Before we take a hands-on look at Microsoft’s implementation of the Common Information Model (CIM) repository, let us first examine its structure as defined by the DMTF. In the previous chapters, you have gleaned an idea of the structure and purpose of the CIM. In simple terms, it consists of the following four components:

- Classes that define the structure of the information held in the store
- Associations that define the relationships between elements in the repository
- Methods that define the behavior of the managed objects
- Properties that define individual characteristics of each managed object

Each of these components can have associated instances that hold the management data. They were designed with the basic purpose of providing a platform-independent means of describing the manageable aspects of all types of logical and physical components in the enterprise.

The objective of this chapter is not to describe every class, property, method, and association within the CIM schema and Win32 extended schema. First, this would not make particularly interesting reading, and second, it would prove somewhat overwhelming as an introduction to the CIM. Instead, the focus of this chapter is to examine the classes and associations that hold the elements of most interest to users of WMI. This includes the WMI system classes that define the properties and methods relevant to WMI’s security model, the classes that define the WMI provider architecture, and the classes of the CIM schema. By the end of the chapter, you should have a solid enough knowledge of the structure of the repository to move through it with confidence.

We shall use Microsoft’s WMI CIM Studio to look at the classes within the CIM repository. During the course of the chapter, we also shall look briefly at the Managed Object Format (MOF) so that you can better understand some of the terminology used in the WMI CIM Studio. For a more complete list of the syntax and semantics used in the MOF, you can refer to the platform SDK documentation available from the MSDN Web site.

Recall from the previous chapter that the CIM is a conceptual model for storing enterprise management information—it is not a physical implementation (that is, the DMTF does not supply a downloadable version of WBEM that you can install as part of your operating system). It is the responsibility of vendors such as Microsoft, Intel, or Hewlett-Packard to adhere to the DMTF’s CIM standard and produce their own physical implementations. Microsoft has done this for its range of Windows operating systems.
The CIM repository supplied as part of WMI is a data store structured in accordance with the DMTF’s CIM. It is supplied with any of Microsoft’s WMI-compliant operating systems (Windows 9x/2000/NT4.0/WinXp/.NET): It holds management information structured in accordance with the CIM.

The aim of this chapter is to familiarize you with the CIM repository that is shipped with WMI. This will serve two purposes: first, it will give you some idea of the vast array of information that you can retrieve from the repository, and second, it will provide you with an introduction to the structure and relationships of the existing classes. This knowledge will prove useful especially if you are faced with the task of instrumenting your own product within the CIM repository or need to elicit management information from the repository about managed objects. To explore the repository, we shall use the tools supplied with the WMI SDK, so if you did not install the SDK during your study of Chapter 3, you should do so now.

### Metadata

“Metadata” is a new term to those who are not familiar with data modeling. The Greek word “meta-” is a prefix that means “behind” or “hidden.” If you are familiar with HTML scripting, you will have encountered the prefix in metatags. Metatags describe the characteristics of information presented on a Web page (for example, to specify text displaying **Bold** or in *Italics*). Metadata is “data that describes data” or, more precisely, *definitional data*.

Admittedly, this is a broad definition, so let us define metadata more fully in terms of the CIM. We know from previous chapters that the information available from CIM is described by a series of classes and associations, and the elements contained therein (methods, properties, and references). These constructs describe the data available to WMI client applications and are classified as metadata, as you can see in Table 4.1.

The information in the Property and Type columns is metadata, as it describes the information we can retrieve, which is listed in the Value column. The Property and Type columns represent “data about data.” The information in the Value column is instance data—the data that metadata describe. Metadata is a generic term and applies to

<table>
<thead>
<tr>
<th>Property (metadata)</th>
<th>Type (metadata)</th>
<th>Value (instance data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>String</td>
<td>CD99A_EXPRESS</td>
</tr>
<tr>
<td>Version</td>
<td>Real32</td>
<td>1.64</td>
</tr>
<tr>
<td>BIOSVersion</td>
<td>String</td>
<td>1.23.54.244</td>
</tr>
<tr>
<td>Read_errors</td>
<td>Sint32</td>
<td>0</td>
</tr>
</tbody>
</table>
a broad spectrum of data; for example, class information is metadata. Metadata also includes descriptive information about the context, quality and condition, or characteristics of the data, such as WMI system classes and properties. It is important to make the distinction between metadata and instance data because the CIM repository was envisaged by the DMTF primarily, although not exclusively, as a store for metadata.

### Dynamic and Static Data

As well as understanding the distinction between metadata and instance data, it is important to understand that the data in WMI is categorized further as either dynamic or static. A simple rule of thumb is that metadata is inherently static and instance data more often is dynamic. As with all generalizations, however, there can always be exceptions to the rule.

Dynamic data is characterized by frequent change. Instance data commonly falls into this category, because the values associated with instance data often change during a managed object’s lifetime. For example, the value assigned to a property that represents the amount of free space left on a disk partition is dynamic in nature, because during the day-to-day operation of the disk drive, the value normally will change. Other reasons that the data may change include:

1. The data may persist in other places, such as Active Directory, a database, the Internet, or simply a file on disk.
2. A WMI class can have properties that are retrieved from several sources, thus unifying separate data sources into one logical means of accessing it and its relationships. For example, the Win32_Service might obtain the Started property at run time and most of the other properties from the registry.

Static data is slow or nonchanging data, such as class or association definitions or a static property value such as a BIOS version number (although these values can change if the BIOS is stored on flash memory that can be reprogrammed) or a manufacturer’s name. WMI class definitions often fall into this category because the classes that define a managed object do not change with great frequency.

Dynamic data (for example, the Read_errors value in the Value column in Table 4.1), is provided externally by WMI providers. Static data and metadata are stored in the CIM repository.

### Dynamic Data and the CIM Repository

An important concept when retrieving dynamic (volatile) data is **temporal correctness**. For data to be temporally correct, it must be valid in relation to the time of the original query. When a managed object supports dynamic data, it is counterintuitive to import
that information into the repository with no means of updating it. Imagine that we are responsible for monitoring the amount of free space on the disk drive partition of a network server. Because the network server is heavily utilized, the space remaining on the disk changes constantly during its day-to-day operation. Without a mechanism for keeping track of these changes, management applications that requested information on the amount of free space could end up retrieving out-of-date information. The solution designed by Microsoft is a WMI provider, a software component that reacts to requests for dynamic data with information that is temporally correct. (Chapter 12 covers WMI providers in detail.) Figure 4.1 shows an example architecture without using WMI providers for dynamic data, and Figure 4.2 demonstrates an architecture that uses WMI providers.

Figure 4.2 shows a management application requesting instance or property data that is supplied dynamically. In this example, class and association information is held statically in the CIM repository. The example does not illustrate the surfacing of events or method invocation.

Figure 4.2 also demonstrates how WMI providers retrieve dynamic data directly from the managed object to respond to the request. The Windows Management service then returns all of the requested information to the client application. Although Figure 4.2 shows a preferred arrangement, it does not preclude storing instance information in the CIM repository, which even may be the preferred solution for situations in which the instance information is not expected to change during the lifetime of the managed object (that is, it is static). It is far more likely, however, that some information will change over the course of its lifetime. Therefore, WMI providers must keep track of these changes.

