though the data type of the object is unknown at compile time.

XmlSerializer, ValueType, and the Microsoft .NET Framework’s DataBinder are a few of the classes in the framework that use reflection for portions of their implementation as well.

Accessing Metadata Using System.Type

The key to reading a type’s metadata is to obtain an instance of System.Type that represents the target type instance. System.Type provides all the methods for retrieving the information about a type. You can use it to answer questions such as the following:

* What is the type’s name (Type.Name)?

Is the type public (Type.IsPublic)?

What is the type’s base type (Type.BaseType)?

Does the type support any interfaces (Type.GetInterfaces())?

Which assembly is the type defined in (Type.Assembly)?

What are a type’s properties, methods, fields, and so on (Type.GetProperties(), Type.GetMethods(), Type.GetFields(), and so on)?

Which attributes decorate a type (Type.GetCustomAttributes())?

There are more such members, but all of them provide information about a particular type. The key is to obtain a reference to a type’s Type object, and the two primary ways to do so are through object.GetType() and typeof().

Note that the GetMethods() call does not return extension methods. These methods are available only as static members on the implementing type.

GetType()

object includes a GetType() member, and therefore, all types include this function. You call GetType() to retrieve an instance of System.Type corresponding to the original object. Listing 18.1 demonstrates this process, using a Type instance from DateTime. Output 18.1 shows the results.

Listing 18.1: Using Type.GetProperties() to Obtain an Object’s Public Properties

DateTime dateTime = new DateTime();

Type type = dateTime.GetType();

foreach (

System.Reflection.PropertyInfo property in

type.GetProperties())

{

Console.WriteLine(property.Name);

}

|  |
| --- |
| Output 18.1 |
| Date  Day  DayOfWeek  DayOfYear  Hour  Kind  Millisecond  Minute  Month  Now  UtcNow  Second  Ticks  TimeOfDay  Today  Year |

After calling GetType(), you iterate over each System.Reflection.PropertyInfo instance returned from Type.GetProperties() and display the property names. The key to calling GetType() is that you must have an object instance. However, sometimes no such instance is available. Static classes, for example, cannot be instantiated, so there is no way to call GetType().

typeof()

Another way to retrieve a Type object is with the typeof expression. typeof binds at compile time to a particular Type instance, and it takes a type directly as a parameter. The exception is for the type parameter on a generic type as it isn't determined until runtime. Listing 18.2 demonstrates the use of typeof with Enum.Parse().

Listing 18.2: Using typeof() to create a System.Type instance

using System.Diagnostics;

// ...

ThreadPriorityLevel priority;

priority = (ThreadPriorityLevel)Enum.Parse(

typeof(ThreadPriorityLevel), "Idle");

// ...

In this listing, Enum.Parse() takes a Type object identifying an enum and then converts a string to the specific enum value. In this case, it converts "Idle" to System.Diagnostics.ThreadPriorityLevel.Idle.

Similarly, Listing 18.3 in the next section uses the typeof expression inside the CompareTo(object obj) method to verify that the type of the obj parameter was indeed what was expected:

if(obj.GetType() != typeof(Contact)) { ... }

The typeof expression is resolved at compile time such that a type comparison—perhaps comparing the type returned from a call to GetType()—can determine if an object is of a specific type.

Member Invocation

The possibilities with reflection don’t stop with retrieving the metadata. The next possible step is to take the metadata and dynamically invoke the members it references. Consider the possibility of defining a class to represent an application’s command line.[[1]](#footnote-1) The difficulty with a CommandLineInfo class such as this relates to populating the class with the actual command-line data that started the application. However, using reflection, you can map the command-line options to property names and then dynamically set the properties at runtime. Listing 18.3 demonstrates this process.

Listing 18.3: Dynamically Invoking a Member

using System;

using System.Diagnostics;

public partial class Program

{

public static void Main(string[] args)

{

CommandLineInfo commandLine = new CommandLineInfo();

if (!CommandLineHandler.TryParse(

args, commandLine, out string? errorMessage))

{

Console.WriteLine(errorMessage);

DisplayHelp();

}

if (commandLine.Help)

{

DisplayHelp();

}

else

{

if (commandLine.Priority !=

ProcessPriorityClass.Normal)

{

// Change thread priority

}

}

// ...

}

private static void DisplayHelp()

{

// Display the command-line help

Console.WriteLine(

"Compress.exe /Out:<filename> /Help \n"

+ "/Priority:RealTime|High|"

+ "AboveNormal|Normal|BelowNormal|Idle");

}

}

using System;

using System.Diagnostics;

public partial class Program

{

private class CommandLineInfo

{

public bool Help { get; set; }

public string? Out { get; set; }

public ProcessPriorityClass Priority { get; set; }

= ProcessPriorityClass.Normal;

}

}

using System;

using System.Diagnostics;

using System.IO;

using System.Reflection;

public class CommandLineHandler

{

public static void Parse(string[] args, object commandLine)

{

if (!TryParse(args, commandLine, out string? errorMessage))

{

throw new InvalidOperationException(errorMessage);

}

}

public static bool TryParse(string[] args, object commandLine,

out string? errorMessage)

{

bool success = false;

errorMessage = null;

foreach (string arg in args)

{

string option;

if (arg[0] == '/' || arg[0] == '-')

{

string[] optionParts = arg.Split(

new char[] { ':' }, 2);

// Remove the slash|dash

option = optionParts[0].Remove(0, 1);

PropertyInfo? property =

commandLine.GetType().GetProperty(option,

BindingFlags.IgnoreCase |

BindingFlags.Instance |

BindingFlags.Public);

if (property != null)

{

if (property.PropertyType == typeof(bool))

{

// Last parameters for handling indexers

property.SetValue(

commandLine, true, null);

success = true;

}

else if (

property.PropertyType == typeof(string))

{

property.SetValue(

commandLine, optionParts[1], null);

success = true;

}

else if (

// property.PropertyType.IsEnum also available

property.PropertyType ==

typeof(ProcessPriorityClass))

{

try

{

property.SetValue(commandLine,

Enum.Parse(

typeof(ProcessPriorityClass),

optionParts[1], true),

null);

success = true;

}

catch (ArgumentException )

{

success = false;

errorMessage =

errorMessage =

$@"The option '{

optionParts[1]

}' is invalid for '{

option }'";

}

}

else

{

success = false;

errorMessage =

$@"Data type '{

property.PropertyType

}' on {

commandLine.GetType()

} is not supported."

}

}

else

{

success = false;

errorMessage =

$"Option '{ option }' is not supported.";

}

}

}

return success;

}

}

Although Listing 18.3 is long, the code is relatively simple. Main() begins by instantiating a CommandLineInfo class. This type is defined specifically to contain the command-line data for this program. Each property corresponds to a command-line option for the program, where the command line is as shown in Output 18.2.

|  |
| --- |
| Output 18.2 |
| Compress.exe /Out:<file name> /Help  /Priority:RealTime|High|AboveNormal|Normal|BelowNormal|Idle |

The CommandLineInfo object is passed to the CommandLineHandler’s TryParse() method. This method begins by enumerating through each option and separating out the option name (e.g., Help or Out). Once the name is determined, the code reflects on the CommandLineInfo object, looking for an instance property with the same name. If the property is found, it assigns the property using a call to SetValue() and specifies the data corresponding to the property type. (For arguments, this call accepts the object on which to set the value, the new value, and an additional index parameter that is null unless the property is an indexer.) This listing handles three property types: Boolean, string, and enum. In the case of enums, you parse the option value and assign the property the text’s enum equivalent. Assuming the TryParse() call was successful, the method exits and the CommandLineInfo object is initialized with the data from the command line.

