

Microsoft

2005
EDITION



PROGRAMMING **MICROSOFT**
VISUAL C# 2005:
THE LANGUAGE

Donis Marshall

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2006 by Donis Marshall

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number 2005934153

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 0 9 8 7 6 5

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft, IntelliSense, Microsoft Press, MSDN, Visual Basic, Visual C#, Visual Studio, the Visual Studio logo, Win32, Windows, Windows CE, the Windows logo, Windows NT, and WinFX are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Ben Ryan
Project Editor: Valerie Woolley
Technical Editor: James D. Rogers
Copy Editor: Nancy Sixsmith
Indexer: Tony Ross and Lee Ross

Body Part No. X11-50074

Table of Contents

Acknowledgments.....	xix
Introduction.....	xxi

Part I Core Language

1	Introduction to Visual C# Programming	3
	Language Origin.....	4
	C# Core Language Features.....	7
	Symbols and Tokens.....	7
	Keywords.....	25
	Primitives.....	29
	Sample C# Program.....	30
	Namespaces.....	31
	Main Entry Point.....	35
	Local Variables.....	36
	Nullable Types.....	37
	Expressions.....	38
	Selection Statements.....	38
	Iterative Statements.....	41
	Classes.....	44
2	Types	45
	Classes.....	46
	Class Members.....	48
	Member Functions.....	54
	Structures.....	76
	Enumeration.....	77
	Bitwise Enumeration.....	79
	Identity versus Equivalence.....	80
	Class Refinement.....	81

What do you think of this book?
We want to hear from you!

Microsoft is interested in hearing your feedback about this publication so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: www.microsoft.com/learning/booksurvey/

3	Inheritance	83
	Inheritance Example	88
	System.Object	90
	Object.Equals Method	92
	Object.GetHashCode Method	93
	Object.GetType Method	93
	Object.ToString Method	94
	Object.MemberwiseClone Method	94
	Object.ReferenceEquals Method	95
	Employee Class	96
	Implementing Inheritance	98
	Access Modifiers	98
	Overriding	99
	Virtual and Override Keywords	99
	Overload versus Override	100
	Overriding Events	101
	New Modifier	102
	Abstract	106
	Sealed	107
	Constructors and Destructors	108
	Interfaces	111
	Implementing Interfaces	113
	Explicit Interface Member Implementation	115
	Reimplementation of Interfaces	119
	Polymorphism	119
	Interface Polymorphism	123
	New Modifier and Polymorphism	124
	Casting	125
	Inheritance Operators	128
	Attribute Inheritance	130
	Visual Studio .NET 2005	131

Part II Core Skills

4	Introduction to Visual Studio 2005	135
	Integrated Development Environment.....	136
	Start Page.....	136
	Community Integration.....	137
	Creating Projects.....	137
	Solution Explorer.....	138
	Adding References.....	141
	Data Menu.....	141
	Managing Windows in Visual Studio.....	142
	AutoRecover.....	143
	Class Hierarchies.....	139
	Class View Window.....	143
	Object Browser.....	144
	Class Diagram.....	145
	Inheritance.....	149
	Association.....	147
	Class Diagram Walkthrough.....	151
	Error List Window.....	156
	Code Editor.....	156
	IntelliSense.....	156
	Surround With.....	158
	Font and Color Formatting.....	158
	Source Code Formatting.....	159
	Change Tracking.....	159
	Profile.....	160
	Code Snippets.....	160
	Copy and Paste.....	161
	Insert a Code Snippet.....	162
	Default Snippets.....	163
	Code Snippets Manager.....	165
	Creating Snippets.....	166

	Refactoring	171
	Refactoring Walkthrough	173
	Building and Deployment	176
	MSBuild	176
	Items	177
	Properties	177
	Tasks	178
	Project File	179
	MSBuild Walkthrough	180
	Click Once Deployment	182
	Publish a ClickOnce Application	185
	Arrays and Collections	186
5	Arrays and Collections.	187
	Arrays	189
	Array Elements	191
	Multidimensional Arrays	191
	Jagged Arrays	194
	System.Array	196
	System.Array Properties	203
	params Keyword	211
	Array Conversion	213
	Collections	214
	ArrayList Collection	215
	BitArray Collection	219
	Hashtable Collection	222
	Queue Collection	226
	Stack Collection	231
	Specialized Collections	232
	Generics	233
6	Generics.	235
	Generic Types	238
	Type Parameters	239
	Constructed Types	243
	Generic Methods	243
	Overloaded Methods	244
	This Reference for Generic Types	246

Constraints	246
Derivation Constraint	248
Interface Constraints	252
Value Type Constraint	254
Reference Type Constraint	254
Default Constructor Constraint	255
Casting	256
Inheritance	257
Overriding Methods	258
Nested Types	259
Static Members	260
Operator Functions	261
Serialization	263
Generics Internals	265
Generic Collections	266
Enumerators	267
7 Iterators	269
Enumerable Objects	270
Generic Enumerators	278
Iterators	283
Delegates and Events	291
Part III More C# Language	
8 Delegates and Events	295
Delegates	296
Define a Delegate	297
Create a Delegate	298
Invoking a Delegate	300
Arrays of Delegates	300
Asynchronous Invocation	307
Asynchronous Delegate Diagram	311
Exceptions	312
Anonymous Methods	313
Outer Variables	316
Generic Anonymous Methods	318
Limitations of Anonymous Methods	318

- Events. 319
 - Publishing an Event. 319
 - Subscribe. 320
 - Raising an Event. 321
- Exception Handling 323

9 Exception Handling 325

- Exception Example. 326
- Common Exception Model. 327
- Structured Exception Handling 327
 - Try Statement 327
 - Catch Statement. 329
 - Finally Statement 332
 - Exception Information Table 333
 - Nested Try Blocks. 333
- System.Exception 335
 - System.Exception Functions 336
 - System.Exception Properties 337
 - Application Exceptions 338
 - Exception Translation 340
 - COM Interoperability Exceptions 341
- Remote Exceptions 345
- Unhandled Exceptions 347
 - Application.ThreadException. 348
 - AppDomain.UnhandledException. 349
- Managing Exceptions in Visual Studio 351
 - Exception Assistant 351
 - Exceptions Dialog Box. 351
- Metadata and Reflections. 352

Part IV Debugging

10 Metadata and Reflection 355

- Metadata. 355
 - Tokens 357
 - Metadata Heaps. 358
 - Streams 358
 - Metadata Validation 359
 - ILDASM Tool. 360

Reflection	364
Obtaining a Type Object	365
Loading Assemblies	367
Browsing Type Information	370
Dynamic Invocation	373
Type Creation	378
Late Binding Delegates	380
Function Call Performance	383
Reflection and Generics	383
IsGeneric and IsGenericTypeDefinition	383
typeof	384
GetType	384
GetGenericTypeDefinition	385
GetGenericArguments	386
Creating Generic Types	387
Reflection Security	388
Attributes	389
Creating a Custom Attribute	392
Attributes and Reflection	396
MSIL	399
11 MSIL Programming	401
"Hello World" Application	403
Evaluation Stack	405
MSIL in Depth	406
Complex Tasks	417
Branching	423
Arrays	428
Arithmetic Instructions	429
Process Execution	433
Debugging with Visual Studio 2005	436
12 Debugging with Visual Studio 2005	437
Debugging Overview	438
Debugging Windows Forms Projects	438
Debugging Setup	441
Debug Settings	443
Visual Studio Debugging User Interface	450

	Breakpoints	453
	Code Stepping	461
	Debug Toolbar	464
	Debug Windows	464
	Tracing	476
	DebuggerDisplayAttribute	492
	Dump Files	496
	Advanced Debugging	498
13	Advanced Debugging	499
	DebuggableAttribute Attribute	501
	Debuggers	503
	Just-In-Time (JIT) Debugging	504
	Managed Debugger	506
	MDBG Commands	511
	WinDbg	512
	WinDbg Basic Commands	513
	Son of Strike (SOS)	519
	SOS Walkthrough Part I	520
	SOS Walkthrough Part II	523
	Dumps	525
	ADPlus	525
	Dr. Watson	527
	Memory Management	530
	Reference Tree	531
	Generations	534
	Finalization	537
	Performance Monitor	538
	Threads	539
	Threads Commands	541
	Exceptions	546
	Symbols	547
	Symsrv Symbol Server	548
	Application Symbols	549
	Memory Management	550

Part V Advanced Concepts

14	Memory Management	553
	Unmanaged Resources	554
	Garbage Collection Overview.....	555
	GC Flavors	559
	Workstation GC Without Concurrent Garbage Collection.....	560
	Server GC.....	560
	Finalizers	562
	IDisposable.Dispose	577
	Disposable Pattern	581
	Disposable Pattern Considerations	582
	Disposing Inner Objects	586
	Weak Reference	588
	Weak Reference Internals	591
	Weak Reference Class	591
	Critical Finalization Objects	592
	Constrained Execution Region	592
	Managing Unmanaged Resources.....	595
	GC Class	598
	Nonsecure Code	598
15	Unsafe Code	601
	Unsafe Keyword	603
	Pointers	604
	Pointer Parameters and Return	608
	Platform Invoke	611
	Summary	625
	Appendix: Operator Overloading	627
	Mathematical and Logical Operators	628
	Implementation	629
	Increment and Decrement Operators.....	632
	LeftShift and RightShift Operators	633
	Operator True and Operator False.....	634
	Paired Operators.....	635

Conversion Operators	639
Operator String	641
Practical Example.....	642
Operator Overloading Internals.....	645
Index.....	649

What do you think of this book?
We want to hear from you!

Microsoft is interested in hearing your feedback about this publication so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: www.microsoft.com/learning/booksurvey/

Metadata and Reflection

In this chapter:

Metadata	355
Reflection	364
Reflection and Generics	383
MSIL	399

Metadata, which is often described as data about data, formalizes the composition of an assembly. Metadata is the actualization of the state of an assembly. As a reflection of an assembly, metadata includes all information pertaining to the assembly, including a detailed description of each type, the attributes of the assembly, and other particulars of the assembly itself. Metadata is similar to a type library in COM, except that metadata is always persisted in the application that it describes. For this reason, assemblies are often referred to as self-describing. Because metadata is indigenous to the assembly, metadata cannot be lost, and versioning problems are avoided. Metadata is emitted primarily by managed language compilers and consumed by metadata browsers, other .NET tools, and managed applications. The Common Language Runtime (CLR) uses metadata extensively. Just-in-time compilation, code access security, garbage collection, and other services of the CLR rely heavily on metadata. Once emitted, metadata is read-only.

Metadata is important to anyone programming in the managed environment. Assembly inspection, late binding, and advanced concepts such as self-generating code require a non-trivial understanding of metadata. You can interrogate metadata by using reflection. *Reflection* facilitates late binding and other means of leveraging metadata. Most important, mastery of metadata promotes a better understanding of the managed world, which (one hopes) translates into better-written code.