![Figure 4.1](image)  
*Figure 4.1   Example topology without any mechanism to supply dynamic data*
In the current release of WMI, the metadata (that is, the classes and associations) in the CIM repository describes local management information pertaining to the host operating system. In future releases, however, WMI technology may concern itself with distributed applications and management information from a variety of sources. Remember that the CIM repository is capable of storing information other than metadata from any source, local or otherwise, although this is not the preferred arrangement for current products instrumented in WMI.

### The Common Information Model

As discussed in previous chapters, the CIM has three conceptual layers, each of which builds upon the previous one, providing a greater level of detail. Notice in Figure 4.3 that layer 3 is sometimes referred to as the “layer of extensibility.”

The core and common models are referred to collectively as the **CIM Schema**.
The purpose of layering the model is to allow schema designers to view the model as a series of logical progressions from which they can build. The CIM Schema represents the basic classes, properties, methods, and associations of all potentially managed objects within an enterprise.

The Core Model
The core model defines a number of very general classes to describe basic structures and relationships between manageable components in an environment. This information model is independent of any product or implementation.

The Common Model
The common model defines a basic set of classes that represent specific areas of management within the enterprise. The common model consists of a number of models, each of which represents specific management areas. Currently the DMTF defines models that represent the network-, user-, system-, support-, policy-, physical-, metrics-, interoperability-, events-, device-, and application-specific aspects of the enterprise. (A number of submodels extend these models. For more information on the submodels, visit http://www.dmtf.org, the DMTF Web site.) The number of areas represented by classes in the common model are expected to increase over time, the better to represent the variety of advances in technology.

Systems Model
The systems model defines the basic characteristics associated with managed systems. A managed system can be an operating system, a network system, or an application system. It defines the root class CIM_ManagedSystemElement, from which all system classes must derive. Included as part of this model are classes that define services, file systems, threads, processes, and software features.

Networks Model
The networks model defines a series of classes, associations, methods, and properties that represent the features of a network environment. Features that are common to a network environment include protocols, services, and the topology of the network. Within the networks model, a number of submodels define the classes, associations, methods, and properties that are specific to their environment.

Devices Model
The devices model defines the physical and logical components that support the system. Examples of classes defined as part of the devices model include CIM_POTS Modem, CIM_Processor, CIM_Printer, and CIM/DesktopMonitor.
Physical Model
The physical model is not to be confused with the devices model. Whereas the devices model defines a set of classes that support the managed system, the physical model represents the physical environment. You will find that the physical environment is of surprisingly little concern in the managed environment. Under normal circumstances, you will not often need to come into direct contact with a physical element within an environment but will instead interact with its logical counterpart. Any manipulation of physical elements within the environment normally is a result of manipulation of an associated logical object. For example, imagine an Ethernet network card in an ISA slot of a personal computer running Windows XP. If you were to perform some action that resulted in a series of packets being sent out of the card to the network, it you might trigger the Tx (Transmit) LED on the network card to start blinking. This behavior was the result of the packet being sent out of the card—the Tx LED cannot be addressed directly or polled for its status.

Because the physical model represents the physical aspects of a system, the classes it contains will differ considerably, depending upon the host system (consider the differences in architecture of an HP Mainframe and a desktop computer) and will change to represent the advances in technology.

Applications Model
The applications model describes the details required to manage a set of software applications. The diversity of software applications demands that the model must be flexible enough to describe multiplatform or distributes applications. The applications model borrows heavily from the application software life cycle. The application software life cycle describes the various states of a software application from its initial purchase through its execution.

Event Model
The event model describes CIM Indications and how they are used to communicate occurrences of events in the CIM. It also describes the classes that enable clients to subscribe to CIM Indications, including how to specify a desired mode of delivery. The Specification for CIM Operations over HTTP defines the XML encoding for CIM Indications over HTTP.

Policy Model
The policy model enables application developers, policy administrators, and network administrators to represent and manage policy across a variety of technical domains that include security, networking, and system administration.

Support Model
The support model describes the object and transaction models for the exchange of knowledge related to support activities (Solutions) and the processing of Service
Incidents. The object and transaction models are referred to as the Problem Resolution Standard or PRS.

**User Model**
The user model provides a set of relationships between the various representations of users, their credentials, and the managed system elements that represent the resources and resource managers in system user administration. Thus, the CIM user and security models added to the preexisting set of requirements for the introduction of a “top” object class in the CIM core model.

**Metrics Model**
The metrics model defines classes to represent a unit of work and its associated metrics. For example, a print job could be a unit of work and the number of pages to be printed could be the metric.

**Interop Model**
The interop model defines architectures and mechanisms that enable WBEM implementations to interoperate in an open, standard manner, and addresses issues that prevent them from doing so.

In addition to these models is a number of submodels that include:

- **CIM Network Submodel—IPsec Policy Model.** Defines a number of CIM extensions that represent the IP security policy model.
- **CIM Submodel—Storage Model.** Defines the Logical Devices associated with data storage.
- **CIM Device Submodel—Sensor Model.** Defines additional properties and methods for the classes CIM_Sensor and CIM_NumericSensor.
- **CIM Device Submodel—Printer Model.** Describes the management of the functionality and protocols specific to printers.
- **CIM Submodel—Fault Tolerant Model.** Defines a number of fault tolerant extensions to the CIM model.
- **CIM System Submodel—Diagnostic Model.** Describes additional aspects needed for successful diagnostics under multi-tasking operating systems.

**The Extended Schemas**
Vendors who wish to represent the manageable aspects of their hardware or software product as part of the CIM define the extended schemas. Extended schemas typically derive from classes defined as part of the common model, although it is possible to extend from the core model. An example of an extended schema is the mechanism with which operating system vendors can model the manageable aspects of their operating systems in the CIM. Microsoft uses the Win32 extended schema to represent its Windows platform (Windows 98, NT4.0/2000/WinXP/.NET). The management information you re-
retrieve from WMI about the Windows environment typically will be retrieved from the
classes defined as part of the Win32 extended schema, because the majority of CIM
schema classes defined within the CIMV2 namespace are declared as abstract.

When designing the CIM, the DMTF envisaged a number of ways to implement it. Among the proposed methods were:

- An application Data Base Management System (DBMS)
- A series of application objects that represent instances of CIM classes
- A structure to pass the instances of CIM between applications

Even from a brief look at the core, common, and extended schemas, it is clear that
the DMTF's task in designing the CIM was complex. Because of its complexity, navigat-
ing the CIM repository and the query response times on such a large volume of infor-
mation are design concerns. Too much information to process can be as detrimental to
the usability of an information model as not enough information. In addition, a CIM
repository might host multiple managed environments, such as local and nonlocal
devices. How can we rationalize them?

To overcome these problems, the DMTF developed *namespaces*, a mechanism for
partitioning sections of CIM into smaller logical groupings.