Interestingly, although CommandLineInfo is a private class nested within Program, CommandLineHandler has no trouble reflecting over it and even invoking its members. In other words, reflection can circumvent accessibility rules as long as appropriate permissions are established. If, for example, Out was private, it would still be possible for the TryParse() method to assign it a value. Because of this, it would be possible to move CommandLineHandler into a separate assembly and share it across multiple programs, each with its own CommandLineInfo class.

In this example, you invoke a member on CommandLineInfo using PropertyInfo.SetValue(). Not surprisingly, PropertyInfo also includes a GetValue() method for retrieving data from the property. For a method, however, there is a MethodInfo class with an Invoke() member. Both MethodInfo and PropertyInfo derive from MemberInfo (albeit indirectly), as shown in Figure 18.1.

\*\*\*COMP: Insert 18fig01

|  |
| --- |
| Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_NumberedFigures:18fig01.pdf |
| Figure 18.1: MemberInfo Derived Classes |

\*\*\*COMP: Insert “Begin 2.0” tab

Reflection on Generic Types

The introduction of generic types in version 2.0 of the Common Language Runtime (CLR) necessitated additional reflection features. Runtime reflection on generics determines whether a class or method contains a generic type and any type parameters or arguments it may include.

Determining the Type of Type Parameters

In the same way that you can use a typeof operator with non-generic types to retrieve an instance of System.Type, so you can use the typeof operator on type parameters in a generic type or generic method. Listing 18.4 applies the typeof operator to the type parameter in the Add method of a Stack class.

Listing 18.4: Declaring the Stack<T> Class

public class Stack<T>

{

// ...

public void Add(T i)

{

// ...

Type t = typeof(T);

// ...

}

// ...

}

Once you have an instance of the Type object for the type parameter, you may then use reflection on the type parameter itself to determine its behavior and tailor the Add method to the specific type more effectively.

Determining Whether a Class or Method Supports Generics

In the System.Type class for the version 2.0 release of the CLR, a handful of methods were added that determine whether a given type supports generic parameters and arguments. A generic argument is a type parameter supplied when a generic class is instantiated. You can determine whether a class or method contains generic parameters that have not yet been set by querying the Type.ContainsGenericParameters property, as demonstrated in Listing 18.5.

Listing 18.5: Reflection with Generics

using System;

public class Program

{

static void Main()

{

Type type = typeof(System.Nullable<>);

Console.WriteLine(type.ContainsGenericParameters);

Console.WriteLine(type.IsGenericType);

type = typeof(System.Nullable<DateTime>);

Console.WriteLine(type.ContainsGenericParameters);

Console.WriteLine(type.IsGenericType);

}

}

Output 18.3 shows the results of Listing 18.5.

|  |
| --- |
| Output 18.3 |
| True  True  False  True |

Type.IsGenericType is a Boolean property that evaluates whether a type is generic.

Obtaining Type Parameters for a Generic Class or Method

You can obtain a list of generic arguments, or type parameters, from a generic class by calling the GetGenericArguments() method. The result is an array of System.Type instances that corresponds to the order in which they are declared as type parameters of the generic class. Listing 18.6 reflects into a generic type and obtains each type parameter; Output 18.4 shows the results.

Listing 18.6: Using Reflection with Generic Types

using System;

using System.Collections.Generic;

public partial class Program

{

public static void Main()

{

Stack<int> s = new Stack<int>();

Type t = s.GetType();

foreach(Type type in t.GetGenericArguments())

{

System.Console.WriteLine(

"Type parameter: " + type.FullName);

}

// ...

}

}

|  |
| --- |
| Output 18.4 |
| Type parameter: System.Int32 |

\*\*\*COMP: Insert “End 2.0” tab

\*\*\*COMP: Insert “Begin 6.0” tab

nameof Operator

We briefly touched on the nameof operator in Chapter 11, where it was used to provide the name of a parameter in an argument exception:

throw new ArgumentException(

"The argument did not represent a digit", nameof(textDigit));

Introduced in C# 6.0, this contextual keyword produces a constant string containing the unqualified name of whatever program element is specified as an argument. In this case, textDigit is a parameter to the method, so nameof(textDigit) returns “textDigit.” (Given that this activity happens at compile time, nameof is not technically reflection. We include it here because ultimately it receives data about the assembly and its structure.)

You might ask what advantage is gained by using nameof(textDigit) over simply "textDigit" (especially given that the latter might even seem easier to use to some programmers). The advantages are twofold:

* The C# compiler ensures that the argument to the nameof operator is, in fact, a valid program element. This helps prevent errors when a program element name is changed, helps prevent misspellings, and so on.

IDE tools work better with the nameof operator than with literal strings. For example, the “find all references” tool will find program elements mentioned in a nameof expression but not in a literal string. The automatic renaming refactoring also works better, and so on.

In the snippet given earlier, nameof(textDigit) produces the name of a parameter. However, the nameof operator works with any program element. For example, Listing 18.7 uses nameof to pass the property name to INotifyPropertyChanged.PropertyChanged.

Listing 18.7: Dynamically Invoking a Member

using System.ComponentModel;

public class Person : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

public Person(string name)

{

Name = name;

}

private string \_Name = string.Empty;

public string Name

{

get { return \_Name; }

set

{

if (\_Name != value)

{

\_Name = value;

// Using C# 6.0 conditional null reference

PropertyChanged?.Invoke(

this,

new PropertyChangedEventArgs(

nameof(Name)));

}

}

}

// ...

}

Notice that whether only the unqualified “Name” is provided (because it’s in scope) or the fully (or partially) qualified name such as Person.Name is used, the result is only the final identifier (the last element in a dotted name).

You can still use C# 5.0’s CallerMemberName parameter attribute to obtain a property’s name; see http://itl.tc/CallerMemberName for an example.

\*\*\*COMP: Insert “End 6.0” tab

Attributes

Before delving into details on how to program attributes, we should consider a use case that demonstrates their utility. In the CommandLineHandler example in Listing 18.3, you dynamically set a class’s properties based on the command-line option matching the property name. This approach is insufficient, however, when the command-line option is an invalid property name. /?, for example, cannot be supported. Furthermore, this mechanism doesn’t provide any way of identifying which options are required versus which are optional.

Instead of relying on an exact match between the option name and the property name, you can use attributes to identify additional metadata about the decorated construct—in this case, the option that the attribute decorates. With attributes, you can decorate a property as Required and provide a /? option alias. In other words, attributes are a means of associating additional data with a property (and other constructs).

Attributes appear within square brackets preceding the construct they decorate. For example, you can modify the CommandLineInfo class to include attributes, as shown in Listing 18.8.

Listing 18.8: Decorating a Property with an Attribute

class CommandLineInfo

{

[CommandLineSwitchAlias("?")]

public bool Help { get; set; }

[CommandLineSwitchRequired]

public string? Out { get; set; }

public System.Diagnostics.ProcessPriorityClass Priority

{ get; set; } =

System.Diagnostics.ProcessPriorityClass.Normal;

}

In Listing 18.8, the Help and Out properties are decorated with attributes. The purpose of these attributes is to allow an alias of /? for /Help, and to indicate that /Out is a required parameter. The idea is that from within the CommandLineHandler.TryParse() method, you enable support for option aliases and, assuming the parsing was successful, you check that all required switches were specified.