Metadata

Metadata about the overall assembly and modules is called the *manifest*. Some of the macro information assembled in the manifest includes the simple name, version number, external references, module name, and public key of the assembly. A portion of the manifest is created from the assembly attributes found in the AssemblyInfo.cs file of a Microsoft Visual Studio .NET C# project. This is a partial listing of a typical AssemblyInfo.cs file:

```
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("WindowsApplication4")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("WindowsApplication4")]
[assembly: AssemblyCopyright("Copyright © 2005")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

Metadata also chronicles the microdata of the assembly, such as types, methods, and attributes. Metadata paints a portrait of each type, including the type name, methods of the type, parameters of each method of the type, each field of the type, and further details related to the loading and executing of that type at run time. Types are probably the most important construct in a .NET application, and metadata about types is used throughout the life cycle of a managed application. Here are a couple of examples. At startup, metadata is used to identify the entry point method where the program starts executing. During program execution, when a class is first touched, an `EECLASS` is built ostensibly from metadata to represent that type to the just-in-time compiler. The `EECLASS` is an important component of the just-in-time process. The `EECLASS` is further described in Chapter 13, “Advanced Debugging.”

To extend either manifest or type-related metadata, employ attributes. *Attributes* are the adjectives of a managed application and extend the description of an assembly, class, method, field, or other target. Attributes are recorded as metadata and extend the axiomatic metadata of an assembly. In addition, the Microsoft .NET Framework class library (FCL) offers predefined-custom and pseudo-custom attributes. *Obsolete* and *StructLayout* attributes are examples of predefined attributes. *Serializable* is an example of a pseudo-custom attribute. The *Obsolete* attribute marks an entity as deprecated, whereas the *StructLayout* attribute stipulates the memory layout of fields in the context of unmanaged memory. The latter attribute is essential when passing a managed type to an unmanaged function or application programming interface (API). You can augment the predefined attributes with programmer-defined custom attributes, where the limit is only your imagination. Applying a version number to a class, assigning the name of the responsible developer or team to a class, and identifying design documents used to architect an application are some ways to exploit custom attributes.

Metadata persisted to an assembly is organized as a nonhierarchical but relational database of cross-referencing tables. The metadata database has many tables that can—and often do—reference each other. However, no parent-child relationship between tables is ever implied. Each categorization of data is maintained in a separate table, such as the `TypeDef` and `MethodDef` tables. Types alone are stored in the `TypeDef` table. Each record of the `TypeDef` table represents a type. If there were six types in the assembly, there would be six records or rows in the `TypeDef` table. Methods of all types are stored in the `MethodDef` table. Each row of the `MethodDef` table describes a method. The `TypeDef` table references the `MethodDef` table to link types to member methods. The `MethodList` column of the `TypeDef` table has record indexes

(RIDs) into the MethodDef table. Extending this model, the MethodDef table has a ParamList column, which is index to the method's parameters found in the Param table.

Metadata tables are assigned unique table identifiers, which are 1-byte unsigned integers. For example, the table identifier for the TypeDef table is 2, whereas 6 identifies the MethodDef table. Metadata tables reserved for the run time are not published and not assigned an external table identifier for the RID. Table 10-1 lists some of the popular metadata tables.

Table 10-1 Metadata Tables

Table Name	Table ID	Table Description
Assembly	0x20	Data related to assembly
Field	0x04	Fields (data member) of types
MethodDef	0x06	Methods (member functions) of types
NestedClass	0x29	Type definitions for nested types
Param	0x08	Method parameters of methods
Property	0x17	Properties of types
TypeDef	0x02	Type definitions of types in current assembly
TypeRef	0x01	Type definitions of types external to this module

Metadata tables are collections of records and columns. A metadata table contains a certain type of data, and each record is an instance of that type. Columns represent specific data on each instance, and each column contains a constant or index. The index references another table or heap where the metadata token is an example of an index. Metadata tokens are used as metadata pointers, allowing tables to cross-reference each other. Metadata tables can be optimized (compressed) or not optimized. For the purpose of this book, it is assumed that metadata is optimized. Metadata that is not optimized requires intermediate tables for ordered access between tables.

Tokens

Metadata tokens cross-reference other metadata tables and heaps. *Tokens* are 4-byte unsigned integers and a combination of the table identifier and RID. As shown in Figure 10-1, the high byte is the table identifier, and the lower 3 bytes are the RID. A token into the Field table might be 04000002. The token refers to the second row of the Field table. The RID is one-based, not zero-based. Because tokens are padded with zeros, the run time might optimize them. Metadata tokens are probably the most public manifestation of metadata. You will repeatedly see metadata tokens over the next few chapters.

Table ID 0	RID 1	2	3
---------------	----------	---	---

Figure 10-1 Layout of a metadata token

In addition to other tables, metadata tables reference metadata heaps. Records of metadata tables hold fixed-length metadata information. Variable-length data is stored in one of the metadata heaps. Method signatures are variable length and typical of content found on the String heap.

Metadata Heaps

The four metadata heaps are as follows: String, Userstring, Blob, and GUID.

- The String heap is an array of null-terminated strings. Namespace, type, field, and method names, as well as other identifiers, are stored on the String heap.
- User-defined strings are not placed on the String heap but instead reside on the Userstring heap, which is also an array of null-terminated strings. String literals from your program are cached on this heap.
- The Blob heap is a binary heap and a composite of length prefix data, such as default values, method signatures, and field signatures.
- The GUID heap is an array of globally unique identifiers (GUIDs). Yes, this is obvious. You might remember GUIDs from COM as 16-byte unique identifiers assigned to almost everything—most notably, class identifiers (CLSIDs) are assigned to class factories. What kind of GUID is stored on the GUID heap? The GUID heap contains module version identifiers (MVIDs).

Streams

Physically, metadata tables and heaps are persisted in *streams* as part of an assembly. Six possible streams, including streams for each metadata heap, are available in .NET. There are also two mutually exclusive streams, optimized and nonoptimized, which are reservoirs of metadata tables. Metadata tables are optimized or not optimized. There is no concept of partially optimized metadata tables. If the metadata tables are optimized, the optimized stream is present. Otherwise, the nonoptimized stream is available. Therefore, a managed application has at most five streams. Table 10-2 provides a complete list of the metadata streams.

Table 10-2 Metadata Streams

Name	Description
#~	Optimized or compressed metadata tables
#-	Nonoptimized metadata tables
#Blob	Physical repository of the blob heap
#GUID	Physical repository of the GUID heap
#String	Physical repository the String heap
#US	Physical repository of the Userstring heap

Metadata Validation

Managed execution is largely dependent on metadata. Improperly formed metadata could cause a managed application to fail unceremoniously. An assembly with bad metadata is like a house built on quicksand. Loading a class, just-in-time compilation, code access security, and other run-time operations depend on robust data. Metadata validation tests the correctness of metadata and is enacted preemptively, preventing applications with inferior metadata from being executed. Preventing application crashes manifested by improper metadata enforces code isolation.

Several tests are performed to validate metadata. Here is a short list:

- Cross-references between tables are validated.
- Offsets into metadata heaps are validated.
- Metadata tables must have a valid number of rows. For example, the Assembly table is allowed one row.
- Metadata tables cannot have duplicate rows.
- Several more tests are enacted to certify metadata.

Developers can request metadata validation on demand with the PEVerify and Intermediate Language Disassembler (ILDASM) tools. Both tools are included in the .NET Framework software development kit (SDK).

PEVerify submits an assembly for metadata validation and Microsoft intermediate language (MSIL) verification and then reports the results. (MSIL verification is discussed in Chapter 11, “MSIL Programming.”) This is the basic syntax for PEVerify:

PEVerify assemblyname

PEVerify validates the metadata of *assemblyname*. If metadata validation is successful, MSIL verification is applied next. MSIL verification is skipped if the metadata validation fails. If validation fails, execution is not viable. This removes a compelling reason to conduct MSIL verification. PEVerify offers a variety of optional arguments, including the capability to force MSIL verification even when the metadata validation fails.

Table 10-3 lists some of the PEVerify arguments.

Table 10-3 PEVerify Options

Argument	Description
<i>/break=errorcount</i>	Aborts verification when errors exceed <i>errorcount</i> .
<i>/clock</i>	Collects data and reports duration of verification and validation tests.
<i>/help</i>	Help information on parameters.

Table 10-3 PEXVerify Options

Argument	Description
<i>/ignore=errorcode1, errorcode2, errorcoden</i>	Ignores listed error codes.
<i>/il</i>	Conducts MSIL verification. When you use this command, if metadata validation is also required it must be requested explicitly.
<i>/md</i>	Conducts metadata validation. If MSIL verification and metadata validation are jointly desired, MSIL verification should be requested explicitly.
<i>/?</i>	Same as the <i>/help</i> argument.

The following is a simple Hello World application, which is compiled to `hello.exe`. It is a minimal application, in which not much can go wrong. PEXVerify will confirm this.

```
using System;

class starter {
    static void Main() {
        Console.WriteLine("Hello, world!");
    }
}
```

The following code shows the result of running PEXVerify on `hello.exe` with the */il* and */clock* options. Since the *md* command is omitted, metadata verification is skipped.

```
c:\>peverify /il /clock hello.exe

All classes and methods in hello.exe verified.
Timing: Total run      125 msec
        IL Ver.cycle   125 msec
        IL Ver.pure    93 msec
```

The elapsed cycle for validation and pure times is listed. Pure time is the duration of the actual metadata validation, whereas cycle encapsulates the startup and shutdown processes.

ILDASM is a .NET tool that performs validation and can browse and display the metadata of an assembly. ILDASM inspects an assembly using reflection and presents the results in a window, console, or file.

ILDASM Tool

ILDASM, which is a .NET disassembler and metadata browser, is a popular tool for developers. It proffers an internal representation of an assembly, which includes the metadata and MSIL code of an assembly in a variety of formats. ILDASM exercises reflection to inspect an assembly. The core syntax of ILDASM requires only an assembly name, which opens ILDASM and displays the metadata of the assembly:

```
ildasm assemblyname
```

The following simple application is a basic .NET application that references a library. The simple application has a *ZClass* and *ZStruct* type, whereas the dynamic-link library (DLL) publishes the *YClass* type.

```
using System;

namespace Donis.CSharpBook{

    interface IA {
    }

    struct ZStruct {
    }

    class Starter {

        public static void Main() {
            YClass obj1=new YClass();
            obj1.DisplayCreateTime();
            ZClass obj2=new ZClass();
            obj2.DisplayCreateTime();
        }
    }

    class ZClass: IA {

        public enum Flag {
            aflag,
            bflag
        }

        public event EventHandler AEvent=null;

        public void DisplayCreateTime() {
            Console.WriteLine("ZClass created at "+m_Time);
        }