### Namespaces

The purpose of a namespace is to group a set of classes and instances that relate to a
particular managed environment logically. The CIMV2 namespace, for example,
groups a set of classes and instances that relate to aspects of the local Windows envi-
ronment. This philosophy would not support defining a network router (its classes,
and so forth) within the CIMV2 namespace. It would be logical instead to define a
router in the CIM repository by creating a new namespace and populating it with the
relevant router classes. To combine instances from both local and nonlocal elements in
the same namespace could lead to confusion, because there is no immediate way to
distinguish between the two. Placing them in separate namespaces further clarifies the
distinction between the nonlocal router's management environment and the local
Windows host. If, however, you were defining the manageable aspects of a local inter-
nal modem, then it would be most likely that you would place your classes in the
CIMV2 namespace. The CIMV2 namespace derives its name from the DMTF's CIM
schema that is the basis of its classes.

### Using the CIMV2 Namespace

Table 4.2 lists an array of hardware- and software-managed objects that you might find
in the enterprise. The purpose of the table is to illustrate which devices most likely
**Table 4.2** Likely Management Object Locations

<table>
<thead>
<tr>
<th>Device</th>
<th>Location</th>
<th>Description</th>
<th>Namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network card</td>
<td>Local</td>
<td>Network card residing on host machine.</td>
<td>CIMV2</td>
</tr>
<tr>
<td>External modem</td>
<td>Local</td>
<td>56K modem attached to COM1.</td>
<td>CIMV2</td>
</tr>
<tr>
<td>Custom designed software application</td>
<td>Local</td>
<td>Simple standalone application that runs on Windows 2000, NT4, and 9x.</td>
<td>CIMV2</td>
</tr>
<tr>
<td>Distributed database application</td>
<td>Distributed, Machine A, n</td>
<td>A distributed database with error-recovery and data integrity/fault tolerance built in. Requires a separate namespace because the CIMV2 namespace does not accommodate distributed applications.</td>
<td></td>
</tr>
<tr>
<td>Fax gateway software application</td>
<td>Local</td>
<td>Located on host machine. Runs under Windows. Allows other users on network to send faxes using host machine's fax card.</td>
<td>CIMV2</td>
</tr>
<tr>
<td>E-mail gateway</td>
<td>Remote</td>
<td>Located remotely on network. Transmits all external email onto the PSTN.</td>
<td></td>
</tr>
<tr>
<td>Network card</td>
<td>Local</td>
<td>Located on host machine. Instrumented using the DMTF's DMI management architecture.</td>
<td></td>
</tr>
<tr>
<td>Smart Array SCSI controller card*</td>
<td>Local</td>
<td>Controls a RAID 5 array of hard disk using the SNMP protocol.</td>
<td></td>
</tr>
</tbody>
</table>

*Products instrumented with either SNMP or DMI can be represented by classes in the CIM repository. However, they cannot be instrumented as part of the CIMV2 namespace. This is because of architectural differences between the two management paradigms that prevent classes from either SNMP or DMI deriving from those in the CIM schema. Instead, they must define classes in a separate namespace that does not derive from the CIM schema, but instead illustrates itself using a custom schema. This dynamic or static information can then be provided to CIM-compliant applications using WMI's built-in SNMP or DMI provider.
should appear in the CIMV2 namespace and which should be placed elsewhere. With a
technology as complex as WMI, there is always a danger of over-simplifying the issues.
For instance, in text we have ignored the fact that a managed object can appear in mul-
tiple namespaces in the same CIM repository, but the table shows when it is and is not
appropriate to use the CIMV2 namespace. The Location field states where the device is
located, either on the local host machine, or on another machine on the network.

The DMTF specifies the following points as valid criteria for defining a namespace:

- To define chunks of management information (objects and associations) to
  limit implementation resource requirements, such as database size
- To define views on the model for applications managing only specific objects,
  such as hubs
- To pre-structure groups of objects for optimized query speed

**WMI Namespaces and the CIM repository**

In a typical WMI installation, you will find a number of namespaces defined in the CIM
repository. The exact number of namespaces will depend largely upon the version of
your operating system and the applications or hardware on your machine. Figure 4.4
shows an example of the namespaces found on an installation of Windows XP.

![Namespace Example](image)

**Figure 4.4** Example of namespaces found in a Windows XP installation
Table 4.3  Windows XP Namespaces

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>The lowest level in the namespace hierarchy.</td>
</tr>
<tr>
<td>CIMV2</td>
<td>Defines the manageable aspects of the host system.</td>
</tr>
<tr>
<td>CIMV2\Applications</td>
<td>Defines miscellaneous application-specific data.</td>
</tr>
<tr>
<td>Cli</td>
<td>Defines the default WMIC aliases (Windows XP only).</td>
</tr>
<tr>
<td>Default</td>
<td>Default namespace (a good place to experiment with your own classes!).</td>
</tr>
<tr>
<td>Directory</td>
<td>Groups directory related namespaces.</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Groups Microsoft technology-specific namespaces, for example, the HomeNet</td>
</tr>
<tr>
<td></td>
<td>subnamespace on Windows XP, which defines the management information for</td>
</tr>
<tr>
<td></td>
<td>home networking.</td>
</tr>
<tr>
<td>MSAPPS10</td>
<td>Defines class information for the Microsoft Office suite of applications.</td>
</tr>
<tr>
<td>NetFrameworkv1</td>
<td>Defines class information for the .NET framework (.NET must be installed on</td>
</tr>
<tr>
<td></td>
<td>the computer for this to be present).</td>
</tr>
<tr>
<td>Perfmonscriptexample</td>
<td>Monitors performance of script-specific information.</td>
</tr>
<tr>
<td>Policy</td>
<td>Defines policy-specific management information.</td>
</tr>
<tr>
<td>RSOP</td>
<td>Defines resultant set of policy security-related classes for centralized</td>
</tr>
<tr>
<td></td>
<td>policy-based administration (Windows XP/.NET only).</td>
</tr>
<tr>
<td>SECURITY</td>
<td>Defines the WMI system security-related management information.</td>
</tr>
<tr>
<td>Subscription</td>
<td>Defines consumer-related classes for triggering scripts in response to an</td>
</tr>
<tr>
<td></td>
<td>event and sending an e-mail (Windows XP/.NET only).</td>
</tr>
<tr>
<td>WMI</td>
<td>Defines management information for the WMI for WDM provider.</td>
</tr>
<tr>
<td>&lt;namespace&gt;\MS_XXX</td>
<td>Defines locale-specific information (e.g., MS_409 is the namespace that</td>
</tr>
<tr>
<td></td>
<td>defines US English locale-specific information).</td>
</tr>
</tbody>
</table>

Table 4.3 provides a quick roundup of the namespaces found in a Windows XP installation.

- **Defining and Using Your Own Namespace**

By using your own namespace, you effectively rid yourself of a number of the constraints of populating a proprietary namespace, such as seeking permission from the owner of the namespace.
You also create potentially more work for yourself because you must define everything from the root up. As we see in Figure 4.5, the CIMV2 namespace contains a collection of classes from the CIM schema as well as from the Win32 extended schema.