There are two ways to combine attributes on the same construct. First, you can separate the attributes with commas within the same square brackets. Alternatively, you can place each attribute within its own square brackets. Listing 18.9 provides examples.

Listing 18.9: Decorating a Property with Multiple Attributes

[CommandLineSwitchRequired]

[CommandLineSwitchAlias("FileName")]

public string? Out { get; set; }

[CommandLineSwitchRequired,

CommandLineSwitchAlias("FileName")]

public string Out { get; set; }

In addition to decorating properties, developers can use attributes to decorate assemblies, classes, constructors, delegates, enums, events, fields, generic parameters, interfaces, methods, modules, parameters, properties, return values, and structs. For the majority of these cases, applying an attribute involves the same square bracket syntax shown in Listing 18.9. However, this syntax doesn’t work for return values, assemblies, and modules.

Assembly attributes are used to add metadata about the assembly. Visual Studio’s Project Wizard for .NET Framework projects (though not .NET Core generated projects), for example, generates an AssemblyInfo.cs file that includes numerous attributes about the assembly. Listing 18.10 is an example of such a file.

Listing 18.10: Assembly Attributes within AssemblyInfo.cs

using System.Reflection;

using System.Runtime.CompilerServices;

using System.Runtime.InteropServices;

// General information about an assembly is controlled

// through the following set of attributes. Change these

// attribute values to modify the information

// associated with an assembly.

[assembly: AssemblyTitle("CompressionLibrary")]

[assembly: AssemblyDescription("")]

[assembly: AssemblyConfiguration("")]

[assembly: AssemblyCompany("IntelliTect")]

[assembly: AssemblyProduct("Compression Library")]

[assembly: AssemblyCopyright("Copyright© IntelliTect 2006-2018")]

[assembly: AssemblyTrademark("")]

[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this

// assembly not visible to COM components. If you need to

// access a type in this assembly from COM, set the ComVisible

// attribute to true on that type.

[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib

// if this project is exposed to COM

[assembly: Guid("417a9609-24ae-4323-b1d6-cef0f87a42c3")]

// Version information for an assembly consists

// of the following four values:

//

// Major Version

// Minor Version

// Build Number

// Revision

//

// You can specify all the values or you can

// default the Revision and Build Numbers

// by using the '\*' as shown below:

// [assembly: AssemblyVersion("1.0.\*")]

[assembly: AssemblyVersion("1.0.0.0")]

[assembly: AssemblyFileVersion("1.0.0.0")]

The assembly attributes define things such as the company, product, and assembly version number. Similar to assembly, identifying an attribute usage as module requires prefixing it with module:. The restriction on assembly and module attributes is that they must appear after the using directive but before any namespace or class declarations. The attributes in Listing 18.10 are generated by the Visual Studio Project Wizard and should be included in all projects to mark the resultant binaries with information about the contents of the executable or dynamic link library (DLL).

Return attributes, such as the one shown in Listing 18.11, appear before a method declaration but use the same type of syntax structure.

Listing 18.11: Specifying a Return Attribute

[return: Description(

"Returns true if the object is in a valid state.")]

public bool IsValid()

{

// ...

return true;

}

In addition to assembly: and return:, C# allows for explicit target identifications of module:, class:, and method:, corresponding to attributes that decorate the module, class, and method, respectively. class: and method:, however, are optional, as demonstrated earlier.

One of the conveniences of using attributes is that the language takes into consideration the attribute naming convention, which calls for Attribute to appear at the end of the name. However, in all the attribute uses in the preceding listings, no such suffix appears, even though each attribute used follows the naming convention. This is because although the full name (DescriptionAttribute, AssemblyVersionAttribute, and so on) is allowed when applying an attribute, C# makes the suffix optional. Generally, no such suffix appears when applying an attribute; rather, it appears only when defining one or using the attribute inline (such as typeof(DescriptionAttribute)).

Note that instead of generating an AssemblyInfo.cs file, .NET Core based projects allow specification for the assembly information in the \*.CSPROJ file. Listing 18.12, for example, injects corresponding assembly attributes into the assembly at compile time:

Listing 18.12: Defining a Custom Attribute

<Project>

<PropertyGroup>

<Company>Addison Wesley</Company>

<Copyright>Copyright © Addison Wesley 2020</Copyright>

<Product>Essential C# 8.0</Product>

<Version>8.0</Version>

</PropertyGroup>

</Project>

These in turn, get converted into generated IL as shown in Output 18.5.

[assembly: AssemblyCompany("Addison Wesley")]  
[assembly: AssemblyCopyright("Copyright © Addison Wesley 2020")]  
[assembly: AssemblyFileVersion("8.0.0.0")]  
[assembly: AssemblyInformationalVersion("8.0")]  
[assembly: AssemblyProduct("Essential C# 8.0")]  
[assembly: AssemblyVersion("8.0.0.0")]

|  |
| --- |
| Output 18.5 |
| [assembly: AssemblyCompany("Addison Wesley")]  [assembly: AssemblyCopyright("Copyright © Addison Wesley 2020")]  [assembly: AssemblyFileVersion("8.0.0.0")]  [assembly: AssemblyInformationalVersion("8.0")]  [assembly: AssemblyProduct("Essential C# 8.0")]  [assembly: AssemblyVersion("8.0.0.0")] |

|  |
| --- |
| Guidelines  DO apply AssemblyVersionAttribute to assemblies with public types.  CONSIDER applying the AssemblyFileVersionAttribute and AssemblyCopyrightAttribute to provide additional information about the assembly.  DO apply the following information assembly attributes:  System.Reflection.AssemblyCompanyAttribute, System.Reflection.AssemblyCopyrightAttribute, System.Reflection.AssemblyDescriptionAttribute, and System.Reflection.AssemblyProductAttribute |

Custom Attributes

Defining a custom attribute is relatively trivial. Attributes are objects; therefore, to define an attribute, you need to define a class. The characteristic that turns a general class into an attribute is that it derives from System.Attribute. Consequently, you can create a CommandLineSwitchRequiredAttribute class, as shown in Listing 18.13.

Listing 18.13: Defining a Custom Attribute

public class CommandLineSwitchRequiredAttribute : Attribute

{

}

With that simple definition, you now can use the attribute as demonstrated in Listing 18.8. So far, no code responds to the attribute; therefore, the Out property that includes the attribute will have no effect on command-line parsing.

|  |
| --- |
| Guidelines  **DO** name custom attribute classes with the suffix Attribute. |

Looking for Attributes

In addition to providing properties for reflecting on a type’s members, Type includes methods to retrieve the Attributes decorating that type. Similarly, all the reflection types (e.g., PropertyInfo and MethodInfo) include members for retrieving a list of attributes that decorate a type. Listing 18.14 defines a method to return a list of required switches that are missing from the command line.

Listing 18.14: Retrieving a Custom Attribute

using System;

using System.Collections.Specialized;

using System.Reflection;

public class CommandLineSwitchRequiredAttribute : Attribute

{

public static string[] GetMissingRequiredOptions(

object commandLine)

{

List<string> missingOptions = new List<string>();

PropertyInfo[] properties =

commandLine.GetType().GetProperties();

foreach (PropertyInfo property in properties)

{

Attribute[] attributes =

(Attribute[])property.GetCustomAttributes(

typeof(CommandLineSwitchRequiredAttribute),

false);

if (attributes.Length > 0 &&

property.GetValue(commandLine, null) == null)

{

missingOptions.Add(property.Name);

}

}

return missingOptions.ToArray();

}

}

The code that checks for an attribute is relatively simple. Given a PropertyInfo object (obtained via reflection), you call GetCustomAttributes() and specify the attribute sought, then indicate whether to check any overloaded methods. (Alternatively, you can call the GetCustomAttributes() method without the attribute type to return all of the attributes.)