        private string m_Time=DateTime.Now.ToLongTimeString();
        public string Time {
            get {
                return m_Time;
            }
        }
    }
}
```

Figure 10-2 is a view of Simple.exe from ILDASM. ILDASM displays a hierarchal object graph with an icon for each element of the application.

Some icons are collapsible and expandable, as indicated by a + or – symbol. The Assembly icon expands to show the details of the loaded assembly, the Namespace icon expands to show the members of the namespace, and so on. You can drill down the object graph from the

assembly down to the class members. An icon depicts each item category of the graph. Table 10-4 describes each icon for which the action is double-clicking the icon.



Figure 10-2 Simple.exe displayed in ILDASM

Table 10-4 Elements of ILDASM

Icon Descriptions	Action
Assembly	Shows elements of the assembly
Class	Shows members of a class
Enum	Shows members of enum type
Event	Views metadata and MSIL code of event
Field	Views metadata of field
Interface	Shows members of interface
Manifest	Views attributes of an assembly
Method	Views metadata and MSIL code of method
Namespace	Shows members of the namespace
Property	Views metadata and MSIL code of property
Static Field	Views metadata of static field
Static Method	Views metadata and MSIL code of static method
Value Type	Shows members of a value type

Some elements are displayed twice. For example, a property is presented as itself and separately as *accessor* and *mutator* methods.

ILDASM has a variety of command-line options. Table 10-5 lists these parameters.

Table 10-5 ILDASM Options

ILDASM Option	Description
<i>Out</i>	Renders metadata and MSIL to a text file.
<i>Text</i>	Renders metadata and related MSIL to console.
<i>HTML</i>	Combines with the <i>out</i> option to display metadata and MSIL in an HTML format.

Table 10-5 ILDASM Options

ILDASM Option	Description
<i>RTF</i>	Renders metadata and MSIL in Rich Text Format.
<i>Bytes</i>	Shows MSIL code with opcodes and related bytes.
<i>Raweh</i>	Shows label form of <i>try</i> and <i>catch</i> directives in raw form.
<i>Tokens</i>	Shows metadata tokens.
<i>Source</i>	Shows MSIL interlaced with commented source code; for this command, the source code and debug file must be accessible.
<i>Linenum</i>	Inserts line directives into an output stream that matches source code to MSIL. This command requires the debug file.
<i>Visibility</i>	Disassembles only members with the stated visibility: <i>pub</i> (public), <i>pri</i> (private), <i>fam</i> (family), <i>asm</i> (assembly), <i>FAA</i> (family and assembly), <i>foa</i> (family and assembly), and <i>PSC</i> (private scope).
<i>Pubonly</i>	Disassembles only public elements; short notation for <i>visibility=pub</i> .
<i>QuoteAllNames</i>	Brackets all identifiers in single quotes.
<i>NOCA</i>	Excludes custom attributes.
<i>CAVerbal</i>	Displays blob information of custom attributes in symbolic form and not binary.
<i>NOBAR</i>	Do not display progress bar.
<i>UTF8</i>	Renders output file in UTF8 (default ANSI).
<i>UNICODE</i>	Renders output file in UNICODE.
<i>NOIL</i>	Do not disassemble language source code.
<i>Forward</i>	Generates forward references and assemble in the Class Structure Declaration section.
<i>TypeList</i>	Displays list of types.
<i>Headers</i>	Includes DOS, PE, COFF, CLR, and metadata header information.
<i>Item</i>	Disassembles a particular class or method.
<i>Stats</i>	Displays statistical information on file, PE Header, CLR Header, and metadata.
<i>ClassList</i>	Provides a commented list of classes with attributes.
<i>All</i>	Combination of the <i>Header</i> , <i>Bytes</i> , <i>Stats</i> , <i>ClassLists</i> , and <i>Tokens</i> commands.
<i>Metadata</i>	Displays specific information related to metadata.
<i>Objectfile</i>	Shows metadata of a library file.

The user interface of ILDASM presents the same choices as the command-line options. The following command line is typical. It disassembles *simple.exe* and outputs the resulting metadata, MSIL, metadata tokens, and source code in the *simple.il* file.

```
ildasm /out=simple.il /source /tokens simple.exe
```

The *source* option of the preceding command interlaces source code in between MSIL code. The source code is commented. Associating MSIL to source code is invaluable when debugging.

The tokens generated per the `tokens` option are also commented. The disassembly created by ILDASM is a valid MSIL program that can be recompiled (which is the reason for the `il` extension, as in `client.il`). The assembly can be reassembled with the ILASM compiler, which compiles MSIL code. The newly assembled assembly is identical to the original assembly.

Some ILDASM options impede the creation of a full disassembly. When a partial disassembly is requested, ILDASM issues a warning, which prevents you from attempting to use a partial assembly as a full assembly. One limitation is that partial assemblies cannot be reassembled using ILASM. The following command creates a partial assembly:

```
ildasm /out=simple.il /item=Donis.CSharpBook.ZClass simple.exe
```

The preceding command targets only the `ZClass` of the `simple.exe` assemblies. Because other types are omitted from the disassembly, it is not complete. For this reason, a warning is added to the output file. Following is a partial listing of the output file with the embedded warning:

```
// Microsoft (R) .NET Framework IL Disassembler. Version 2.0.50601.0
// Microsoft Corporation. All rights reserved.

// warning : THIS IS A PARTIAL DISASSEMBLY, NOT SUITABLE FOR RE-ASSEMBLING

.class private auto ansi beforefieldinit Donis.CSharpBook.ZClass
    extends [mscorlib]System.Object
    implements Donis.CSharpBook.IA
{
    .field private class [mscorlib]System.EventHandler AEvent
    .field private string m_Time
    .method public hidebysig specialname instance void
```

This is the final example of ILDASM and command-line options. This command profiles the metadata yields counts, validates the metadata, and persists the results to the `simple.txt` file:

```
ildasm /metadata=csv /metadata=validate /out=simple.txt simple.exe
```

Reflection

An assembly is a piñata stuffed with goodies such as type information, MSIL code, and custom attributes. You use reflection to break open the assembly piñata to examine the contents. Reflection adds important features to .NET, such as metadata inspection, run-time creation of types, late binding, MSIL extraction, self-generating code, and so much more. These features are crucial to solving complex real-world problems that developers face every day.

The *Reflection* namespace is the container of all things related to reflection. *Assembly*, *Module*, *LocalVariableInfo*, *MemberInfo*, *MethodInfo*, *FieldInfo*, and *Binder* are some of the important members of the *Reflection* namespace. Reflection exposes several predefined customer

attributes, such as *AssemblyVersionAttribute*, *AssemblyKeyFileAttribute*, and *AssemblyDelaySignAttribute*. The *Reflection* namespace contains other reflection-related namespaces; most notably the *Reflection.Emit* nested namespace. *Reflection.Emit* is a toolbox filled with tools for building assemblies, classes, and methods at run time, including the ability to emit metadata and MSIL code. *Reflection.Emit* is reviewed in Chapter 11.

The central component of reflection is the *Type* object. Its interface can be used to interrogate a reference or value type. This includes browsing methods, fields, parameters, and custom attributes. General information pertaining to the type is also available via reflection, including identifying the hosting assembly. Beyond browsing, *Type* objects support more intrusive operations. You can create instances of classes at run time and perform late binding of methods.

Obtaining a Type Object

The *Object.GetType* method, the *typeof* operator, and various methods of the *Assembly* object return a *Type* object. *GetType* is a member of the *Object* class, which is the ubiquitous base class. Therefore, *GetType* is inherited by .NET types and available to all managed instances. Each instance can call *GetType* to return the related *Type* object. The *typeof* operator extracts a *Type* object directly from a reference or value type. Assembly objects have several members that return one or more *Type* objects. For example, the *Assembly.GetTypes* method enumerates and returns the *Types* of the target assembly.

As a member base class object, *GetType* is accessible to all instances of reference and values types. *GetType* returns the *Type* object of the instance. This is the syntax of the *GetType* method:

```
Type Type.GetType method:  
Type GetType()
```

The following code creates instances of a value and a reference type, which are passed individually as object parameters to successive calls to the *DisplayType* method, which homogenizes the objects, where each object loses its distinction. The function extracts the type of the instance and displays the *Type* name. Finally, if the *Type* represents a *ZClass*, the *ZClass.Display* method is called:

```
using System;  
  
namespace Donis.CSharpBook{  
    class Starter{  
  
        static void Main() {  
            int localvalue=5;  
            ZClass objZ=new ZClass();  
            DisplayType(localvalue);  
            DisplayType(objZ);  
        }  
    }  
}
```

```

static void DisplayType(object parameterObject) {
    Type parameterType=parameterObject.GetType();
    string name=parameterType.Name;
    Console.WriteLine("Type is "+name);
    if(name == "ZClass") {
        ((ZClass) parameterObject).Display();
    }
}

}

class ZClass {

    public void Display() {
        Console.WriteLine("ZClass::Display");
    }
}
}

```

The *typeof* operator returns a *Type* object from a type. The *typeof* operator is evaluated at compile time, whereas the *GetType* method is invoked at run time. For this reason, the *typeof* operator is quicker but less flexible than the *GetType* method:

typeof operator:
typeof(*type*)

An assembly is typically the host of multiple types. *Assembly.GetTypes* enumerates the types defined in an assembly. *Assembly.GetType* is overloaded four times. The zero-argument method returns the *Type* object of the *Assembly* and essentially the *GetType* method derived from the *Object* class. The one-argument version, in which the parameter is a string, returns a specific type defined in the assembly. The final two versions of *GetType* are an extension of the one-argument overloaded method. The two-argument method also has a Boolean parameter. When true, the method throws an exception if the type is not located. The three-argument version has a second Boolean parameter that stipulates case sensitivity. If this parameter is false, the case of the type name is significant.

Here are the methods:

Assembly.GetTypes method:
Type [] GetTypes()
Assembly.GetType method:
Type GetType()
Type GetType(string typename)
Type GetType(string typename, bool throwError)
Type GetType(string typename, bool throwError, bool ignoreCase)

An assembly can be diagrammed through reflection. The result is called the Reflection tree of that assembly. Each element of reflection, such as an *AssemblyInfo*, *Type*, *MethodInfo*, and *ParameterInfo* component, is placed on the tree. *AppDomain* is the root of the tree; *GetAssemblies* expands the tree from the root. The Reflection tree is a logical, not a physical representation.

Assembly, *Type*, and *ParameterInfo* are some of the branches on the Reflection tree. You can explore the Reflection tree while inspecting the metadata of the application by using enumeration. For example, *Assembly.GetCurrentAssembly* returns the current assembly. *Assembly.GetTypes* will enumerate the types defined in the assembly. *Type.GetMethods* will enumerate the methods of each type. This process can continue until the application is fully reflected.

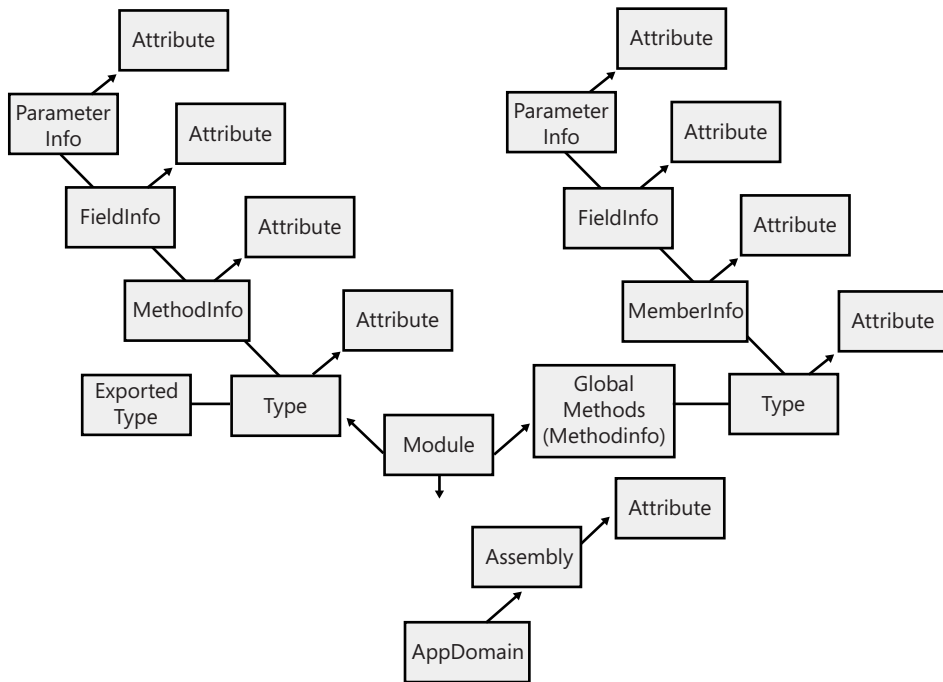


Figure 10-3 Diagram of the Reflection tree

Loading Assemblies

Assemblies near the root of the reflection tree can be loaded at run time using *Assembly.Load* and *Assembly.LoadFrom*. *Assembly.Load* uses the assembly loader to locate and bind to the correct assembly. *Assembly.LoadFrom* consults the assembly resolver to locate an assembly, which uses a combination of the strong name identifier and probing to bind and then load an assembly. The strong name includes the simple name, version, culture, and public key token of the assembly. Probing is the algorithm for locating an assembly. Both *Assembly.Load* and *Assembly.LoadFrom* are static methods that are overloaded several times. This is the core syntax:

```

Assembly.Load method:
static Assembly Assembly.Load(string assemblyName)
Assembly.LoadFrom method:
static Assembly Assembly.LoadFrom(string AssemblyFileName)

```

When the chosen assembly isn't found, *Assembly.Load* and *Assembly.LoadFrom* will fail. The Assembly Binding Log Viewer Tool (fuslogvw.exe), included in the .NET Framework SDK, is a useful tool for diagnosing probing failures because it displays information on the binding error.

The following sample code accentuates the difference between *Assembly.Load* and *Assembly.LoadFrom*:

```
using System;
using System.Reflection;

namespace Donis.CSharpBook{
    class Starter{
        static void Main(){
            Assembly library=Assembly.Load("library, Version=2.0.0.0, "+
                "Culture=Neutral, PublicKeyToken=9b184fc90fb9648d");
            Console.WriteLine("Assembly.Load: {0}", library.FullName);
            library=Assembly.LoadFrom("library.dll");
            Console.WriteLine("Assembly.LoadFrom {0}", library.FullName);
        }
    }
}
```

Assembly.Load and *Assembly.LoadFrom* reference external assemblies. How about referencing the currently executing assembly? *Assembly.GetExecutingAssembly* is a static method and returns a reference to the currently executing assembly. This is valuable for interrogating the metadata or MSIL of the running assembly.

Assembly.GetExecutingAssembly method:
static Assembly Assembly.GetExecutingAssembly()

Both *Assembly.Load* and *Assembly.LoadFrom* return a reference to an assembly. That assembly can be reflected, code loaded, and then executed. *Assembly.ReflectionOnlyLoad* and *Assembly.ReflectionOnlyLoadFrom* load an assembly only for reflection, so the code cannot be executed later. Although you could reflect a type and iterate all the methods, you could not invoke a method of that type. To confirm the way an *Assembly* is loaded, use the *ReflectionOnly* property. *ReflectionOnly* is a Boolean attribute of an *Assembly* and is true if an *Assembly* is loaded for reflection only. *Assembly.ReflectionOnlyLoad* and *Assembly.ReflectionOnlyLoadFrom* are equivalent to *Assembly.Load* and *Assembly.LoadFrom*, respectively, without the execute capability. *ReflectionOnly* yields performance benefits.

The following program benchmarks the performance of *Assembly.Load* versus *Assembly.ReflectionOnlyLoad*. The *DateTime* function has poor resolution and is inferior for most testing scenarios. Instead, a high-performance timer is needed to obtain added resolution. The *QueryPerformanceCounter* API returns a high-performance counter accurate to a nanosecond. Prior to .NET Framework 2.0, *QueryPerformanceCounter* was available only through interoperability. The *Stopwatch* class, which is a thin wrapper for the *QueryPerformanceCounter* and

related APIs, was introduced in .NET Framework 2.0 and found in the *System.Diagnostics* namespace:

```
using System;
using System.Reflection;
using System.Diagnostics;

namespace Donis.CSharpBook{
    class OnlyLoad{
        static void Main() {
            Stopwatch duration=new Stopwatch();
            duration.Reset();
            duration.Start();
            Assembly a=Assembly.Load("library");
            duration.Stop();
            Console.WriteLine(duration.ElapsedTicks.ToString());
            duration.Reset();
            duration.Start();
            a=Assembly.ReflectionOnlyLoad("library");
            duration.Stop();
            Console.WriteLine(duration.ElapsedTicks.ToString());
        }
    }
}
```

Execution time of *Assembly.Load* versus *Assembly.ReflectionOnlyLoad* might vary between different implementations of the CLI and other factors. The program compares ticks, which is an abstraction of time. Running the program several times indicates that *Assembly.ReflectionOnlyLoad* is about 29 percent faster than *Assembly.Load*. This is only an approximation.

Type.ReflectionOnlyGetType combines the functionality of *Assembly.ReflectionOnlyLoad* and the *typeof* operator. The named assembly is loaded for inspection only, and a *Type* object is returned to the specified type. Because the assembly is opened only for browsing, you cannot create an instance of the type or invoke a method on the type. *ReflectionOnlyGetType* is a static method of the *Type* class and loads a *Type* for inspection only. The method returns a *Type* object:

```
Type.ReflectionOnlyType method:
static Type ReflectionOnlyType(string typeName, bool notFoundException,
    bool ignoreCase)
```

The *typeName* parameter is a combination of the *Assembly* and *Type* names. The *Assembly* and *Type* names are comma-delimited. To raise an exception if the type is not found, set the *notFoundException* parameter to *true*. When false, the *ignoreCase* parameter indicates that the *Type* name is case sensitive:

```
using System;
using System.Reflection;

namespace Donis.CSharpBook{
    class ReflectOnlyType{
```

```

static void Main() {
    Type zType=Type.ReflectionOnlyGetType("Donis.CSharpBook.ClassA,
        Library", false, false);
    Console.WriteLine(zType.Name);
}
}
}

```

Browsing Type Information

Inspecting a type begins with the *Type* object. *Reflection* has a straightforward interface for mining the metadata of a type. Inspecting the metadata of a type essentially consists of spanning a series of collections. The *Type* object interface publishes several methods and properties related to reflection.

Type.GetMembers returns the ultimate collection: the collection of all members. All members, whether the member is a method, field, event, or property, are included in the collection. *GetMembers* returns a *MemberInfo* array that contains an item for each member. *GetMember* returns a single *MemberInfo* object for the named member. *MemberInfo.MemberType* is a property of the *MemberInfo* type, which is a bitwise enumeration distinguishing a member as a method, field, property, event, constructor, or something else. (See Table 10-6.) *MemberInfo* has relatively few properties and operations. Here are some of the more useful. The *MemberInfo.Name* offers the name of the type member. *MemberInfo.MetadataToken*, another property, returns the metadata token of the member. *MemberInfo.ReflectedType* provides the *Type* object from which the *MemberInfo* object was extracted.

Table 10-6 *MemberTypes* Enumeration

MemberType	Value
<i>MemberTypes.Constructor</i>	0x01
<i>MemberTypes.Custom</i>	0x40
<i>MemberTypes.Event</i>	0x02
<i>MemberTypes.Field</i>	0x04
<i>MemberTypes.Method</i>	0x08
<i>MemberTypes.NestedType</i>	0x80
<i>MemberTypes.Property</i>	0x10
<i>MemberTypes.TypeInfo</i>	0x20
<i>MemberTypes.All</i>	0xBF

Type.GetMembers creates a basket that contains all the members of the reflected type. You can be somewhat more granular. *Type.GetMethods* or *Type.GetMethod* returns a collection of methods or a specific method. *Type.GetFields* or *Type.GetField* similarly returns a collection of fields or a specific field. Table 10-7 lists the methods that return specific collections where the non-plural method returns a specific member of that collection.

Table 10-7 Type Methods That Return Metadata Collections

Method	Returns	Type of Member
<i>GetConstructors</i>	<i>ConstructorInfo []</i>	Constructor
<i>GetCustomAttributes</i>	<i>Object []</i>	Custom attribute
<i>GetDefaultMembers</i>	<i>MemberInfo []</i>	Default member
<i>GetEvents</i>	<i>EventInfo []</i>	Event
<i>GetFields</i>	<i>FieldInfo []</i>	Field
<i>GetInterfaces</i>	<i>Type []</i>	Implemented interface
<i>GetMembers</i>	<i>MemberInfo []</i>	All members
<i>GetMethods</i>	<i>MethodInfo []</i>	Method
<i>GetNestedTypes</i>	<i>Type []</i>	Nested Type
<i>GetProperties</i>	<i>PropertyInfo []</i>	Property

The *Type.GetMethods* that return *MethodInfo* arrays are overloaded to be called with no parameters or with a single parameter, which is *BindingFlags*:

```
Type.GetMethods method:
MethodInfo [] GetMethods();
MethodInfo[] GetMethods(BindingFlags binding)
```

BindingFlags is a bitwise enumeration that filters the results of a collection. For example, to include private members in a collection stipulates the *BindingFlags.NonPublic* flag. Some *BindingFlags*, such as *InvokeMember*, are not applicable in this context. When stipulating *BindingFlags*, there are no default flags. If you specify one flag, you must specify every required flag. Each inclusion must be specified explicitly. The zero-argument version of *GetMethods* grants default bindings, which vary depending on the method. At a minimum, most of the methods default to *BindingFlags.Public* and *BindingFlags.Instance*. Notably, the *BindingFlags.Static* flag is not always defaulted and static members are often excluded from a collection. The following code first iterates private instance (nonpublic) members. Afterward, static public members are iterated.

```
using System;
using System.Reflection;

namespace Donis.CSharpBook{
    class DumpType{
        public static void Main() {
            ZClass zObj=new ZClass();
            Type tObj=zObj.GetType();
            MemberInfo [] members=tObj.GetMembers(
                BindingFlags.Instance|
                BindingFlags.NonPublic);
            foreach(MemberInfo member in members) {
                Console.WriteLine(member.Name);
            }
            members=tObj.GetMembers(
                BindingFlags.Public|BindingFlags.Static);
```

```

Console.WriteLine(" ");
    foreach(MemberInfo member in members) {
        Console.WriteLine(member.Name);
    }
}

class ZClass {
    private int vara=5;
    public int PropA {
        get {
            return vara;
        }
    }
    static public void MethodA() {
        Console.WriteLine("ZClass::MethodA called.");
    }
}
}

```

The following application calls *DumpMethods* to dump the public methods of a class. This code demonstrates various aspects of *Reflection*.

```

using System;
using System.Reflection;

namespace Donis.CSharpBook{
    class DumpType{
        static void Main(string [] argv) {
            targetType=LoadAssembly(argv[0], argv[1]);
            DumpReportHeader();
            DumpMethods();
        }

        static public Type LoadAssembly(string t, string a) {
            return Type.ReflectionOnlyGetType(t+"a, false, true);
        }

        static void DumpReportHeader() {
            Console.WriteLine("\n{0} type of {1} assembly",
                targetType.Name, targetType.Assembly.GetName().Name);
            Console.WriteLine("\n{0,22}\n", "[ METHODS ]");
        }

        static void DumpMethods() {
            string dashes=new string('-', 50);
            foreach(MethodInfo method in targetType.GetMethods()) {
                Console.WriteLine("{0,12}{1,-12}", " ", method.Name+" "+
                    "<" +method.ReturnParameter.ParameterType.Name+">");
                int count=1;
                foreach(ParameterInfo parameter in method.GetParameters()){
                    Console.WriteLine("{0, 35}{1, -12}",
                        " ", (count++).ToString()+" "+ parameter.Name+
                        " (" +parameter.ParameterType.Name+"");
                }
            }
        }
    }
}

```

```

        Console.WriteLine("{0,12}{1}", " ", dashes);
    }
}

private static Type targetType;
}
}

```

In the preceding code, a type name and an assembly name are read from the command line. The type to be dumped is *argv[0]*, while the assembly hosting the type is *argv[1]*. With this information, the *LoadAssembly* method employs *Type.ReflectionOnlyGetType* and loads the type for inspection only. The *DumpMethods* function iterates the methods of the target type, and then iterates the parameters of each method. The name of each method and parameter is displayed. This dumps the members of the *Console* class:

```
dumpmethods System.Console mscorlib.dll
```

Dynamic Invocation

Methods can be dynamically invoked at run time using reflection. The benefits and perils of early binding versus late binding were reviewed in the discussion on delegates earlier in the book.

In dynamic binding, you build a method signature at run time and then invoke the method. This is somewhat later than late binding with delegates. When compared with delegates, dynamic binding is more flexible, but is regrettably slower. At compile time, the method signature must match the delegate at the site of the run-time binding. Dynamic binding removes this limitation, and any method can be invoked at the call site, regardless of the signature. This is more flexible and extensible than run-time binding.

In reflection, there are two approaches to invoking a method dynamically: *MethodInfo.Invoke* and *Type.InvokeMember*, where *MethodInfo.Invoke* is the simplest solution. However, *Type.InvokeMember* is more malleable. The basic syntax of *MethodInfo.Invoke* requires only two parameters: an instance of a type and an array of parameters. The method is bound to the instance provided. If the method is static, the instance parameter is null. To avoid an exception at run time, which is never fun, care must be taken to ensure that the instance and parameters given to *MethodInfo.Invoke* match the signature of the function.

This is the *MethodInfo.Invoke* syntax:

```

object Invoke1(object obj, object [] arguments)
object Invoke2(object obj, BindingFlags flags, Binder binderObj,
    object [] arguments, CultureInfo culture)

```

The second *Invoke* method has several additional parameters. The *obj* parameter is the instance that method is bound. If invoking a static method, the *obj* parameter should be null but is otherwise ignored. *BindingFlags* are next and further describe the *Invoke* operation,

such as *Binding.InvokeMethod*. When no binding flags are specified, the default is *BindingFlags.DefaultBinding*. *Binderobj* is used to select the appropriate candidate among overloaded methods. *Arguments* is the array of method arguments as defined by the method signature. The *culture* argument sets the culture, which defaults to the culture of the system. Return is the method return or null for a void return.

Alternatively, you can invoke a method dynamically at run time using *Type.InvokeMember*, which is overloaded several times.

This is the *Type.InvokeMember* syntax:

```
object InvokeMember1(string methodName, BindingFlags flags,
    Binder binderObj, object typeInstance, object [] arguments)
object InvokeMember2(string methodName, BindingFlags flags,
    Binder binderObj, object typeInstance, object [] arguments,
    CultureInfo culture)
object InvokeMember3(string methodName, BindingFlags flags,
    Binder binderObj, object typeInstance, object [] arguments,
    ParameterModifier [] modifiers, CultureInfo culture,
    string [] namedParameters)
```

InvokeMember¹ is the core overloaded method, and it has several parameters. The *methodName* parameter is the name of the method to invoke. The next parameter is *BindingFlags*. *Binderobj* is the binder used to discriminate between overloaded methods. The object to bind the method again is *typeInstance*. *arguments* is the array of method parameters. *InvokeMember²* adds an additional parameter. To set the culture, use the culture parameter. *InvokeMember³* is the final overload with one additional parameter: *namedParameters*, which is used to specify named parameters.

In the following code, dynamic invocation is demonstrated with *MethodInfo.Invoke* and *Type.InvokeMember*:

```
using System;
using System.Reflection;

namespace Donis.CSharpBook{

    class Starter{

        static void Main(){
            ZClass obj=new ZClass();
            Type tObj=obj.GetType();
            MethodInfo method=tObj.GetMethod("MethodA");

            method.Invoke(obj, null);
            tObj.InvokeMember("MethodA", BindingFlags.InvokeMethod,
                null, obj, null);
        }
    }
}

class ZClass {
```

```

        public void MethodA() {
            Console.WriteLine("ZClass.Method invoked");
        }
    }
}

```

Binders

Members such as methods can be overloaded. In reflection, binders determine the specific method to invoke from a list of possible candidates. The default binder selects the best match based on the number and type of arguments. You can provide a custom binder and choose a specific overloaded member. Both *MethodInfo.Invoke* and *Type.InvokeMember* offer a binder argument for this reason.

The *Binder* class is an abstract class; as such, it is implemented through a derived concrete class. *Binder* has abstracted methods to select a field, property, and method from available overloaded candidates. *Binder* is a member of the *Reflection* namespace. Table 10-8 lists the public members of the *Binder* class. Each method included in the table is abstract and must be overridden in any derivation.

Table 10-8 Abstract Methods of the *Binder* Class

Binder Method	Description
<i>BindToField</i>	Selects a field from a set of overloaded fields
<i>BindToMethod</i>	Selects a method from a set of overloaded methods
<i>ChangeType</i>	Coerces the type of an object
<i>ReorderArgumentArray</i>	Resets the argument array; associated with the state parameter of <i>BindToMethod</i> member
<i>SelectMethod</i>	Selects a method from candidate methods
<i>SelectProperty</i>	Selects a property from candidate properties

Binder.BindToMethod is called when a method is invoked dynamically. To override the default selection criteria of overloaded methods, create a custom binder class. Inherit the binder class and at least minimally override and implement each abstract method of the base class. How the binder is used determines the methods to fully implement. If the *MethodInfo.Invoke* and *Type.InvokeMember* methods are called with the *BindingFlags.InvokeMethod* flag set, *BindToMethod* will be invoked and should be completely implemented.

The *Binder.BindToMethod* syntax is as follows:

```

public abstract BindToMethod(BindingFlags flags, MethodBase [] match,
    ref object [] args, ParameterModifier [] modifiers, CultureInfo culture,
    string [] names, out object state)

```

The first parameter of *BindToMethod* is *BindingFlags*, which is the usual assortment of binding flags. *match* is a *MethodBase* array with an element for each possible candidate. If there are three candidates, *match* has three elements. At present, *MethodBase*-derived classes are limited

to *ConstructorInfo*; *MethodInfo.args* are the values of method parameters. *Modifiers* is an array of *ParameterModifier* used with parameter signatures of modified types. *Culture* sets the culture. *Names* are the identifiers of methods included as candidates. The *modifiers*, *culture*, and *names* parameters can be null. The final parameter, *state*, is used with parameter reordering. If this parameter is non-null, *Binder.ReorderArgumentArray* is called after *BindToMethod* and returns the parameters to the original order.

There is no prescription for selecting a method from a set of candidates. You are free to be creative and employ whatever logic seems reasonable. Here is a partially implemented but workable custom *Binder* class. *BindToMethod* is implemented but the other methods are essentially stubbed. This code is somewhat circumscribed and not written for general-purpose application.

```
using System;
using System.Reflection;
using System.Globalization;

class CustomBinder:Binder {
    public override FieldInfo BindToField(BindingFlags bindingAttr,
        FieldInfo[] match, object value, CultureInfo culture) {
        return null;
    }

    public override MethodBase BindToMethod(BindingFlags bindingAttr,
        MethodBase[] match, ref object[] args,
        ParameterModifier[] modifiers, CultureInfo culture,
        string[] names, out object state) {
        Console.WriteLine("Overloaded Method:");
        foreach(MethodInfo method in match) {
            Console.WriteLine("\n {0} (" + method.Name);
            foreach(ParameterInfo parameter in
                method.GetParameters()) {
                Console.WriteLine(" "+parameter.ParameterType.ToString());
            }
            Console.WriteLine(" )");
        }
        Console.WriteLine();
        state=null;
        if(long.Parse(args[0].ToString())>int.MaxValue) {
            return match[0];
        }
        else {
            return match[1];
        }
    }

    public override object ChangeType(object value, Type type,
        CultureInfo culture) {
        return null;
    }
}
```

```

    public override void ReorderArgumentArray(ref object[] args,
        object state){
    }

    public override MethodBase SelectMethod(BindingFlags bindingAttr,
        MethodBase[] match, Type[] types,
        ParameterModifier[] modifiers) {
        return null;
    }

    public override PropertyInfo SelectProperty(BindingFlags bindingAttr,
        PropertyInfo[] match, Type returnType, Type[] indexes,
        ParameterModifier[] modifiers) {
        return null;
    }
}

class ZClass {
    public void MethodA(long argument) {
        Console.WriteLine("Long version: "+argument.ToString());
    }

    public void MethodA(int argument) {
        Console.WriteLine("Int version: "+argument.ToString());
    }

    public void MethodA(int argument, int argument2) {
        Console.WriteLine("ZClass:Method 2 arg");
    }
}

class Starter {
    public static void Main() {
        ZClass obj=new ZClass();
        Type tObj=obj.GetType();
        CustomBinder theBinder=new CustomBinder();
        tObj.InvokeMember("MethodA", BindingFlags.InvokeMethod,
            theBinder, obj, new Object[] {int.MinValue});
        Console.WriteLine();
        tObj.InvokeMember("MethodA", BindingFlags.InvokeMethod, theBinder,
            obj, new Object[] {long.MaxValue});
    }
}

```

In the preceding code, *CustomBinder* inherits from the *Binder* class. As mentioned, *BindToMethod* is the sole method implemented in the code. This function lists the signatures of each candidate. The appropriate method is then chosen. The first *foreach* loop iterates the candidates while listing the method names. The inner *foreach* loop iterates and lists the parameters of each *MethodInfo* object. This version of *BindToMethod* is written specifically for the one-argument methods of the *ZClass BindToMethod*. The argument value is tested. If the value is within the range of a long, the first method is returned. This method has a long parameter. Otherwise, the second candidate, which has an integer parameter, is returned.

In *Main*, an instance of the *ZClass* and custom binder is created. *ZClass* is a simple class with an overloaded method. *ZClass.MethodA* is overloaded three times. The type object is then extracted from the *ZClass* instance, and *Type.InvokeMember* is called twice with the custom binder. *InvokeMember* is called to dynamically invoke “MethodA”, first with an integer parameter and then with a long parameter. This is the output from the application:

```
Overloaded Method:
MethodA ( System.Int64 )

MethodA ( System.Int32 )

Int version: -2147483648

Overloaded Method:

MethodA ( System.Int64 )

MethodA ( System.Int32 )

Long version: 9223372036854775807
```

Type Creation

Until now, the emphasis of this chapter has been on reflecting existing objects. Object instances can also be created dynamically at run time. You can reflect, bind methods, and otherwise treat the dynamic object as a static object. As often with *Reflection*, the primary benefit is added flexibility. What if the type of an instance is not determinable at compile time? With *Reflection*, that decision can be delayed until run time, when the particular class can be chosen by the user, stipulated in a configuration file, or otherwise selected by some intermediary.

The *Activator* class is a member of the *Reflection* namespace and facilitates the creation of objects at run time. *Activator* consists of four static member methods: *CreateInstance* creates an instance of a type; *CreateInstanceFrom* leverages *Assembly.LoadFrom* to reference an assembly and then create an instance of a type found in the assembly; and *CreateComInstanceFrom* and *GetObject* instantiate a COM object and a proxy to a *Remote* object, respectively. To simply create an instance of a .NET type, call *CreateInstance* or *CreateInstanceFrom*. Both *CreateInstance* and *CreateInstanceFrom* are overloaded several times. This is the list of *CreateInstance* methods:

```
Activator.CreateInstance syntax:
static T CreateInstance<T>()
static public ObjectHandle CreateInstance(ActivationContext context)
static object CreateInstance(Type type)
static public ObjectHandle CreateInstance(ActivationContext context, string [] customData)
static ObjectHandle CreateInstance(string assemblyName, string typeName)
static object CreateInstance(Type type, bool ctorPublic)
static object CreateInstance(Type type, object [] ctorArgs)
static ObjectHandle CreateInstance<T, U> (T, U)
static ObjectHandle CreateInstance(string assemblyName, string typeName,
    object [] activationAttributes)
static object CreateInstance(string assemblyName, string typeName,
```

```

    object [] activationAttributes)
static object CreateInstance(Type type, BindingFlags bindingAttr,
    Binder binder, object [] args, CultureInfo culture)
static object CreateInstance(Type type, object [] args, object [] activationAttributes)
static object CreateInstance(Type type, BindingFlags bindingAttr, Binder binder,
    object [] args, CultureInfo culture, object [] activationAttributes)
static ObjectHandle CreateInstance(string assemblyName, string typeName,
    bool ignoreCase, BindingFlags bindingAttr, Binder binder,
    object [] args, CultureInfo culture, object [] activationAttributes,
    Evidence securityInfo)

```

Some *CreateInstance* methods—and all *CreateInstanceFrom* methods—return an *ObjectHandle*, which is usually returned when creating an instance of a type foreign to the current assembly. *ObjectHandle* is found in the *System.Runtime.Remoting* namespace. *ObjectHandle.Unwrap* unwraps the *ObjectHandle* to disclose a proxy to the remoted object. Alternatively, the *AppDomain.CreateInstanceFromAndUnwrap* method creates an instance of the type and returns the object unwrapped with one less step. Of course, this means that an additional step of obtaining an *AppDomain* object is necessary. However, if the *AppDomain* object is handy, *CreateInstanceFromAndUnwrap* is convenient.

The following code creates three instances of the same type. A local and two remote proxies to object instances are constructed:

```

using System;
using System.Reflection;
using System.Runtime.Remoting;

namespace Donis.CSharpBook{

    class ZClass {
        public void MethodA(DateTime dt) {
            Console.WriteLine("MethodA invoked at "+
                dt.ToLongTimeString());
        }
    }

    class Starter{

        static void Main() {
            CreateLocal();
            CreateRemote1();
            CreateRemote2();
        }

        static void CreateLocal() {
            object obj=Activator.CreateInstance(typeof(ZClass));
            ((ZClass) obj).MethodA(DateTime.Now);
        }

        static void CreateRemote1() {
            ObjectHandle hobj=Activator.CreateInstance("library",
                "Donis.CSharpBook.ZClass");
            object obj=hobj.Unwrap();

```

```

        MethodInfo method=obj.GetType().GetMethod("MethodA");
        method.Invoke(obj, new object [1] {DateTime.Now});
    }

    static void CreateRemote2() {
        AppDomain domain=AppDomain.CurrentDomain;
        object obj=domain.CreateInstanceFromAndUnwrap("library.dll",
            "Donis.CSharpBook.ZClass");
        MethodInfo method=obj.GetType().GetMethod("MethodA");
        method.Invoke(obj, new object [1] {DateTime.Now});
    }
}
}
}

```

The preceding code presents three vehicles for creating an instance of a type, binding a method to the type, and finally invoking that method dynamically. Dynamically invoking a method through casting is a mechanism not previously demonstrated. (*Method.Invoke* and *Type.InvokeMember* were reviewed earlier.)

The code for engaging a method through casting is listed as follows. Calling a method dynamically through casting has substantial performance gains when compared with *MethodInfo.Invoke* or *Type.InvokeMember*. (*MethodInfo.Invoke* and *Type.InvokeMember* were reviewed earlier in this chapter.)

```
((ZClass) obj).MethodA(DateTime.Now);
```

Either directly or indirectly, the *Activator.CreateInstance* and *CreateInstanceFrom* methods return an object. As the preceding code demonstrates, you can cast the generic object to a specific type and invoke the chosen method. This combines late and early binding, which has significant performance benefits when compared with late binding the type and method.

Late Binding Delegates

A *delegate* is a repository of type-safe function pointers. A *single-cast delegate* holds one function pointer, whereas a *multicast delegate* is a basket of one or more delegates. Delegates are type-safe because the signatures of the delegate and function pointer must match. Normally a compile error is generated if there is a mismatch. Unlike *MethodInfo*, a function pointer is discriminatory and bound to a specific object. *MethodInfo* is nondiscriminatory and can be associated with *any* affiliated object. This is the reason why *MethodInfo.Invoke* and *Type.InvokeMember* methods have an object parameter to associate an instance with the target method. This section assumes a fundamental understanding of delegates. If you would like a review, take a look at Chapter 8, “Delegates and Events.” The following code is typical of delegates:

```

using System;
using System.Reflection;

namespace Donis.CSharpBook{
    delegate void xDelegate(int arga, int argb);
}

```

```

class ZClass {
    public void MethodA(int arga, int argb) {
        Console.WriteLine("ZClass.MethodA called: {0} {1}", arga, argb);
    }
}

class Starter{
    static void Main(){
        ZClass obj=new ZClass();
        XDelegate delObj=new XDelegate(obj.MethodA);
        delObj.Invoke(1,2);
        delObj(3,4);
    }
}

```

In this code, *XDelegate* is the delegate type. *MethodA* is then added to the delegate and invoked. First, invoke using the *Delegate.Invoke* method. Second, invoke the function through the delegate using the C# syntax. At compile time, *XDelegate* expands into a class derived from a *Delegate* type. The *Invoke* method is the most important member implemented and added to the class interface of the derived type. The signature of *Invoke* matches the signature of the delegate. Therefore, *XDelegate.Invoke* has two integer parameters and enforces type safeness of related function pointers.

The preceding code assumes the delegate signature is known at compile time. What if the signature of the delegate is not known at compile time? Because a delegate is an implied class that is similar to any class, you can create an instance of a delegate at run time, bind a method to the delegate, and invoke the method. Instead of invoking a member method, a function pointer is bound and executed against the delegate. The *Delegate.CreateDelegate* and *Delegate.DynamicInvoke* methods provide this behavior. Late binding is not as type-safe as compile-time type checking. (This also pertains to late binding of delegates.) Care must be taken to avoid run-time exceptions. As always, the seminal benefit of late binding is additional flexibility, but performance might suffer.

CreateDelegate constructs a new delegate at run time and then assigns a function pointer to the newly invented delegate. *CreateDelegate* is an overloaded method where the essential parameters are the delegate type and method identity. The *delegateType* is the type of delegate being created. *Method* is the initial function pointer being assigned to the delegate. The signature of the method represented by a *MethodInfo* object should match that of the delegate type. These are the overloaded methods:

The *Delegate.CreateDelegate* syntax is as follows:

```

static Delegate CreateDelegate(Type type, MethodInfo method)
static Delegate CreateDelegate(Type type, MethodInfo method, bool thrownOnBindFailure)
static Delegate CreateDelegate(Type type, object firstArgument, MethodInfo method)
static Delegate CreateDelegate(Type type, object target, string method)
static Delegate CreateDelegate(Type type, Type target, string method)

```

```

static Delegate CreateDelegate(Type type, object firstArgument, MethodInfo method, bool
throwOnBindFailure)
static Delegate CreateDelegate(Type type, object target, string method,
    bool ignoreCase)
static Delegate CreateDelegate(Type type, object firstArgument, Type target, string method)
static Delegate CreateDelegate(Type type, Type target, string method, bool ignoreCase)
static Delegate CreateDelegate(Type type, object target, string method, bool ignoreCase,
bool throwOnBindFailure)
static Delegate.CreateDelegate(Type type, object firstArgument, Type target, string method,
bool ignoreCase)
static Delegate.CreateDelegate(Type type, Type target, string method, bool ignoreCase, bool
throwOnBindFailure)
static Delegate.CreateDelegate(Type type, object firstArgument, Type target, string method,
bool ignoreCase, bool throwOnBindFailure)

```

After creating a delegate at run time, call *DynamicInvoke* to invoke any function pointers assigned to the delegate. The dynamically created delegate is deprived of the *Invoke* method and language-specific operations. Subsequently, you cannot call the *Invoke* method on a delegate returned from *CreateDelegate*. This is the major difference between compile-time and run-time instances of delegates. An array of function arguments is the only parameter of *DynamicInvoke*.

This is the *DynamicInvoke* syntax:

```
object DynamicInvoke(object [] args)
```

CreateDelegate and *DynamicInvoke* are demonstrated in the following code:

```

using System;
using System.Reflection;

namespace Donis.CSharpBook{
    delegate void theDelegate(int arga, int argb);

    class ZClass {
        public void MethodA(int arga, int argb) {
            Console.WriteLine("ZClass.MethodA called: {0} {1}", arga, argb);
        }
    }

    class Starter{
        static void Main(){
            Type tObj=typeof(System.MulticastDelegate);
            ZClass obj=new ZClass();
            Delegate del=Delegate.CreateDelegate(typeof(theDelegate), obj,
                "MethodA");
            del.DynamicInvoke(new object [] {1,2});
        }
    }
}

```

Function Call Performance

Several ways to invoke a method have been presented in this chapter—from a simple method call to the more complex dynamic invocation. Performance is an important criterion when evaluating competing methodologies. For example, a simple call bound at compile time should be quicker than a method bound at run time. Depending on the application, the differentiation might be material. Losing a few nanoseconds occasionally might not be conspicuous in a user interface–driven application. However, a few lost nanoseconds in a server application multiplied by thousands of users can pose a real problem.

Reflection and Generics

In .NET Framework 2.0, *Reflection* is extended to accommodate open and closed generic types and methods. Predictably, the *Type* class is the focal point of changes to accommodate generic types. For generic methods, *MethodInfo* has been enhanced to reflect generic methods. (Generics were introduced in Chapter 6.) Open types are generic types with unbound type parameters, whereas closed types have bound type parameters. With *Reflection*, you can browse bound and unbound parameters, create instances of generic types, and invoke generic methods at run time.

IsGeneric and IsGenericTypeDefinition

With *Reflection*, you can query the state of a generic type or method. Is this a generic type or method? If confirmed, is the generic type or method open or closed? *Type.IsGeneric* is a Boolean property that confirms the presence of a generic type; *Type.IsGenericTypeDefinition*, another Boolean property, indicates whether the generic is open or closed. For methods, the *MethodInfo.IsGenericMethod* property confirms that a method is a generic method, whereas the *MethodInfo.IsGenericTypeDefinition* property indicates whether the generic method is open or closed. This program demonstrates the four properties:

```
using System;
using System.Reflection;

public class ZClass<T, V> {
    public T memberA;
}

public class XClass {
    public void MethodA<T>(C) {
    }
}

namespace Donis.CSharpBook{

    class Starter{
```

```

static void Main(){
    Type [] types={typeof(ZClass<,>), typeof(ZClass<int,int>)};
    bool [,] bresp= { {types[0].IsGenericType,
                      types[0].IsGenericTypeDefinition},
                    {types[1].IsGenericType,
                      types[1].IsGenericTypeDefinition}};
    Console.WriteLine("Is ZClass<,> a generic type? "+bresp[0,0]);
    Console.WriteLine("Is ZClass<,> open? "
        +bresp[0,1]);
    Console.WriteLine("Is ZClass<int,int> a generic type? "
        +bresp[1,0]);
    Console.WriteLine("Is ZClass<int,int> open? "+bresp[1,1]);

    Type tObj=typeof(XClass);
    MethodInfo method=tObj.GetMethod("MethodA");
    bool [] bMethod={method.IsGenericMethod,
                    method.IsGenericMethodDefinition};
    Console.WriteLine("Is XClass.MethodA<T> a generic method? "
        +bMethod[0]);
    Console.WriteLine("Is XClass.MethodA<T> open? "+bMethod[1]);
    }
}

```

typeof

The *typeof* operator is used in the preceding code to extract the *Type* object of an open or constructed type. The *Type* object has a single parameter, which is a string and identifies the generic type. Connote an open type using empty parameters. For example, *ZClass<T>* would be *ZClass<>*. Multiple generic type parameters are indicated with n-1 commas. For the *typeof* operator, *ZClass<,>* is compatible with *ZClass<K,V>*. Indicate a closed type by including the actual parameter types, such as *ZClass<int, int>*. In addition to the *typeof* operator, *Type.GetType* and *Type.GetGenericTypeDefinition* methods return *Type* objects representing generic types.

GetType

Type.GetType is available in two flavors: an instance and a static method. *Type.GetType* is an instance method and returns the open type used to construct the object. The method has no parameters. The *Type.GetType* static method is overloaded to return a *Type* object for either an open or a closed type. The pivotal parameter of the static *GetType* method is a string naming the generic type. To specify an open type, the string is the name of the generic with the number of parameters affixed. The suffix is preceded with a “`”. For example, “NamespaceA.XClass`2” would represent *XClass<K,V>*. *XClass* has two type parameters. For a closed type, you need to add the bound type parameters. After the number of type parameters, list the bound type parameters. The bound type parameters are contained in square brackets.

“NamespaceB.ZClass`3[System.Int32, System.Int32, System.Decimal]” identifies *ZClass* <*int*, *int*, *decimal*>. This is the general format:

Open type: `GenericType`NumberOfParameters`
 Closed Type: `GenericType`NumofParameters[parameter list]`

The following sample code demonstrates both the instance and static *GetType* methods:

```
using System;

namespace Donis.CSharpBook {

    class ZClass<K,V> {
        public void FunctionA(K argk, V argv) {
        }
    }

    class XClass<T> {
        public void FunctionB(T argt) {
        }
    }

    class Starter {

        public static void Main() {
            ZClass<int, decimal> obj=new ZClass<int, decimal>();
            Type typeClosed=obj.GetType();
            Console.WriteLine(typeClosed.ToString());

            Type typeOpen=Type.GetType("Donis.CSharpBook.XClass`1");
            Console.WriteLine(typeOpen.ToString());
            Type typeClosed2=Type.GetType(
                "Donis.CSharpBook.ZClass`2[System.Int32, System.Decimal]");
            Console.WriteLine(typeClosed2.ToString());
        }
    }
}
```

GetGenericTypeDefinition

Closed types are constructed from open types. *Type.GetGenericTypeDefinition* returns the *Type* object of the open type used to construct a closed type.

This is the *Type.GetGenericTypeDefinition* syntax:

```
Type GetGenericTypeDefinition()
```

The following code highlights the *GetGenericTypeDefinition* method:

```
using System;

namespace Donis.CSharpBook {
```

```

class ZClass<K,V> {
    public void FunctionA(K argk, V argv) {
    }
}

class Starter {

    public static void Main() {
        ZClass<int, decimal> obj=new ZClass<int, decimal>();
        ZClass<string, float> obj2=new ZClass<string, float>();

        Type closedType=obj.GetType();
        Type openType=closedType.GetGenericTypeDefinition();

        Type closedType2=obj2.GetType();
        Type openType2=closedType2.GetGenericTypeDefinition();

        Console.WriteLine(openType.ToString());
        Console.WriteLine(openType2.ToString());
    }
}

```

The preceding code displays identical strings. Why? The open type of `ZClass<int, decimal>` and `ZClass<string, float>` is the same, which is `ZClass<K, V>`.

`Type.GetMethod` and `MethodInfo.GetGenericMethodDefinition` are comparable to `Type.GetType` and `Type.GetGenericTypeDefinition`, respectively, but pertain to generic methods.

GetGenericArguments

You can now extract a `Type` object for a generic type and a `MethodInfo` object for a generic method. Determining the number of unbound or bound parameters is a natural next step. If bound, the type of each parameter would be useful. `GetGenericArguments` is the universal method for enumerating parameters of generic type or method. `GetGenericArguments` enumerates unbound and bound parameters.

This is the `Type.GetGenericArguments` syntax:

```

Type [] GetGenericArguments()
MethodInfo.GetGenericArguments syntax:
Type [] GetGenericArguments()

```

The following code demonstrates both `Type.GetGenericArguments` and `MethodInfo.GetGenericArguments`:

```

using System;

namespace Donis.CSharpBook {

    class ZClass<K,V> {
        public void FunctionA(K argk, V argv) {

```

```

    }
}

class Starter {

    public static void Main() {
        ZClass<int, decimal> obj=new ZClass<int, decimal>();
        ZClass<string, float> obj2=new ZClass<string, float>();

        Type closedType=obj.GetType();
        Type openType=closedType.GetGenericTypeDefinition();

        Type closedType2=obj2.GetType();
        Type openType2=closedType2.GetGenericTypeDefinition();

        Console.WriteLine(openType.ToString());
        Console.WriteLine(openType2.ToString());
    }
}
}

```

Creating Generic Types

Generic types can be created at run time using *Reflection*. First, extract the open type of the generic. This chapter has shown several ways to accomplish this, including using *GetType* and *GetGenericTypeDefinition*. Next, bind the type arguments of the open type. The result will be a bound (closed) type. Finally, create an instance of the bound type in the customary manner. (*Activator.CreateInstance* works well.)

The *Type.MakeGenericType* method binds type parameters to an open type of a generic. *MakeGenericType* has a single parameter, which is an array of *Type* objects. Each element of the array matches a specific type to a generic parameter. If the generic has three parameters, the array passed to *MakeGenericType* will have three elements—each representing a bound parameter.

This is the prototype of the *MakeGenericType* method:

```

Type.MakeGenericType syntax:
void MakeGenericType(Type [] genericArguments)

```

Generic methods, like nongeneric methods, can be invoked dynamically at run time. *MethodInfo.MakeGenericType* binds parameters to generic types, similar to *Type.MakeGenericType*. The benefits and pitfalls of dynamic invocation are similar to nongeneric methods. After binding parameters to a generic method, the method can be invoked using *Reflection*. The following code creates a generic type and then invokes a generic method at run time:

```

using System;
using System.Reflection;

namespace Donis.CSharpBook{

```

```

public class GenericZ <K, V, Z>
    where K: new()
    where V: new() {

    public void MethodA<A>(A argument1, Z argument2) {
        Console.WriteLine("MethodA invoked");
    }

    private K field1=new K();
    private V field2=new V();
}

class Starter{

    static void Main(){
        Type genericType=typeof(GenericZ<,,>);
        Type [] parameters={typeof(int),typeof(float),
            typeof(int)};
        Type closedType=genericType.MakeGenericType(parameters);
        MethodInfo openMethod=closedType.GetMethod("MethodA");
        object newObject = Activator.CreateInstance(closedType);
        parameters=new Type [] {typeof(int)};
        MethodInfo closedMethod=
            openMethod.MakeGenericMethod(parameters);
        object[] methodargs={2, 10};
        closedMethod.Invoke(newObject, methodargs);
    }
}
}

```

Reflection Security

Some reflection operations, such as accessing protected and private members, require *ReflectionPermission* security permission. *ReflectionPermission* is typically granted to local and intranet applications, but not to Internet applications. Set the appropriate *ReflectionPermission* flag to grant or deny specific reflection operations.

Table 10-9 lists the *ReflectionPermission* flags.

Table 10-9 *ReflectionPermission* Flags

Flag	Description
<i>MemberAccess</i>	Reflection of nonvisible members granted.
<i>NoFlags</i>	Reflection denied to nonvisible types.
<i>ReflectionEmit</i>	<i>System.Reflection.Emit</i> operations granted.
<i>TypeInformation</i>	This flag is now deprecated.
<i>AllFlags</i>	Combines <i>TypeInformation</i> , <i>MemberAccess</i> , and <i>ReflectionEmit</i> flags.

Attributes

Attributes extend the description of some elements of an assembly. Attributes fulfill the role of adjectives in .NET and annotate assemblies, classes, methods, parameters, and other elements of an assembly. Attributes are commonplace in .NET and fulfill many roles: They delineate serialization, stipulate import linkage, set class layout, indicate conditional compilation, mark a method as deprecated, and much more.

Attributes extend the metadata of the targeted element. An instance of the attribute is stored alongside the metadata and sometimes the MSIL code of the constituent. This is demonstrated in the following code and in Figure 10-4. The *Conditional* attribute marks a method for conditional compilation. Use this attribute to flag one or more symbols. If the symbol is defined, the target method and call sites are included in the compiled applications. If not defined, the method and any invocation are ignored.

```
#define LOG

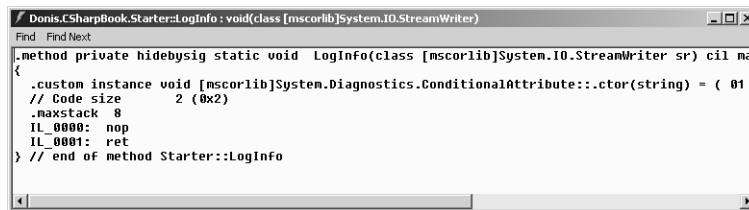
using System;
using System.IO;
using System.Diagnostics;

namespace Donis.CSharpBook{

    class Starter{
        static void Main(){
            LogInfo(new StreamWriter(@"c:\logfile.txt"));
        }

        [Conditional("LOG")]
        private static void LogInfo(StreamWriter sr) {
            // write information to log file
        }
    }
}
```

This is the MSIL code of the *LogInfo* method as displayed in ILDASM. Notice the custom directive that defines the *Conditional* attribute. It is integrated into the MSIL code of the method (see Figure 10-4).



```
Donis.CSharpBook.Starter::LogInfo : void(class [mscorlib]System.IO.StreamWriter)
Find Find Next
.method private hidebysig static void LogInfo(class [mscorlib]System.IO.StreamWriter sr) cil na
{
    .custom instance void [mscorlib]System.Diagnostics.ConditionalAttribute::.ctor(string) = ( 01
    // Code size
    // Code size      2 (0x2)
    .maxstack 8
    IL_0000: nop
    IL_0001: ret
} // end of method Starter::LogInfo
```

Figure 10-4 The *Conditional* attribute in MSIL code

Attributes are available in different flavors: predefined custom attributes, programmer-defined custom attributes, and pseudo-custom attributes.

Predefined Attributes

Predefined custom attributes, which are defined in the .NET FCL, are the most prevalent custom attributes. There is an encyclopedic list of predefined custom attributes fulfilling a variety of responsibilities in .NET. A short list of predefined custom attributes includes *Assembly-Version*, *Debuggergable*, *FileIOPermission*, *Flags*, *Obsolete*, and *Serializable*. The following code uses the *Obsolete* attribute to flag a method as deprecated:

```
[Obsolete("Deprecated Method", false)]
public static void MethodA() {
    Console.WriteLine("Starter.MethodA");
}
```

Combining Attributes

An entity can be assigned multiple attributes. In certain circumstances, even the same attribute can be applied multiple times. There are two ways to combine attributes: They can be grouped together or listed separately. Here, the attributes are applied separately:

```
[Obsolete("Deprecated Method", false)]
[FileIOPermission(SecurityAction.Demand,
    Unrestricted=true)]
public static void MethodA() {
    Console.WriteLine("Starter.MethodA");
}
```

In the following code, two attributes are combined and applied to a method:

```
[Obsolete("Deprecated Method", false),
    FileIOPermission(SecurityAction.Demand,
        Unrestricted=true)]
public static void MethodA() {
    Console.WriteLine("Starter.MethodA");
}
```

Programmer-defined Custom Attributes

You can also create programmer-defined custom attributes to personally extend metadata of an application. There are few limitations to a programmer-defined custom attribute: Your attributes can be applied to any entity, except another programmer-defined custom attribute, and can be designed for almost any purpose.

Pseudo-custom Attributes

Pseudo-custom attributes are interpreted by the run time and modify the metadata of the assembly. Unlike predefined or custom attributes, pseudo-custom attributes do not extend metadata. Examples of pseudo-custom attributes include *DllImport*, *MarshalAs*, and *Serializable*.

Anatomy of an Attribute

What exactly is an attribute? Attributes are derived from the *System.Attribute* class, which is the common template of all attributes. *System.Attribute* is an abstract class defining the intrinsic services of an attribute. Manage compilers and the run time often accord special treatment to attributes. For example, the *ObsoleteAttribute* causes the compiler to generate error or warning messages when a deprecated member is used.

This is the syntax of an attribute:

```
[target type: attribute name(positional parameter1, positional parametern, named parameter1=value, named parametern=value)] target
```

The target type designates the type of the target and must be included in the defined attribute usages. Attribute target is optional and usually inferred correctly if not omitted.

Here is a list of the valid target types:

- field
- event
- method
- *Param*
- *Property*
- *Return*
- *Type*

The attribute name is the class name of the attribute. By convention, attribute names have an *Attribute* suffix. *OneWayAttribute* is representative of an attribute name. You can also use the alias of an attribute, which omits the *Attribute* suffix. The alias for *OneWayAttribute* is *OneWay*.

Attributes accept zero or more positional parameters. If present, position parameters are not optional and must be listed in a declared sequence. In addition, an attribute can offer any number of named parameters, which must follow positional parameters and are optional. Named parameters are not ordered and can be presented in any sequence.

The following code applies the *UIPermissionAttribute* to the *Starter* class:

```
[type: UIPermissionAttribute(SecurityAction.Demand,  
    Clipboard=UIPermissionClipboard.OwnClipboard)]  
class Starter{  
    static void Main(){  
  
    }  
}
```

This is the same attribute expressed somewhat more succinctly:

```
[UIPermission(SecurityAction.Demand)]
class Starter{
    static void Main(){

    }
}
```

Creating a Custom Attribute

You can create custom attributes for personal consumption or to be published in a library for others. There are definitive procedures to creating a programmer-defined custom attribute. Here are the steps to follow:

1. Select an appropriate name for the custom attribute. As mentioned, attribute names should conclude with the *Attribute* suffix.
2. Define an attribute class that derives from *System.Attribute*.
3. Set potential targets with the *AttributeUsage* attribute.
4. Implement class constructors, which determine the positional parameters.
5. Implement write-accessible properties, which define the named parameters.
6. Implement other members of the attribute class.

ClassVersionAttribute is a programmer-defined custom attribute that can apply a version number to types. It illuminates the steps to creating a custom attribute. The attribute assigns a target and current version number to a type. The target version is the version number of the type that the attribute adorns. The current version number is the version applied to the most recent version of the type. When the target type of the attribute is the most recent version, the target and current version are identical. In addition, the *ClassVersionAttribute* can request that future instances of the target type be replaced with the current type.

The first and most important task of creating a programmer-defined attribute is selecting a fabulous name. Alas, *ClassVersionAttribute* is a mundane but descriptive name. As an attribute, *ClassVersionAttribute* must inherit *System.Attribute*. *AttributeUsageAttribute* sets the potential target of the attributes. *AttributeTargets* is the only positional parameter of the *AttributeUsageAttribute* and is a bitwise enumeration. *AttributeUsageAttribute* has three named parameters: *AllowMultiple*, *Inherited*, and *ValidOn*. *AllowMultiple* is a Boolean flag. When true, the attribute can be applied multiple times to the same target. The default is false. *Inherited* is another Boolean flag. If true, which is the default, the attribute is inheritable from base classes. *ValidOn* is an *AttributeTargets* enumeration. This is an alternate method of setting the potential target of the attribute.

Here is a list of the available *AttributeTargets* flags:

- *All*
- *Assembly*
- *Class*
- *Constructor*
- *Delegate*
- *Enum*
- *Event*
- *Field*
- *GenericParameter*
- *Interface*
- *Method*
- *Module*
- *Parameter*
- *Property*
- *ReturnValue*
- *Struct*

This is the start of the *ClassVersionAttribute* class:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct,  
    Inherited=false)]  
public class ClassVersionAttribute: System.Attribute {  
}
```

Instance constructors of an attribute provide the positional parameters for the attribute. Attributes are created at compile time, upon reflection, and when applied at run time. When attributes are created, the selected constructor runs. The positional arguments should match the signature of a constructor in the attribute. Overloading the constructor allows different sets of positional parameters. Positional and named parameters are restricted to certain types. The following is a list of available attribute parameter types.

- *bool*
- *byte*
- *char*
- *double*

- *float*
- *int*
- *long*
- *sbyte*
- *short*
- *string*
- *uint*
- *ulong*
- *ushort*

ClassVersionAttribute has two overloaded constructors. The first constructor accepts the target version; the second constructor accepts the target and current version number:

```
public ClassVersionAttribute(string target)
    :this(target, target) {
}

public ClassVersionAttribute(string target,
    string current) {
    m_TargetVersion=target;
    m_CurrentVersion=current;
}
```

Define named parameters as write-only or read-write instance properties of the derived attribute class. You can duplicate positional parameters as named parameters for additional flexibility. *ClassVersionAttributes* offers a *UseCurrentVersion* and *CurrentName* named parameter. *UseCurrentVersion* is a Boolean value. If true, instances of the current type should be substituted for the target type. *CurrentName* names the type of the current version. This is how the named parameters are described in the *ClassVersionAttribute* class:

```
private bool m_UseCurrentVersion=false;
public bool UseCurrentVersion {
    set {
        if(m_TargetVersion != m_CurrentVersion) {
            m_UseCurrentVersion=value;
        }
    }
    get {
        return m_UseCurrentVersion;
    }
}

private string m_CurrentName;
public string CurrentName {
    set {
        m_CurrentName=value;
    }
}
```

```

    get {
        return m_CurrentName;
    }
}

```

For completeness, read-only properties are added to the *ClassVersionAttribute* class for the target and current version. Here is the completed *ClassVersionAttribute* class:

```

using System;

namespace Donis.CSharpBook{
    [AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct,
        Inherited=false)]
    public class ClassVersionAttribute: System.Attribute {

        public ClassVersionAttribute(string target)
            :this(target, target) {
        }

        public ClassVersionAttribute(string target,
            string current) {
            m_TargetVersion=target;
            m_CurrentVersion=current;
        }

        private bool m_UseCurrentVersion=false;
        public bool UseCurrentVersion {
            set {
                if(m_TargetVersion != m_CurrentVersion) {
                    m_UseCurrentVersion=value;
                }
            }
            get {
                return m_UseCurrentVersion;
            }
        }

        private string m_CurrentName;
        public string CurrentName {
            set {
                m_CurrentName=value;
            }
            get {
                return m_CurrentName;
            }
        }

        private string m_TargetVersion;
        public string TargetVersion {
            get {
                return m_TargetVersion;
            }
        }
    }
}

```

```

    private string m_CurrentVersion;
    public string CurrentVersion {
        get {
            return m_CurrentVersion;
        }
    }
}
}
}

```

The *ClassVersionAttribute* class is compiled and published in a DLL. The following application minimally uses the *ClassVersionAttribute*:

```

using System;
namespace Donis.CSharpBook{
    class Starter{
        static void Main(){

        }
    }

    [ClassVersion("1.1.2.1", UseCurrentVersion=false)]
    class ZClass {
    }
}

```

Attributes and Reflection

Programmer-defined custom attributes are valuable as information. However, the real fun and power is in associating a behavior with an attribute. You can read custom attributes with *Reflection* using the *Attribute.GetCustomAttribute* and *Type.GetCustomAttributes* methods. Both methods return an instance or instances of an attribute. You can downcast the attribute instance to a specific attribute type and leverage the intricacies of the attribute to implement the appropriate behavior.

Type.GetCustomAttributes returns attributes applied to a type. *GetCustomAttributes* is also available with other metadata elements, including *Assembly.GetCustomAttributes*, *MemberInfo.GetCustomAttributes*, and *ParameterInfo.GetCustomAttributes*. *GetCustomAttributes* has a single Boolean parameter. If true, the inheritance hierarchy of the type is evaluated for additional attributes.

This is the *Type.GetCustomAttributes* method:

```

object [] GetCustomAttributes(bool inherit)
object [] GetCustomAttribute(type AttributeType, bool inherit)

```

Attribute.GetCustomAttribute, which is a static method, creates and returns an instance of a specific attribute. When a specific attribute is desired, *GetCustomAttribute* is invaluable and more efficient than *GetCustomAttributes*. *GetCustomAttribute* assumes that the attribute is assigned only once to the target. *GetCustomAttribute* is overloaded for each potential target,

with one notable exception: *GetCustomAttribute* is not overloaded for a type target, which is inexplicable. *Type.GetCustomAttributes* is the sole option for enumerating attributes of a *Type*. *GetCustomAttribute* is overloaded for two and three arguments. The two-argument versions have the target and attribute type as parameters. The three-argument version has an additional parameter, which is the inherit parameter. If the inherit parameter is true, the ascendants of the target class are also searched for the attribute.

This is the *Attribute.GetCustomAttribute* method:

```
static Attribute GetCustomAttribute(Assembly targetAssembly, type attributeType)
static Attribute GetCustomAttribute(MemberInfo targetMember, type attributeType)
static Attribute GetCustomAttribute(Module targetModule, type attributeType)
static Attribute GetCustomAttribute(ParameterInfo targetParameter, type attributeType)
```

This following sample code uses *GetCustomAttribute*:

```
using System;
using System.Reflection;

namespace Donis.CSharpBook{

    class Starter{

        static void Main(){
            Type tObj=typeof(Starter);
            MethodInfo method=tObj.GetMethod("AMethod");
            Attribute attrib=Attribute.GetCustomAttribute(
                method, typeof(ObsoleteAttribute));
            ObsoleteAttribute obsolete=(ObsoleteAttribute) attrib;
            Console.WriteLine("Obsolete Message: "+obsolete.Message);
        }

        [Obsolete("Deprecated function.", false)]
        public void AMethod() {
        }
    }
}
```

The following is sample code of *GetCustomAttributes*. The application inspects the *ClassVersionAttribute* attribute with the *GetCustomAttributes* method and acts upon the results. The application contains two versions of *ZClass* and both are applied to the *ClassVersionAttribute*. Each version of *ZClass* shares a common interface: *IZClass*. The *ClassVersionAttribute* of *Donis.CSharpBook.ZClass* lists *ANamespace.ZClass* as the current version or most recent version. Also, *UseCurrentVersion* is assigned *true* to indicate that the newer version should replace instances of *Donis.CSharpBook.ZClass*. The function *CreateZClass* accepts a *ZClass* as defined by the *IZClass* interface. Within the *foreach* loop, *GetCustomAttributes* enumerates the attributes of the *ZClass*. If the attribute is a *ClassVersionAttribute*, the attribute is saved and the *foreach* loop is exited. Next, the properties of the attribute are examined. If *UseCurrentVersion*

is true and a current version is named, an instance of the new version is created and returned from the method. Otherwise, the target version is returned.

```
using System;
using System.Reflection;

interface IZClass {
    void AMethod();
}

namespace Donis.CSharpBook{
    class Starter{
        static void Main(){
            IZClass obj=CreateZClass(typeof(ZClass));
            obj.AMethod();
        }

        private static IZClass CreateZClass(Type tObj) {
            ClassVersionAttribute classversion=null;
            foreach(Attribute attrib in tObj.GetCustomAttributes(false)) {
                if(attrib.ToString()==
                    typeof(ClassVersionAttribute).ToString()) {
                    classversion=(ClassVersionAttribute) attrib;
                }
                else {
                    return null;
                }
            }
            if(classversion.UseCurrentVersion &&
                (classversion.CurrentName != null)) {
                AppDomain currDomain=AppDomain.CurrentDomain;
                return (IZClass) currDomain.CreateInstanceFromAndUnwrap(
                    "client.exe", classversion.CurrentName);
            }
            else {
                return (IZClass) Activator.CreateInstance(tObj);
            }
        }
    }

    [ClassVersion("1.1.2.1", "2.0.0.0", UseCurrentVersion=true,
        CurrentName="Donis.CSharpBook.ANamespace.ZClass")]
    public class ZClass: IZClass {
        public void AMethod() {
            Console.WriteLine("AMethod: old version");
        }
    }
}
```

```
namespace ANamespace {
    [ClassVersion("2.0.0.0", UseCurrentVersion=false)]
    public class ZClass: IZClass {
        public void AMethod() {
            Console.WriteLine("AMethod: new version");
        }
    }
}
```

MSIL

Metadata and the ability to inspect metadata using *Reflection* were the topics of this chapter. .NET assemblies consist of metadata and MSIL, in which understanding MSIL is equally important to a developer. (MSIL is the topic of Chapter 11.) Understanding MSIL is important for writing, maintaining, and, later, debugging a C# application. *Reflection* can be used with MSIL code. You can inspect MSIL code at run time, create self-generating code, and otherwise manipulate MSIL using *Reflection*.