If you decide to create a new namespace to define the manageable aspects of your environment, you are not obliged to use or to derive from all (or any) of the classes defined in the CIM Schema or Win32 extended schema. Indeed, as the CIM repository currently stands, a number of namespaces populated by legacy schemas such as the SNMP and DMI exist. Schema designers are required to populate their namespaces only with classes and associations that are pertinent to the managed environment under scrutiny. If, however, you decide not to derive from any of the classes in the CIM schema, then you must question whether the object that you are defining belongs to the enterprise modeled within the CIM. Remember that the CIM schema models the most basic manageable aspects of the enterprise from which you can derive. (If you feel that a vital class that will enable you to instrument your product is missing within the CIM schema, then you are encouraged to e-mail the DMTF at cim@dmtf.org with a change request and open a channel of discussion with them. For the Win32 extended schema, e-mail Microsoft at wmgmts@microsoft.com.)

**Namespaces and Schemas**

Imagine a schema as a collection of classes aimed at instrumenting a particular management environment. The scope of a schema can be as diverse or as specialized as the schema designer sees fit, with the understanding that the objects defined within it are related. The DMTF designed the CIM schema to represent the different manageable
aspects of the enterprise. Within it are numerous models that target specific aspects of that environment. The Network model, as it defines a group of classes and associations that represent the enterprise network managed environment, is an example. The Win32 extended schema then adds a further level of specialization to this by focusing on the Windows platform. Because schemas are diverse, they contain a large amount of information that may or may not be relevant to a particular managed environment. For example, a schema could contain classes that define every type of microprocessor architecture available. In a real machine, however, only a fraction of those classes may be relevant (perhaps only the x86). A namespace, then, represents the real managed environment by hosting only these relevant classes.

A namespace is a logical abstraction of classes, associations, and instances that limits the visibility and scope to suit a particular managed environment, such as the Windows OS. By limiting the scope, we can improve response times, reduce the resource overhead for managing the environment, and rationalize the amount of information presented to the end user. A namespace can contain many different classes from many different schemas.

The classes defined within a schema, such as the Win32 extended schema, can be contained in one or many namespaces of the same CIM repository without fear of conflict. The CIM repository can differentiate between these classes by examining the full object path of the class, which can include both the server and the namespace of the CIM repository.

### Using Existing Schemas

If you use classes defined by the DMTF, Microsoft, or any other schema designer, then you must be careful to adhere to the guidelines and context for their use. Indeed, by using the schemas defined by a third party, you can provide a set of building blocks from which to derive your own classes. If you do not adhere to these rules, there is the risk that the new classes you define may violate the integrity of the schema, thus invalidating the entire class hierarchy. Put simply: your code won’t work. The schema designers normally detail the classes that can have subclasses and/or be instantiated in their documentation. Microsoft provides this information in its WMI SDK and in the MSDN platform SDK documentation. If you are still in doubt, e-mail Microsoft at wmgmts@microsoft.com.

### Subclassing and Instantiating Existing Classes in the CIMV2 Namespace

If your product populates the CIMV2 namespace, you must check with the schema designer for the conditions of its use when subclassing existing classes or creating in-
stances of them. As part of the WMI SDK, Microsoft provides a text file that details which classes may be subclassed or instantiated and which may not. It is vital to check these details before you start designing your classes and deriving them from existing schema.

**Modifying Existing Classes**

If you wish to modify existing classes within the CIM schema or Win32 extended schema, you must obtain permission first from the schema designer. Situations in which you must obtain permission include the addition of a method or property to an existing class or the addition of a completely new class or association. If you decide to populate Microsoft's CIMV2 namespace with your own classes, then you must be careful to obtain permission in case any of your additional classes conflict with the designer's plans for the namespace.

**Managed Object Format**

Beyond the CIM and extended schemas, we look further at the structure of classes, properties, methods, and associations, initially through examples written in the MOF defined by the DMTF. Later in this chapter, we shall use Microsoft's WMI CIM Studio to explore and modify the CIM repository.

We use the MOF language to define object definitions within the CIM in a textual format. As we shall see in the WMI CIM Studio, MOF is not the only way to modify the CIM repository, but it is beneficial to have some knowledge of the MOF syntax and its semantics. MOF provides a basic understanding of the terminology you will encounter when you use Microsoft's WMI CIM Studio to move through the CIM repository. In addition, it provides you with a concept of the structures that make up the CIM.

MOF files can populate technologies that implement the CIM. In other words, a MOF file exported from a Windows XP/.NET WMI installation theoretically would compile on a UNIX implementation of the CIM. Although this is the DMTF's intention, currently various minor differences exist between Microsoft's and the DMTF's interpretation of the language. In reality, you may have to modify certain elements of the MOF file to remove Microsoft-specific qualifiers to export to non-Microsoft platforms.

To add classes, instances, or associations to the CIM repository, you first must specify the destination namespace of the objects that you wish to add. If you do not specify a destination namespace, WMI will place all entries in the `root/default` namespace in the CIM repository. You have several options for specifying a destination namespace. First, you can specify a namespace as a command-line switch for the MOF comp executable by using the `-N switch. This approach is limited, because you can specify only a single destination namespace for the entire MOF file. Second, you can
specify the namespace as part of the MOF file. This option will override all others. To do this use, the #pragma namespace compiler directive that follows:

```
#pragma namespace (destination namespace)
```

In the next example, any classes, instances, or associations that follow this statement will be destined for the local CIMV2 namespace.

```
#pragma namespace("\\\\\Root\\CIMV2")
```

The MOF includes various compiler directives that allow developers to set conditions that affect the compilation process. Compiler directives include locale, source, nonlocal, include, and instancelocale.

```
#pragma locale(language_country)  
```

The locale directive tells the compiler which language and country settings the MOF file will use. Remember that all MOF keywords (such as “class,” “instance,” and “association”) are always listed in English. When not specified, the default value is en_US (language = English and country = United States).

```
#pragma source(<namespace type>://<namespace_handle>)
```

Tells the compiler on which CIM implementation to locate the metadata that follows. You can use this pragma on class instances or association instances. You also can use the source qualifier on individual class and association instances.

```
#pragma nonlocal(<namespace type>://<namespace_handle>)
```

Tells the compiler to reference an object instance in another CIM implementation. This allows the CIM to share management information between machines and to allow enterprise-wide identification of objects. With this compiler directive, we can identify objects located on a machine across the network from the current host machine. For instance, while compiling on host machine A, we could reference an object instance held on a CIM-compliant repository on machine B on a different part of the network (that is, #pragma nonlocal("namespacetype://namespace_handle").

```
#pragma include("another.mof")
```

Tells the compiler to insert the contents of the MOF file given as a parameter into the current MOF file from this point onward.

```
#pragma instancelocale()
```

Tells the compiler to use the locale that follows the directive for instances declared in the MOF file. This, in effect, means that instances within the MOF file can have a different locale from that declared for classes. We shall cover how to declare instances using MOF later in this chapter.

**Terminology checkpoint:**

You compile class definitions into the CIM repository.

You import instance information into the CIM repository.
**MOF Class Declaration**

Declaring a class using MOF syntax is quite simple and not dissimilar from using C++. In Figure 4.6, we declare a fictitious class that represents some of the attributes of a CD-ROM drive. For the sake of simplicity, we do not derive this class from any classes within the CIM schema or Win32 extended schema.