Although it is possible to place code for finding the CommandLineSwitchRequiredAttribute attribute within the CommandLineHandler’s code directly, it makes for better object encapsulation to place the code within the CommandLineSwitchRequiredAttribute class itself. This is frequently the pattern for custom attributes. What better location to place code for finding an attribute than in a static method on the attribute class?

Initializing an Attribute through a Constructor

The call to GetCustomAttributes() returns an array of objects that can be cast to an Attribute array. Because the attribute in our example didn’t have any instance members, the only metadata information that it provided in the returned attribute was whether it appeared. Attributes can also encapsulate data, however. Listing 18.15 defines a CommandLineAliasAttribute attribute—a custom attribute that provides alias command-line options. For example, you can provide command-line support for /Help or /? as an abbreviation. Similarly, /S could provide an alias to /Subfolders that indicates the command should traverse all the subdirectories.

To support this functionality, you need to provide a constructor for the attribute. Specifically, for the alias, you need a constructor that takes a string argument. (Similarly, if you want to allow multiple aliases, you need to define an attribute that has a params string array for a parameter.)

Listing 18.15: Providing an Attribute Constructor

public class CommandLineSwitchAliasAttribute : Attribute

{

public CommandLineSwitchAliasAttribute(string alias)

{

Alias = alias;

}

public string Alias { get; }

}

class CommandLineInfo

{

[CommandLineSwitchAlias("?")]

public bool Help { get; set; }

// ...

}

When applying an attribute to a construct, only constant values and typeof() expressions are allowed as arguments. This constraint is required to enable their serialization into the resultant CIL. It implies that an attribute constructor should require parameters of the appropriate types; creating a constructor that takes arguments of type System.DateTime would be of little value, as there are no System.DateTime constants in C#.

The objects returned from PropertyInfo.GetCustomAttributes() will be initialized with the specified constructor arguments, as demonstrated in Listing 18.16.

Listing 18.16: Retrieving a Specific Attribute and Checking Its Initialization

PropertyInfo property =

typeof(CommandLineInfo).GetProperty("Help");

CommandLineSwitchAliasAttribute attribute =

(CommandLineSwitchAliasAttribute)

property.GetCustomAttribute(

typeof(CommandLineSwitchAliasAttribute), false);

if(attribute?.Alias == "?")

{

Console.WriteLine("Help(?)");

};

Furthermore, as Listings 18.17 and 18.18 demonstrate, you can use similar code in a GetSwitches() method on CommandLineAliasAttribute that returns a dictionary collection of all the switches, including those from the property names, and associate each name with the corresponding attribute on the command-line object.

Listing 18.17: Retrieving Custom Attribute Instances

using System;

using System.Reflection;

using System.Collections.Generic;

public class CommandLineSwitchAliasAttribute : Attribute

{

public CommandLineSwitchAliasAttribute(string alias)

{

Alias = alias;

}

public string Alias { get; set; }

public static Dictionary<string, PropertyInfo> GetSwitches(

object commandLine)

{

PropertyInfo[] properties;

Dictionary<string, PropertyInfo> options =

new Dictionary<string, PropertyInfo>();

properties = commandLine.GetType().GetProperties(

BindingFlags.Public | BindingFlags.Instance);

foreach (PropertyInfo property in properties)

{

options.Add(property.Name, property);

foreach (CommandLineSwitchAliasAttribute attribute in

property.GetCustomAttributes(

typeof(CommandLineSwitchAliasAttribute), false))

{

options.Add(attribute.Alias.ToLower(), property);

}

}

return options;

}

}

Listing 18.18: Updating CommandLineHandler.TryParse() to Handle Aliases

using System;

using System.Reflection;

using System.Collections.Generic;

public class CommandLineHandler

{

// ...

public static bool TryParse(

string[] args, object commandLine,

out string? errorMessage)

{

bool success = false;

errorMessage = null;

Dictionary<string, PropertyInfo> options =

CommandLineSwitchAliasAttribute.GetSwitches(

commandLine);

foreach (string arg in args)

{

string option;

if (arg[0] == '/' || arg[0] == '-')

{

string[] optionParts = arg.Split(

new char[] { ':' }, 2);

option = optionParts[0].Remove(0, 1).ToLower();

if (options.TryGetValue(option, out PropertyInfo? property))

{

success = SetOption(

commandLine, property,

optionParts, ref errorMessage);

}

else

{

success = false;

errorMessage =

$"Option '{ option }' is not supported.";

}

}

}

return success;

}

private static bool SetOption(

object commandLine, PropertyInfo property,

string[] optionParts, ref string? errorMessage)

{

bool success;

if (property.PropertyType == typeof(bool))

{

// Last parameters for handling indexers

property.SetValue(

commandLine, true, null);

success = true;

}

else

{

if (optionParts.Length < 2

|| optionParts[1] == "")

{

// No setting was provided for the switch

success = false;

errorMessage =

$"You must specify the value for the { property.Name } option.";

}

else if (

property.PropertyType == typeof(string))

{

property.SetValue(

commandLine, optionParts[1], null);

success = true;

}

else if (

// property.PropertyType.IsEnum also available

property.PropertyType ==

typeof(ProcessPriorityClass))

{

success = TryParseEnumSwitch(

commandLine, optionParts,

property, ref errorMessage);

}

else

{

success = false;

errorMessage =

$@"Data type '{ property.PropertyType.ToString() }' on {

commandLine.GetType().ToString() } is not supported.";

}

}

return success;

}

}

|  |
| --- |
| Guidelines  **DO** provide get-only properties (without public setters) on attributes with required property values.  **DO** provide constructor parameters to initialize properties on attributes with required properties. Each parameter should have the same name (albeit with different casing) as the corresponding property.  AVOID providing constructor parameters to initialize attribute properties corresponding to the optional arguments (and therefore, avoid overloading custom attribute constructors). |

System.AttributeUsageAttribute

Most attributes are intended to decorate only particular constructs. For example, it makes no sense to allow CommandLineOptionAttribute to decorate a class or an assembly. The attribute in those contexts would be meaningless. To avoid inappropriate use of an attribute, custom attributes can be decorated with System.AttributeUsageAttribute (yes, an attribute is decorating a custom attribute declaration). Listing 18.19 (for CommandLineOptionAttribute) demonstrates how to do this.

Listing 18.19: Restricting the Constructs an Attribute Can Decorate

[AttributeUsage(AttributeTargets.Property)]

public class CommandLineSwitchAliasAttribute : Attribute

{

// ...

}

If the attribute is used inappropriately, as it is in Listing 18.20, it will cause a compile-time error, as Output 18.6 demonstrates.

Listing 18.20: AttributeUsageAttribute Restricting Where to Apply an Attribute

// ERROR: The attribute usage is restricted to properties

[CommandLineSwitchAlias("?")]

class CommandLineInfo

{

}

...Program+CommandLineInfo.cs(24,17): error CS0592: Attribute

'CommandLineSwitchAlias' is not valid on this declaration type. It is

valid on 'property, indexer' declarations only.

|  |
| --- |
| Output 18.6 |
| ...Program+CommandLineInfo.cs(24,17): error CS0592: Attribute  'CommandLineSwitchAlias' is not valid on this declaration type. It is  valid on 'property, indexer' declarations only. |

AttributeUsageAttribute’s constructor takes an AttributeTargets flag. This enum provides a list of all possible targets that the runtime allows an attribute to decorate. For example, if you also allowed CommandLineSwitchAliasAttribute on a field, you would update the AttributeUsageAttribute class, as shown in Listing 18.21.

Listing 18.21: Limiting an Attribute’s Usage with AttributeUsageAttribute

// Restrict the attribute to properties and methods

[AttributeUsage(

AttributeTargets.Field | AttributeTargets.Property)]

public class CommandLineSwitchAliasAttribute : Attribute

{

// ...

}

|  |
| --- |
| Guidelines  DO apply the AttributeUsageAttribute class to custom attributes. |

Named Parameters

In addition to restricting what an attribute can decorate, AttributeUsageAttribute provides a mechanism for allowing duplicates of the same attribute on a single construct. The syntax appears in Listing 18.22.

Listing 18.22: Using a Named Parameter

[AttributeUsage(AttributeTargets.Property, AllowMultiple=true)]

public class CommandLineSwitchAliasAttribute : Attribute

{

// ...

}

This syntax is different from the constructor initialization syntax discussed earlier. The AllowMultiple parameter is a named parameter, similar to the named parameter syntax used for optional method parameters (added in C# 4.0). Named parameters provide a mechanism for setting specific public properties and fields within the attribute constructor call, even though the constructor includes no corresponding parameters. The named attributes are optional designations, but they provide a means of setting additional instance data on the attribute without providing a constructor parameter for the purpose. In this case, AttributeUsageAttribute includes a public member called AllowMultiple. Therefore, you can set this member using a named parameter assignment when you use the attribute. Assigning named parameters must occur as the last portion of a constructor, following any explicitly declared constructor parameters.

Named parameters allow for assigning attribute data without providing constructors for every conceivable combination of which attribute properties are specified and which are not. Given that many of an attribute’s properties may be optional, this is a useful construct in many cases.

|  |  |
| --- | --- |
|  | |
|  | Beginner Topic  FlagsAttribute  Chapter 9 introduced enums and included an Advanced Topic covering FlagsAttribute. This framework-defined attribute targets enums that represent flag type values. The Beginner Topic here also addresses FlagsAttribute, starting with the sample code shown in Listing 18.23.  Listing 18.23: Using FlagsAttribute  // FileAttributes defined in System.IO  [Flags] // Decorating an enum with FlagsAttribute  public enum FileAttributes  {  ReadOnly = 1<<0, // 000000000000001  Hidden = 1<<1, // 000000000000010  // ...  }  using System;  using System.Diagnostics;  using System.IO;  class Program  {  public static void Main()  {  // ...  string fileName = @"enumtest.txt";  FileInfo file = new FileInfo(fileName);  file.Attributes = FileAttributes.Hidden |  FileAttributes.ReadOnly;  Console.WriteLine("\"{0}\" outputs as \"{1}\"",  file.Attributes.ToString().Replace(",", " |"),  file.Attributes);  FileAttributes attributes =  (FileAttributes)Enum.Parse(typeof(FileAttributes),  file.Attributes.ToString());  Console.WriteLine(attributes);  // ...  }  }  Output 18.7 shows the results of Listing 18.23.   |  | | --- | | Output 18.7 | | "ReadOnly | Hidden" outputs as "ReadOnly, Hidden" |   The flag documents that the enumeration values can be combined. Furthermore, it changes the behavior of the ToString() and Parse() methods. For example, calling ToString() on an enumeration that is decorated with FlagsAttribute writes out the strings for each enumeration flag that is set. In Listing 18.23, file.Attributes.ToString() returns "ReadOnly, Hidden" rather than the 3 it would have returned without the FlagsAttribute flag. If two enumeration values are the same, the ToString() call would return the first one. As mentioned earlier, however, you should use caution when relying on this outcome because it is not localizable.  Parsing a value from a string to the enumeration also works, provided each enumeration value identifier is separated by a comma.  Note that FlagsAttribute does not automatically assign the unique flag values or check that flags have unique values. The values of each enumeration item still must be assigned explicitly. |
|  | |

Predefined Attributes

The AttributeUsageAttribute attribute has a special characteristic that you haven’t seen yet in the custom attributes you have created in this book. This attribute affects the behavior of the compiler, causing the compiler to sometimes report an error. Unlike the reflection code you wrote earlier for retrieving CommandLineRequiredAttribute and CommandLineSwitchAliasAttribute, AttributeUsageAttribute has no runtime code; instead, it has built-in compiler support.

AttributeUsageAttribute is a predefined attribute. Not only do such attributes provide additional metadata about the constructs they decorate, but the runtime and compiler also behave differently to facilitate these attributes’ functionality. Attributes such as AttributeUsageAttribute, FlagsAttribute, ObsoleteAttribute, and ConditionalAttribute are examples of predefined attributes. They implement special behavior that only the CLI provider or compiler can offer because there are no extension points for additional non-custom attributes. In contrast, custom attributes are entirely passive. Listing 18.23 includes a couple of predefined attributes; Chapter 19 includes a few more.

System.ConditionalAttribute

Within a single assembly, the System.Diagnostics.ConditionalAttribute attribute behaves a little like the #if/#endif preprocessor identifier. However, instead of eliminating the CIL code from the assembly, System.Diagnostics.ConditionalAttribute will optionally cause the call to behave like a no-op, an instruction that does nothing. Listing 18.24 demonstrates the concept, and Output 18.8 shows the results.

Listing 18.24: Using ConditionalAttribute to Eliminate a Call

#define CONDITION\_A

using System;

using System.Diagnostics;

public class Program

{

public static void Main()

{

Console.WriteLine("Begin...");

MethodA();

MethodB();

Console.WriteLine("End...");

}

[Conditional("CONDITION\_A")]

static void MethodA()

{

Console.WriteLine("MethodA() executing...");

}

[Conditional("CONDITION\_B")]

static void MethodB()

{

Console.WriteLine("MethodB() executing...");

}

}

|  |
| --- |
| Output 18.8 |
| Begin...  MethodA() executing...  End... |

This example defined CONDITION\_A, so MethodA() executed normally. CONDITION\_B, however, was not defined either through #define or by using the csc.exe /Define option. As a result, all calls to Program.MethodB() from within this assembly will do nothing.

Functionally, ConditionalAttribute is similar to placing an #if/#endif around the method invocation. The syntax is cleaner, however, because developers create the effect by adding the ConditionalAttribute attribute to the target method without making any changes to the caller itself.

The C# compiler notices the attribute on a called method during compilation, and assuming the preprocessor identifier exists, it eliminates any calls to the method. ConditionalAttibute, however, does not affect the compiled CIL code on the target method itself (besides the addition of the attribute metadata). Instead, it affects the call site during compilation by removing the calls. This further distinguishes ConditionalAttribute from #if/#endif when calling across assemblies. Because the decorated method is still compiled and included in the target assembly, the determination of whether to call a method is based not on the preprocessor identifier in the callee’s assembly but rather on the caller’s assembly. In other words, if you create a second assembly that defines CONDITION\_B, any calls to Program.MethodB() from the second assembly will execute. This is a useful characteristic in many tracing and testing scenarios. In fact, calls to System.Diagnostics.Trace and System.Diagnostics.Debug use this trait with ConditionalAttributes on TRACE and DEBUG preprocessor identifiers.