![A MOF class declaration](image)

```mofang
class Storage
{
    [read, key: DisableOverride ToSubClass]
    uint64 DeviceId;
    // DeviceId is an unsigned integer 8 bytes
};
class MySchema_CDROM: Storage
{
    string ManufacturerName;
    string Model;
    uint16 ReadSpeed;
    string BIOSVersion;
    // This declares the properties for the CD ROM.
};
```

The class name is the *domain* of the properties or methods that it contains. In Figure 4.6, the property DeviceId has a domain of class Storage and the property Model has the domain of MySchema_CDROM. In the case of associations, which we discuss shortly, each reference has a *range*. The range refers to the class to which the reference points.

**Note:** All keywords in MOF are case **INSENSITIVE**

In Figure 4.6, we declared the class CD_ROM as a subclass of class Storage. To compile correctly, we must declare the class Storage *before* class CD_ROM.
We have declared four properties, ManufacturerName, Model, BIOSVersion, and ReadSpeed, as part of the class. Property names must not contain any spaces or underscores; otherwise, the compilation process will fail. Also observe that we used the // to place comments in the class declaration. The compilation process ignores these comments.

## Qualifiers

Qualifiers are the mechanism in MOF through which we can assign characteristics to any of the elements within a schema. These elements include methods, method parameters, properties, classes, and associations. A qualifier has the following components:

- A name (that is, description, abstract, read, write,...)
- An intrinsic data type (that is, real32, string, uint16,...)
- A value of the corresponding type (that is, Manufacturer_Name could be assigned the value “Mitsubishi”)
- A scope, to determine whether the qualifier can be applied to classes, methods, properties, and associations
- A flavor, to determine whether the qualifier can be inherited by subclasses or instances, or can be overridden

The DMTF defines a number of standard qualifiers for use when working with any schema (which are listed in the DMTF CIM Specification document). It is also possible to define your own qualifiers. Microsoft defines a number of custom qualifiers as part of their implementation of WMI. In Figure 4.7, we add default qualifiers and Microsoft-specific qualifiers to the class.

---

**Figure 4.7** Example qualifiers

```plaintext
[ Description("The CD_Rom class contains characteristics of CDROM drives"):
  ToSubClass,
  Locale(0x409),
  UUID("{BA46D060-7A6D-11d2-BC85-00104B2CF71C}") ]

class MySchema_CDROM:Storage
{
  [ write (true): ToSubClass ]
  string ManufacturerName;
  [ write (true): ToSubClass ]
```
string Model;
[ read,
    Description("This property contains the read speed of the CDROM" "Example: 32"): ToSubClass ]
uint16 ReadSpeed;
[ read,
    Description("This property defines the BIOS revision of the CDROM" "Example: 7.13.1200"): ToSubClass ]
string BIOSVersion;
};

Figure 4.7 shows the qualifiers listed in bold type. Always arrange qualifiers in blocks that immediately precede the class, method, or property that they characterize, and always enclose the blocks in square brackets: [ ]. For example:

[ Description ("This is a legal qualifier block")]

The Description qualifier at the start of the class declaration in Figure 4.7 is the qualifier name and provides a simple text description of class MySchema_CDROM. You must enclose the text itself in curved brackets and straight (nondirectional) quotation marks. Later, applications can use this text to retrieve a description of the class from the CIM repository.

### Flavors

Qualifiers also are characterized by how they transmit their information from parent to child class or from class to instance. To determine the rules by which a qualifier transmits its information, the MOF language defines special keywords called flavors. A colon always must precede qualifier flavors, as in Figure 4.7, and immediately after the description text, a semicolon precedes the qualifier flavor ToSubClass. This flavor indicates that every subclass of MySchema_CDROM inherits the qualifiers Description, Locale, and UUID. Figure 4.8 demonstrates some examples of flavors.

**Figure 4.8**  Example flavors

[ Description("The CDRom class contains characteristics of CDROM drives"): ToSubClass, Locale(0x409),
    UUID ("{BA46D060-7A6D-11d2-BC85-00104B2CF71C}")]
Excerpt from Chapter 4: A Guided Tour of the Common Information Model Repository

Other flavors include:

**EnableOverride.** This indicates that the qualifier can be overridden by child classes. The default value for this parameter is yes.

**DisableOverride.** This indicates that the qualifier cannot be overridden by child classes. The default value for this is no.

**ToSubClass.** This indicates that the qualifier is inherited by its child classes.

**ToInstance.** This indicates that the qualifier is propagated to instances.

**NotToInstance.** This indicates that the qualifier is not propagated to instances. This is the default.

**NotToSubclass.** This indicates that the qualifier is not propagated to derived classes. This is the default.

**Restricted.** This indicates that the qualifier is only valid for the class in which it is declared. Child classes do not inherit this qualifier.

**Translatable.** This indicates that the value can be specified in multiple locales.

### Custom Qualifiers

In designing the WBEM initiative, one of the goals of the DMTF was to provide technologies that were powerful and flexible enough to serve an environment as diverse as the enterprise. The CIM, for example, allows us to design specialized extensions using the extended schema. The MOF language also follows this philosophy to some extent with the creation of user-defined or custom qualifiers, whose only constraint is that qualifiers conform to one of the intrinsic data types defined in the MOF language. Schema designers can then use these qualifiers to represent specific characteristics unique to their schema. In Figure 4.8, the Locale qualifier and UUID qualifiers are both custom qualifiers defined by Microsoft and are not part of the default qualifiers defined by the DMTF. Examples of other qualifiers to which you will become accustomed in using WMI are Dynamic and Provider, which specify the use of Microsoft's dynamic providers for a class, instance, or property within the CIM repository.

The Locale qualifier is another Microsoft custom qualifier that provides WMI with a hex value that indicates the language and country of the class information that follows it. In this case, the locale ID 0x409 refers to US English. WMI can use this information to set specific information, such as date and time formats, character-sorting order, and decimal separators, when reading the following class or instance information.

In Figure 4.7, you can also see that the **MySchema_CDROM** class uses another custom qualifier, UUID. The UUID qualifier (which stands for Universally Unique IDentifier)
assigns a unique 128-bit value to each class. UUIDs are randomly generated 8-byte values, which are represented by a string of hex values in the CIM repository. They are so random that no two UUIDs generated on two different machines at the same time ever should be the same. If you are familiar with COM, then you will know UUIDs as the unique means of representing Class Identifiers (CLSIDs) and Interface Identifiers (IIDs). UUIDs also are referred to as GUIDs or Globally Unique Identifiers; the two terms can be used interchangeably. The Microsoft Platform SDK supplies two executables, guidgen.exe and Uuidgen.exe, which can generate these random values on request. WMI uses these values automatically to identify classes and events internally. Then the properties and methods within each class are referred to using a unique index value in conjunction with this UUID. The advantage of using UUIDs/GUIDs is that they are locale independent, that is, they are represented by the same binary value regardless of the locale setting. Microsoft developed a technique called the “Data Block GUID Mapping control” method for WMI that maps these unique UUIDs to a two-character ID within the driver. The driver understands the two-character IDs and therefore can interact with WMI. This mapping between the ID and UUID also enables drivers to support custom events, methods, and properties, which can be instrumented via WMI.