Because methods don’t execute whenever the preprocessor identifier is not defined, ConditionalAttribute may not be used on methods that include an out parameter or specify a return other than void. Doing so causes a compile-time error. This makes sense because potentially none of the code within the decorated method will execute, so it is unknown what to return to the caller. Similarly, properties cannot be decorated with ConditionalAttribute. The AttributeUsage (see the section titled “System.AttributeUsageAttribute” earlier in this chapter) for ConditionalAttribute[[2]](#footnote-2) is AttributeTargets.Class and AttributeTargets.Method, which allows the attribute to be used on either a method or a class. However, the class usage is special because ConditionalAttribute is allowed only on System.Attribute-derived classes.

When ConditionalAttribute decorates a custom attribute, the latter can be retrieved via reflection only if the conditional string is defined in the calling assembly. Without such a conditional string, reflection that looks for the custom attribute will fail to find it.

System.ObsoleteAttribute

As mentioned earlier, predefined attributes affect the compiler’s and/or the runtime’s behavior. ObsoleteAttribute provides another example of attributes affecting the compiler’s behavior. Its purpose is to help with the versioning of code, providing a means of indicating to callers that a member or type is no longer current. Listing 18.25 is an example of ObsoleteAttribute usage. As Output 18.9 shows, any callers that compile code that invokes a member marked with ObsoleteAttribute will cause a compile-time warning, optionally an error.

Listing 18.25: Using ObsoleteAttribute

class Program

{

public static void Main()

{

ObsoleteMethod();

}

[Obsolete]

public static void ObsoleteMethod()

{

}

}

|  |
| --- |
| Output 18.9 |
| c:\SampleCode\ObsoleteAttributeTest.cs(24,17): warning CS0612:  Program.ObsoleteMethod()' is obsolete |

In this case, ObsoleteAttribute simply displays a warning. However, there are two additional constructors on the attribute. One of them, ObsoleteAttribute(string message), appends the additional message argument to the compiler’s obsolete message. The best practice for this message is to provide direction on what replaces the obsolete code. The second constructor is a bool error parameter that forces the warning to be recorded as an error instead.

ObsoleteAttribute allows third parties to notify developers of deprecated APIs. The warning (not an error) allows the original API to continue to work until the developer is able to update the calling code.



\*\*\*COMP: Insert “Begin 4.0” tab

Programming with Dynamic Objects

The introduction of dynamic objects in C# 4.0 simplified a host of programming scenarios and enabled several new ones previously not available. At its core, programming with dynamic objects enables developers to code operations using a dynamic dispatch mechanism that the runtime will resolve at execution time rather than the compiler verifying and binding to it at compile time.

Why? Many times, objects are inherently not statically typed. Examples include loading data from an XML/CSV file, a database table, the Internet Explorer DOM, or COM’s IDispatch interface, or calling code in a dynamic language such as an IronPython object. C# 4.0’s Dynamic object support provides a common solution for talking to runtime environments that don’t necessarily have a compile-time–defined structure. In the initial implementation of dynamic objects in C# 4.0, four binding methods are available:

1. Using reflection against an underlying CLR type

2. Invoking a custom IDynamicMetaObjectProvider that makes available a DynamicMetaObject

3. Calling through the IUnknown and IDispatch interfaces of COM

4. Calling a type defined by dynamic languages such as IronPython

Of these four approaches, we will delve into the first two. The principles underlying them translate seamlessly to the remaining cases—COM interoperability and dynamic language interoperability.

Invoking Reflection Using dynamic

One of the key features of reflection is the ability to dynamically find and invoke a member on a type based on an execution-time identification of the member name or some other quality, such as an attribute (see Listing 18.3). However, C# 4.0’s addition of dynamic objects provides a simpler way of invoking a member by reflection, assuming compile-time knowledge of the member signature. To reiterate, this restriction states that at compile time we need to know the member name along with the signature (the number of parameters and whether the specified parameters will be type-compatible with the signature). Listing 18.26 (with Output 18.10) provides an example.

Listing 18.26: Dynamic Programming Using Reflection

using System;

// ...

dynamic data =

"Hello! My name is Inigo Montoya";

Console.WriteLine(data);

data = (double)data.Length;

data = data \* 3.5 + 28.6;

if(data == 2.4 + 112 + 26.2)[[3]](#footnote-3)

{

Console.WriteLine(

$"{ data } makes for a long triathlon.");

}

else

{

data.NonExistentMethodCallStillCompiles();

}

// ...

|  |
| --- |
| Output 18.10 |
| Hello! My name is Inigo Montoya  140.6 makes for a long triathlon. |

In this example, there is no explicit code for determining the object type, finding a particular MemberInfo instance, and then invoking it. Instead, data is declared as type dynamic and methods are called against it directly. At compile time, there is no check as to whether the members specified are available or even a check regarding which type underlies the dynamic object. Hence, it is possible at compile time to make any call so long as the syntax is valid. At compile time, it is irrelevant whether there is really a corresponding member.

However, type safety is not abandoned altogether. For standard CLR types (such as those used in Listing 18.26), the same type checker normally used at compile time for non-dynamic types is instead invoked at execution time for the dynamic type. Therefore, at execution time, if no such member is available, the call will result in a Microsoft.CSharp.RuntimeBinder.RuntimeBinderException.

Note that this capability is not nearly as flexible as the reflection described earlier in the chapter, although the API is undoubtedly simpler. The key difference when using a dynamic object is that it is necessary to identify the signature at compile time rather than determine things such as the member name at runtime (as we did when parsing the command-line arguments).

dynamic Principles and Behaviors

Listing 18.26 and the accompanying text reveal several characteristics of the dynamic data type:

* dynamic is a directive to the compiler to generate code.

dynamic involves an interception mechanism so that when a dynamic call is encountered by the runtime, it can compile the request to CIL and then invoke the newly compiled call. (See the Advanced Topic titled “dynamic Uncovered” later in this chapter for more details.)

The principle at work when a type is assigned to dynamic is to conceptually “wrap” the original type so that no compile-time validation occurs. Additionally, when a member is invoked at runtime, the wrapper intercepts the call and dispatches it appropriately (or rejects it). Calling GetType() on the dynamic object reveals the type underlying the dynamic instance—it does not return dynamic as a type.

Any type[[4]](#footnote-4) will convert to dynamic.

In Listing 18.26, we successfully cast both a value type (double) and a reference type (string) to dynamic. In fact, all types can successfully be converted into a dynamic object. There is an implicit conversion from any reference type to dynamic. Similarly, there is an implicit conversion (a boxing conversion) from a value type to dynamic. In addition, there is an implicit conversion from dynamic to dynamic. This is perhaps obvious, but with dynamic this process is more complicated than simply copying the “pointer” (address) from one location to the next.

Successful conversion from dynamic to an alternative type depends on support in the underlying type.

Conversion from a dynamic object to a standard CLR type is an explicit cast (e.g., (double)data.Length). Not surprisingly, if the target type is a value type, an unboxing conversion is required. If the underlying type supports the conversion to the target type, the conversion from dynamic will also succeed.

The type underlying the dynamic type can change from one assignment to the next.