### User-Defined Qualifiers

New qualifier types are declared at the start of a MOF file: They begin with the qualifier keyword. The UUID qualifier, for example, could have been defined using MOF syntax similar to the following:

```
qualifier UUID :string = null, scope (class);
```

This syntax declares the qualifier called UUID, assigns it type string, and gives it a default value of zero. It then declares the qualifier to have a scope that affects only class declarations. Other settings that could have been used here include property, method, and reference, which would have allowed this qualifier to be used on these elements, also.

The read qualifier sets the property BIOSVersion to the read-only attribute. It then provides a description for the BIOSVersion property. ToSubClass signifies that any subclasses will inherit the BIOSVersion property.

### Intrinsic Data Types

In the example class, all of the properties are assigned one of the intrinsic data types. Table 4.4 shows the other property data types allowed in MOF format.
Instance creation can take place only after the corresponding class definition within a MOF file, which we declared in the previous section. The only case in which this rule does not apply is if the class definition already exists in the CIM repository within the target namespace. We create instances of classes using the MOF statement

\[
\text{instance of } \text{classname}
\]

Remember that MOF keywords are not case sensitive, so the case you use does not matter. To create an instance of the `CD_ROM` class, we could use the declaration in Figure 4.9.
Figure 4.9 Creating an instance of MOF

```MOF
instance of MySchema_CDROM
{
    DeviceId = 18446744071574384224; //property inherited by class storage
    ManufacturerName = "Sambutsu";
    Model = "CD1F";
    BIOSVersion = "1.34.230";
    ReadSpeed = 64;
};
```

Aliasing

When writing MOF files, it sometimes is expedient to create an alias that refers to a specific object instance. This alias can be used later in conjunction with a variable that is a pointer to an object (for example, the ref: keyword) to point to that specific object instance. If in the example in Figure 4.9 we declared an association class that referenced the CDROM class, we could use aliasing as in Figure 4.10.

Figure 4.10 Using an alias in MOF

```MOF
instance of MySchema_CDROM as $CDROM
{
    DeviceId = 18446744071574384224; //property inherited by class storage
    ManufacturerName = "Sambutsu";
    Model = "CD1F";
    BIOSVersion = "1.34.230";
    ReadSpeed = 64;
};
[association]
class MYAssociation
{
    MySchema_CDROM REF CDDrive;
    // Note that ordinarily this would not be a legal association as it does not contain the minimum of two REF keywords.
};
instance of MYAssociation
{
    CDDrive = $CDROM;
};
```
A Compileable MOF File

While examining the syntax and semantics of the MOF language by reading examples is an important part of the learning process, it is far more beneficial to gain practical experience through typing and compiling your own examples. Type the following MOF file, which is based upon the previous examples, and follow the instructions to compile it. After you are confident with this process, then experiment and alter sections of the file to see the results.

In the example, we shall create our own namespace, to avoid cluttering any existing system namespaces. Microsoft recommends using the root/default namespace in developing schemas, although we shall create our own in this example.

To write a MOF file, you will need a simple ASCII text editor, such as Microsoft's WordPad or Notepad.

If you have Microsoft Developer Studio, you can configure the settings under Tools/Customize/Tools so the development environment will compile a MOF file for you. For more information, see Microsoft's Learn WMI tutorial at http://msdn.microsoft.com/downloads/sample.asp?url=/msdn-files/027/001/574/msdncompositedoc.xml.

First, open your ASCII text editor or Developer Studio. Within the window, type the MOF text in Figure 4.11.

Figure 4.11 A compilable MOF (part 1 of 2)

```moi
//Beginning of sample MOF file.
#pragma classflags("forceupdate")
#pragma namespace ("\\.\\Root")
instance of __Namespace
{
    Name = "Example1";
};
```

The initial `#pragma classflags("forceupdate")` directive tells the compiler to update any existing class definitions in the CIM repository that match those with the class definitions supplied in this MOF file and to resolve conflicts where possible. The update process applies to properties, methods, and qualifiers contained within a class. If the changes within a class are extensive, then it is always better first to delete the existing class from the repository and then to compile the new one. Be aware, however, that doing this deletes all child class definitions!

The CIM repository employs inheritance, which leads to some very real consequences when adding new classes to the repository. For example, replacing an existing
class in the hierarchy or changing one of the qualifiers in an existing base class can have direct consequences to the later classes.

Consider a parent class, Storage, and a child class, CDROM. Imagine that the child class is defined with the class qualifier Write set to true. If we try to add base class, Storage, with the class qualifier Read, set to true, a conflict exists between the child class's qualifier and the base class. In this instance, with the forceupdate flag specified, WMI would remove the qualifier from the child class to make way for the base class. The only situation in which this would not occur is if the child class has instances. If the specified flag were safeupdate, then the update would fail at that point because of the conflict with the child class.

In addition to the forceupdate flag, a number of additional settings exist to resolve conflicts when adding classes. You can combine these options, although you should avoid combining two options that effectively cancel each other out, such as updateonly and createonly. The available options follow:

- **forceupdate.** Upon conflict with existing classes, override this option with new settings. It fails if child classes have instances.
- **safeupdate.** This option allows a safe update of the classes. Do not override existing settings with new settings.
- **updateonly.** This option updates existing classes only. It fails if the classes do not already exist in the target namespace.
- **createonly.** This option adds new classes only. It fails if classes already exist in the repository.

The second `#pragma namespace( )` (see Figure 4.12) tells the compiler to place any classes or instances following the statement into the root namespace on the host machine. The instance declaration of type class `__namespace` tells the compiler to create a new namespace instance. This tells WMI to create a new namespace in the CIM repository with the value assigned to the `name` property. The `__namespace` class in the root namespace is a WMI system class that WMI uses to store all the namespaces held in the CIM repository. We have assigned the key property name a value of `Example1`. The CIM repository now has an additional namespace into which we shall place our example classes.

In Figure 4.13, this section of the MOF file starts by instructing the compiler to place all of the new classes and instances into the newly created Example1 namespace. It does this by specifying `#pragma namespace ("\\\"><\Root\Example1")`. In experimenting with new classes and associations within the CIM repository, it is good practice to keep all example classes away from the important systems management classes (such as those stored in the CIMV2 namespace) where they could accidentally cause problems. We do this by placing them in a separate namespace. When you have tested
them and are satisfied that they are correct, you can merge them back into your target namespace (for example, the CIMV2 namespace).

Figure 4.12  The CIM repository with the new namespace, Example1

them and are satisfied that they are correct, you can merge them back into your target namespace (for example, the CIMV2 namespace).

Figure 4.13  A compileable MOF (part 2 of 3)
The next statement declares an abstract base class called `MySchema_Storage`. This class has a single property called `DeviceId`, which is declared as both a read-only property and the key property. The read-only qualifier signifies that the value of the property cannot be changed after the instance has been created. From the perspective of a client application, this means that the value of this property cannot be altered. This property also is flagged as a key value because it will uniquely identify instances of the class.

Now type the code in Figure 4.14.