Unlike an implicitly typed variable (var), which cannot be reassigned to a different type, dynamic involves an interception mechanism for compilation before the underlying type’s code is executed. Therefore, it is possible to successfully swap out the underlying type instance to an entirely different type. This will result in another interception call site that will need to be compiled before invocation.

Verification that the specified signature exists on the underlying type doesn’t occur until runtime—but it does occur.

The compiler makes almost no verification of operations on a dynamic type, as the method call to data.NonExistentMethodCallStillCompiles() demonstrates. This step is left entirely to the work of the runtime when the code executes. Moreover, if the code never executes, even though surrounding code does (as with data.NonExistentMethodCallStillCompiles()), no verification and binding to the member will ever occur.

The result of any dynamic member invocation is of compile-time type dynamic.

A call to any member on a dynamic object will return a dynamic object. Therefore, calls such as data.ToString() will return a dynamic object rather than the underlying string type. However, at execution time, when GetType() is called on the dynamic object, an object representing the runtime type is returned.

If the member specified does not exist at runtime, the runtime will throw a Microsoft.CSharp.RuntimeBinder.RuntimeBinderException exception.

If an attempt to invoke a member at execution time does occur, the runtime will verify that the member call is truly valid (e.g., that the signatures are type-compatible in the case of reflection). If the method signatures are not compatible, the runtime will throw a Microsoft.CSharp.RuntimeBinder.RuntimeBinderException.

dynamic with reflection does not support extension methods.

Just like with reflection using System.Type, reflection using dynamic does not support extension methods. Invocation of extension methods is still available on the implementing type (e.g., System.Linq.Enumerable), just not on the dynamic type directly.

At its core, dynamic is a System.Object.

Given that any object can be successfully converted to dynamic, and that dynamic may be explicitly converted to a different object type, dynamic behaves like System.Object. Like System.Object, it even returns null for its default value (default(dynamic)), indicating it is a reference type. The special dynamic behavior of dynamic that distinguishes it from a System.Object appears only at compile time.

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  dynamic Uncovered  The CIL disassembler reveals that within the CIL, the dynamic type is actually a System.Object. In fact, without any invocations, declaration of the dynamic type is indistinguishable from System.Object. However, the difference becomes apparent when invoking a member. To invoke the member, the compiler declares a variable of type System.Runtime.CompilerServices.CallSite<T>. T varies on the basis of the member signature, but something simple such as the invocation of ToString() would require instantiation of the type CallSite<Func<CallSite, object, string>>, along with a method call with parameters of CallSite site, object dynamicTarget, and string result. site is the call site itself, dynamicTarget is the object on which the method call is invoked, and result is the underlying return value from the ToString() method call. Rather than instantiate CallSite<Func<CallSite \_site, object dynamicTarget, string result>> directly, a Create() factory method is available for instantiating it. (Create() takes a parameter of type Microsoft.CSharp.RuntimeBinder.CSharpConvertBinder.) Given an instance of the CallSite<T>, the final step involves a call to CallSite<T>.Target() to invoke the actual member.  Under the covers at execution time, the framework uses reflection to look up members and to verify that the signatures match. Next, the runtime builds an expression tree that represents the dynamic expression as defined by the call site. Once the expression tree is compiled, we have a CIL method body that is similar to what the compiler would have generated had the call not been dynamic. This CIL code is then cached in the call site, and the invocation occurs using a delegate invoke. As the CIL is now cached at the call site, the next invocation doesn’t require all the reflection and compilation overhead again. |
|  | |

Why Dynamic Binding?

In addition to reflection, we can define custom types that we invoke dynamically. We might consider using dynamic invocation to retrieve the values of an XML element, for example. Rather than using the strongly typed syntax of Listing 18.27, using dynamic invocation we could call person.FirstName and person.LastName.

Listing 18.27: Runtime Binding to XML Elements without dynamic

using System;

using System.Xml.Linq;

// ...

XElement person = XElement.Parse(

@"<Person>

<FirstName>Inigo</FirstName>

<LastName>Montoya</LastName>

</Person>");

Console.WriteLine("{0} {1}",

person.Descendants("FirstName").FirstOrDefault().Value,

person.Descendants("LastName").FirstOrDefault().Value);

// ...

Although the code in Listing 18.27 is not overly complex, compare it to Listing 18.28—an alternative approach that uses a dynamically typed object.

Listing 18.28: Runtime Binding to XML Elements with dynamic

using System;

// ...

// See Listing 13.32 for DynamicXml listing.

dynamic person = DynamicXml.Parse(

@"<Person>

<FirstName>Inigo</FirstName>

<LastName>Montoya</LastName>

</Person>");

Console.WriteLine(

$"{ person.FirstName } { person.LastName }");

// ...

The advantages are clear, but does that mean dynamic programming is preferable to static compilation?

Static Compilation versus Dynamic Programming

In Listing 18.28, we have the same functionality as in Listing 18.27, albeit with one very important difference: Listing 18.27 is entirely statically typed. Thus, at compile time, all types and their member signatures are verified with this approach. Method names are required to match, and all parameters are checked for type compatibility. This is a key feature of C# and something we have highlighted throughout the book.

In contrast, Listing 18.28 has virtually no statically typed code; the variable person is instead dynamic. As a result, there is no compile-time verification that person has a FirstName or LastName property—or any other members, for that matter. Furthermore, when coding within an IDE, there is no IntelliSense identifying any members on person.

The loss of typing would seem to result in a significant decrease in functionality. Why, then, is such a possibility even available in C#—a functionality that was added in C# 4.0, in fact?

To understand this apparent paradox, let’s reexamine Listing 18.28. Notice the call to retrieve the "FirstName" element:

Element.Descendants("LastName").FirstOrDefault().Value

The listing uses a string ("LastName") to identify the element name, but there is no compile-time verification that the string is correct. If the casing was inconsistent with the element name or if there was a space, the compile would still succeed, even though a NullReferenceException would occur with the call to the Value property. Furthermore, the compiler does not attempt to verify that the "FirstName" element even exists; if it doesn’t, we would also get the NullReferenceException message. In other words, in spite of all the type-safety advantages, type safety doesn’t offer many benefits when you’re accessing the dynamic data stored within the XML element.

Listing 18.28 is no better than Listing 18.27 when it comes to compile-time verification of the element retrieval. If a case mismatch occurs or if the FirstName element didn’t exist, there would still be an exception.[[5]](#footnote-5) However, compare the call to access the first name in Listing 18.28 (person.FirstName) with the call in Listing 18.27. The call in the latter listing is undoubtedly significantly simpler.

In summary, there are situations in which type safety doesn’t—and likely can’t—make certain checks. In such cases, code that makes a dynamic call that is verified only at runtime, rather than also being verified at compile time, is significantly more readable and succinct. Obviously, if compile-time verification is possible, statically typed programming is preferred because readable and succinct APIs can accompany it. However, in the cases where it isn’t effective, C# 4.0’s dynamic capabilities enables programmers to write simpler code rather than emphasizing the purity of type safety.

Implementing a Custom Dynamic Object

Listing 18.28 included a method call to DynamicXml.Parse(...) that was essentially a factory method call for DynamicXml—a custom type rather than one built into the CLR framework. However, DynamicXml doesn’t implement a FirstName or LastName property. To do so would break the dynamic support for retrieving data from the XML file at execution time rather than fostering compile-time-based implementation of the XML elements. In other words, DynamicXml does not use reflection for accessing its members but rather dynamically binds to the values based on the XML content.