**Figure 4.14**  A compileable MOF (Part 3 of 3)

```plaintext
instance of MySchema_CDROM
{
    DeviceId = 384224; //property inherited by class storage
    ManufacturerName = "Sambutsu";
    Model = "CD1F";
    BIOSVersion = "1.34.230";
    ReadSpeed = 64;
}; // End of sample MOF File
```

The final section of the example declares a single instance of class `MySchema_CDROM`. The instance assigns values to each of the properties and places them in the CIM repository.
Now save the file as myexample.mof, go to a command-line prompt, and type the following:

```
C:\ mofcomp myexample.mof
Microsoft (R) 32-bit MOF Compiler Version 5.1.2600.0
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: myexample.mof
MOF file has been successfully parsed
Storing data in the repository...
Done!
C:\
```

Congratulations! You have created your first namespace, added classes to it, and created an instance of a class within the CIM repository.

**Note:** At this point, you could have used the MOF generator wizard that is part of the WMI Developer Studio to compile your sample MOF file into the CIM repository. We shall look at using the wizards in more detail in Chapter 5.

### ActiveX Components

You must have Microsoft's WMI SDK installed to participate in the tour of the CIM repository. The WMI SDK contains a collection of ActiveX components to help you administer and access WMI. They install only as part of the WMI SDK: All are in wbemtool.cab file. (CAB, shorthand for CABinet, is Microsoft's mechanism for compressing and packaging files. For more information, go to http://msdn.microsoft.com/workshop/components/activex/packaging.asp.) You can unpack them and use them independently in any compliant ActiveX container. Microsoft supplies the ActiveX controls as freely distributable components with the WMI SDK. You need Internet Explorer 4.01 or later installed to run programs that come with the WMI SDK. Full details of the various ActiveX controls are in the WMI SDK documentation.

You will find short cuts to these tools under the Start/All Programs/WMI SDK folder. This folder contains useful utilities for administering and testing WMI. The ActiveX components all use the WMI API to access WinMgmt.exe.
Excerpt from Chapter 7: Developing Management Applications

With schema design and provider development complete, the next step is to learn to write applications to administer the managed objects. Typically, client-side applications will provide a presentation layer for the managed objects. The presentation layer usually includes creating and deleting management objects, getting and setting properties, and calling methods.

For example, let’s assume that you have created a class for managing the debug-logging configuration of your Windows 2000 service. A client-side application will allow you set the properties that will enable debug logging, and your Windows 2000 service will typically retrieve this configuration to decide whether it should perform debug logging.

This chapter focuses on the COM APIs that allow you to interact with WMI so that you can manage your objects. Later, in Chapter 9, we will focus on how you can write an MMC snap-in to provide a presentation layer for your managed objects.

This chapter assumes your familiarity with C++ programming, COM, and the Active Template Library (ATL). It also assumes that you are aware that managed objects can be created, updated, deleted, enumerated, and queried. You can manipulate the properties on objects, execute methods, and subscribe to receive WMI events. In this chapter, we discuss in detail how you can write applications to manage management objects. We discuss issues such as flag parameters to give you a complete guide to making WMI calls in your applications. By the end of this chapter, you should be knowledgeable about virtually every key aspect of writing a managed application. The first half of this chapter covers the basics, probably enough for most people; the second half covers more advanced topics.

In this chapter we use the terms “management object,” “managed object,” “object,” and “instance” interchangeably.

Getting Started

The first thing you must do is connect to the WMI service. The WMI architecture is based on manipulating objects in a single namespace. In the debug logging example, to enable debug logging on machine A and machine B, you must connect to machine A, set the appropriate properties, and then connect to machine B and do the same for that machine. Let us look at some code that connects to WMI.

```cpp
HRESULT hr = S_OK;

CComPtr<IWbemLocator> spLoc;
hr = spLoc.CoCreateInstance(CLSID_WbemLocator);
```
CComBSTR bstrNamespace("\\\.\root\CIMV2");
CComPtr<IWbemServices> spServices;

// Connect to CIM
hr = spLoc->ConnectServer(bstrNamespace, NULL, NULL, 0,
    NULL, 0, 0, &spServices);

// Switch the security level to IMPERSONATE so that the provider
// will grant access to system-level objects.
hr = CoSetProxyBlanket(spServices, RPC_C_AUTHN_DEFAULT,
    RPC_C_AUTHZ_NONE, NULL, RPC_C_AUTHN_LEVEL_CALL,
    RPC_C_IMP_LEVEL_IMPERSONATE, NULL, EOAC_NONE);

The first call we make is to get a WMI locator interface from the Windows 2000/XP WMI service. This is always the first operation you will do. The IWbemLocator interface is our gateway to connect to any namespace on any machine. This, of course, assumes that the namespace exists and that the machine has the WMI service. Under Windows 2000, the WMI service runs in a process called WinMgmt.exe under the local system account. Under Windows XP, WMI is a service process within SVCHOST.

The WMI locator interface looks like this:

```
interface IWbemLocator
{
    HRESULT ConnectServer(
        const BSTR strNetworkResource,
        const BSTR strUser,
        const BSTR strPassword,
        const BSTR strLocale,                   // MS_409 for example
        LONG fSecurityFlags,                     // must be zero
        const BSTR strAuthority,                 // Security authority
        IWbemContext *pCtx,
        IWbemServices **ppNamespace);         // Your link to the WMI world
};
```

The first parameter, `strNetworkResource`, specifies both the machine and the namespace to which to connect. Format the string in the following way:

```
\<machine name>\<namespace>
```

In the previous code sample, “\.” specifies the current machine, although we could have specified “\MACHINE_A” to connect to a specific machine. The “root\CIMV2” specifies the namespace. The connect string to connect to the CIMV2 namespace on MACHINE_A would look like:

```
\MACHINE_A\root\CIMV2
```
The `strLocale` parameter allows you to specify the locale for retrieving localized class information, such as class or property descriptions. The format of this parameter starts with "MS_” and you append the Win32 LCID locale identifier in hex. “MS_409” is U.S. English and “MS_407” is German. Inclusion of this parameter makes sense only when you have a localized namespace. If a namespace is localized in U.S. English and German, you will see subnamespaces “ms_409” and “ms_407”, respectively. Localized namespaces use amended qualifiers, which we discuss later in the chapter. If the namespace is not localized, then you should pass NULL.

Use the `pCtx` parameter only to make WMI calls within dynamic providers. We discuss this in Chapter 12.

The final parameter, `ppNamespace`, is virtually everything you will need when you access management objects. You will need the `IWbemServices` interface for:

- Getting, deleting, and updating WMI instances
- Enumerating all instances of a specific WMI class
- Performing WQL queries
- Subscribing to events

Before you can use the `IWbemServices` interface, you must set up the proxy’s security context by calling `CoSetProxyBlanket`, which allows you to specify the authentication and impersonation details of the user’s security context. If you do not call `CoSetProxyBlanket`, then you may experience access-denied errors from the WMI provider. This is the case when accessing management objects from the Win32 providers supplied with Windows 2000/XP. Providers may make calls to system-level APIs and/or make security checks against an ACL, and, therefore, you must ensure that you use the correct security context. In this chapter, we assume that you will not be able to call `CoInitializeSecurity`; this is why we call `CoSetProxyBlanket` instead. We will discuss security in greater detail later in the chapter.

All the code samples in this chapter demonstrate how to use the `IWbemServices` interface. First, let’s examine the three ways that you can make method calls.

1. **Synchronous.** This performs an operation, and the thread stays blocked until the call completes. Depending on the operation, a synchronous method call can turn out to be lengthy, especially when performing queries and enumerating large sets of instances. Most of the code samples in this chapter use the synchronous approach to describe the steps necessary to code your own applications. It is a good starting point.