The key to defining a custom dynamic type is implementation of the System.Dynamic.IDynamicMetaObjectProvider interface. Rather than implementing the interface from scratch, however, the preferred approach is to derive the custom dynamic type from System.Dynamic.DynamicObject. This provides default implementations for a host of members and allows you to override the ones that don’t fit. Listing 18.29 shows the full implementation.

Listing 18.29: Implementing a Custom Dynamic Object

using System;

using System.Dynamic;

using System.Xml.Linq;

public class DynamicXml : DynamicObject

{

private XElement Element { get; set; }

public DynamicXml(System.Xml.Linq.XElement element)

{

Element = element;

}

public static DynamicXml Parse(string text)

{

return new DynamicXml(XElement.Parse(text));

}

public override bool TryGetMember(

GetMemberBinder binder, out object? result)

{

bool success = false;

result = null;

XElement firstDescendant =

Element.Descendants(binder.Name).FirstOrDefault();

if (firstDescendant != null)

{

if (firstDescendant.Descendants().Any())

{

result = new DynamicXml(firstDescendant);

}

else

{

result = firstDescendant.Value;

}

success = true;

}

return success;

}

public override bool TrySetMember(

SetMemberBinder binder, object value)

{

bool success = false;

XElement firstDescendant =

Element.Descendants(binder.Name).FirstOrDefault();

if (firstDescendant != null)

{

if (value.GetType() == typeof(XElement))

{

firstDescendant.ReplaceWith(value);

}

else

{

firstDescendant.Value = value.ToString();

}

success = true;

}

return success;

}

}

The key dynamic implementation methods for this use case are TryGetMember() and TrySetMember() (assuming you want to assign the elements as well). Only these two method implementations are necessary to support the invocation of the dynamic getter and setter properties. Furthermore, the implementations are straightforward. First, they examine the contained XElement, looking for an element with the same name as the binder.Name—the name of the member invoked. If a corresponding XML element exists, the value is retrieved (or set). The return value is set to true if the element exists and false if it doesn’t. A return value of false will immediately cause the runtime to throw a Microsoft.CSharp.RuntimeBinder.RuntimeBinderException at the call site of the dynamic member invocation.

System.Dynamic.DynamicObject supports additional virtual methods if more dynamic invocations are required. Listing 18.30 produces a list of all overridable members.

Listing 18.30: Overridable Members on System.Dynamic.DynamicObject

using System.Dynamic;

public class DynamicObject : IDynamicMetaObjectProvider

{

protected DynamicObject();

public virtual IEnumerable<string> GetDynamicMemberNames();

public virtual DynamicMetaObject GetMetaObject(

Expression parameter);

public virtual bool TryBinaryOperation(

BinaryOperationBinder binder, object arg,

out object result);

public virtual bool TryConvert(

ConvertBinder binder, out object result);

public virtual bool TryCreateInstance(

CreateInstanceBinder binder, object[] args,

out object result);

public virtual bool TryDeleteIndex(

DeleteIndexBinder binder, object[] indexes);

public virtual bool TryDeleteMember(

DeleteMemberBinder binder);

public virtual bool TryGetIndex(

GetIndexBinder binder, object[] indexes,

out object result);

public virtual bool TryGetMember(

GetMemberBinder binder, out object result);

public virtual bool TryInvoke(

InvokeBinder binder, object[] args, out object result);

public virtual bool TryInvokeMember(

InvokeMemberBinder binder, object[] args,

out object result);

public virtual bool TrySetIndex(

SetIndexBinder binder, object[] indexes, object value);

public virtual bool TrySetMember(

SetMemberBinder binder, object value);

public virtual bool TryUnaryOperation(

UnaryOperationBinder binder, out object result);

}

As Listing 18.30 shows, there are member implementations for everything—from casts and various operations, to index invocations. In addition, there is a method for retrieving all the possible member names: GetDynamicMemberNames().

\*\*\*COMP: Insert “End 4.0” tab

Summary

This chapter discussed how to use reflection to read the metadata that is compiled into the CIL. Using reflection, it is possible to provide a late binding in which the code to call is defined at execution time rather than at compile time. Although reflection is entirely feasible for deploying a dynamic system, it executes considerably more slowly than statically linked (compile-time), defined code. This tends to make it more prevalent and useful in development tools when performance is potentially not as critical.

Reflection also enables the retrieval of additional metadata decorating various constructs in the form of attributes. Typically, custom attributes are sought using reflection. You can define your own custom attributes that insert additional metadata of your own choosing into the CIL. At runtime, you can then retrieve this metadata and use it within the programming logic.

Many programmers view attributes as a precursor to a concept known as aspect-oriented programming, in which you add functionality through constructs such as attributes instead of manually implementing the functionality wherever it is needed. It will take some time before you see true aspects within C# (if ever); however, attributes provide a clear steppingstone in that direction, without creating a significant risk to the stability of the language.

Finally, this chapter included a feature introduced in C# 4.0—dynamic programming using the new type dynamic. This coverage included a discussion of why static binding, although preferred when the API is strongly typed, has limitations when working with dynamic data.

The next chapter looks at multithreading, where attributes are used for synchronization.

[Reflection 1](#_Toc37532932)

[Accessing Metadata Using System.Type 2](#_Toc37532933)

[GetType() 3](#_Toc37532934)

[typeof() 4](#_Toc37532935)

[Member Invocation 5](#_Toc37532936)

[Reflection on Generic Types 11](#_Toc37532937)

[Determining the Type of Type Parameters 11](#_Toc37532938)

[Determining Whether a Class or Method Supports Generics 11](#_Toc37532939)

[Obtaining Type Parameters for a Generic Class or Method 12](#_Toc37532940)

[nameof Operator 13](#_Toc37532941)

[Attributes 15](#_Toc37532942)

[Custom Attributes 19](#_Toc37532943)

[Looking for Attributes 20](#_Toc37532944)

[Initializing an Attribute through a Constructor 21](#_Toc37532945)

[System.AttributeUsageAttribute 25](#_Toc37532946)

[Named Parameters 27](#_Toc37532947)

[Predefined Attributes 29](#_Toc37532948)

[System.ConditionalAttribute 30](#_Toc37532949)

[System.ObsoleteAttribute 32](#_Toc37532950)

[Programming with Dynamic Objects 33](#_Toc37532951)

[Invoking Reflection Using dynamic 33](#_Toc37532952)

[dynamic Principles and Behaviors 35](#_Toc37532953)

[Why Dynamic Binding? 38](#_Toc37532954)

[Static Compilation versus Dynamic Programming 39](#_Toc37532955)

[Implementing a Custom Dynamic Object 40](#_Toc37532956)

1. . The .NET Standard 1.6 added the CommandLineUtils NuGet package that also provides a command-line parsing mechanism. For more information, see my MSDN article on the topic at http://itl.tc/sept2016. [↑](#footnote-ref-1)
2. . A feature started in Microsoft .NET Framework 2.0. [↑](#footnote-ref-2)
3. The distances (in miles) for the swim, bike, and run portions of an Ironman, respectively. [↑](#footnote-ref-3)
4. Technically, it is restricted to any type that converts to object - which excludes unsafe pointers, lambdas and method groups. [↑](#footnote-ref-4)
5. . You cannot use a space in the FirstName property call, but neither does XML support spaces in element names, so let’s ignore this fact. [↑](#footnote-ref-5)