2. **Asynchronous.** This starts an operation and immediately returns. You must implement `IWbemObjectSink`, which WMI will use to send all instances and
other progress information back to you. For example, `IWbemObjectSink`'s `Indicate` method passes an instance found during a query back to you. This mechanism may be a little harder to use and involves the development of more code. Asynchronous development usually involves writing applications with multiple threads and determining program flow may be a little unpredictable. We cover this in more detail later in the chapter.

3. **Semisynchronous.** This combines parts of the synchronous and asynchronous approaches. It allows you to make a call and return immediately without having to provide an implementation of `IWbemObjectSink`. Instead, the call returns an `IWbemCallResult` interface for you to poll and gather the results of the operation, such as a query. This solves the problem of thread blockage and the use of multiple threads with no additional code. We cover this in more detail later in the chapter.

### Object Paths Explained

Throughout the rest of this chapter, you will see the terms “object paths” or “object reference” mentioned regularly. Understanding object paths is important because it is your way to ask WMI for a management object that you want, and it is also WMI's way of providing you a reference about a particular object.

An object path is a string that can reference either a class or object instance specifically. This can be easily explained with an example from file systems. The following string represents a path to a particular file:

```
C:\MyDocuments\MyLetter.doc
```

If you provide this path to the file system (using the appropriate APIs), the file system can provide you with a file handle that you can later use either to read or to write to. This concept is the same for object paths.

An object path can be either a fully qualified path or a relative path. The format for a fully qualified path is:

```
\\machine\namespace:classname.keyproperty1=value,keyproperty2=value
```

In the following two examples, the first references a user account and the second references a shared directory.

```
\\GWCOLE\root\CIMV2:Win32_UserAccount.Domain="GWCOLE",Name="Administrator"
\\GWCOLE\root\CIMV2:Win32_Share.Name="C$"
```
Notice that a dot separates the class name from the first key property and that a comma separates all subsequent key properties. We collectively refer to a class with multiple key properties as a compound key.

Relative object paths omit the machine and namespace names. The format for a relative object path is:

\classname.keyproperty1=value,keyproperty2=value

The same object path examples above would look like this as relative object paths:

\Win32_UserAccount.Domain="GWCOLE",Name="Administrator"
\Win32_Share.Name="C$"

Using relative object paths is easiest in most cases because you specified the machine and namespace during the initial connection to WMI (using the ConnectServer call).

All key properties that use a string data type need to be enclosed in quotation marks (as in the above examples). The following data types use strings: strings, dates and times, and object references. Key properties that are numbers follow the equals operator, as in this example:

\Win32_DMAChannel.DMAChannel=2

All the examples so far have been references to actual management objects. Let us look at how you can refer to classes, which is useful when you want to gather information about a class (such as enumerating the defined properties). As you will see later, static method calls require class references, rather than an object reference. You will find more on method calls later. The format for a fully qualified path to a class is:

\\machine
namespace:classname

Here is an example referencing the Win32_VideoController class on a particular machine:

\\GWCOLE\ROOT\CIMV2:Win32_VideoController

As before, relative class paths are a bit friendlier than fully qualified paths. The format for a relative path to a class is:

classname
Finally, this is an example of a relative reference to the `Win32_VideoController` class:

```
Win32_VideoController
```

Object paths are not case sensitive, and they must not contain spaces except inside string literals.

## Getting an Object

Getting an object is one of the most basic operations to perform. This kind of operation is especially useful when you know which management object you’re after. For example, if you decide to gain access to some properties of a shared directory on a machine, `GetObject` is the operation you should perform. Before you can gain access to a management object, you need a valid object path. Here is an example: If you look at the `Win32_Share` class in the CIMV2 namespace, you will see that it has one key property, “Name”. If you expect the C$ shared directory on a machine to exist, you can get a management object that represents the shared directory. You need to generate an object path that includes the one and only key value, `Win32_Share.Name="C$"`, and call `GetObject`.

You may also get an object path from a property. WMI includes a data type that can reference other objects; this is an object reference. Association classes include properties that are object references to make an association. You may read one of these properties and pass the object reference (which is a string) straight into the `GetObject` call. Either way, as long as you have a valid object path, you will be able to access the management object.

The following code illustrates how to get the path of a shared directory:

```c
CComBSTR bstrPath("Win32_Share.Name="C$"");

// Get the shared directory object
CComPtr<IWbemClassObject> spInstance;
hr = spServices->GetObject(bstrPath, WBEM_FLAG_RETURN_WBEM_COMPLETE,
    NULL, &spInstance, NULL);

// Get a property from the object
CComVariant var;
hr = spInstance->Get(CComBSTR("Path"), 0, &var, 0, 0);

_tprintf(_T("Win32_Share.Name="C$" path is %ls\n"), V_BSTR(&var));
```
The above example merely retrieves the path of the shared directory and outputs it to the screen. `WBEM_FLAG_RETURN_WBEM_COMPLETE` means complete the entire call before returning (implied by default because it has a value of zero). Manipulating object properties will be covered later in the chapter, as will other flags that can be used.

A portion of the `IWbemServices` interface looks like this:

```c
interface IWbemServices
{
    HRESULT GetObject(
        const BSTR strObjectPath, // The object path
        LONG lFlags,             // How to obtain the object
        IWbemContext *pCtx,      // Used in provider
        IWbemClassObject **ppObject, // The managed object
        IWbemCallResult **ppCallResult); // Don't use this for now
};
```

The `strObjectPath` parameter lets you pass the object path of the management object to which you’d like to gain access.

The `lFlags` parameter specifies how the `GetObject` call should be made. More advanced flags that can be passed, such as `WBEM_FLAG_RETURN_IMMEDIATELY`, which allows you to perform semisynchronous calls, will be covered later in the chapter.

The `ppObject` out-parameter returns the management object specified in the `strObjectPath` parameter. All management objects are represented through an `IWbemClassObject` interface, which provides access to all the properties as defined in the management class.

### Enumerating Objects

In some cases, you will want a complete list of all management objects of a given class. Suppose you have a user interface that allows the user to select a shared directory to complete some task; you would want to display a list of all the available shared directories. You can achieve this by enumerating all the instances of the `Win32_Share` class. You could perform the following query to achieve the same result:

```
SELECT * FROM Win32_Share
```

However, performing a query involves processing the query, obtaining a list of objects, and comparing the query against each object. In this case, it is more efficient to enumerate the objects in a class. To perform an enumeration, all you need to know is the name of the class you would like to enumerate.

Enumerating objects for a given class is one of the easiest ways to discover what management objects are available. When you move through classes in CIM Studio
Excerpt from Chapter 7: Developing Management Applications

(which comes as a developer tool in the Platform SDK), one of the first things you do is to look at a list of instances to see what can be managed. Review Chapter 4’s quick introduction to CIM Studio. You may also want to check out wbemtest, another WMI SDK tool that very closely mimics the IWbemServices interface. It can be useful in developing your management applications.

The following code sample illustrates how to obtain a list of shared directories:

```c++
// Get list of objects for the Win32_Share class
CComPtr<IEnumWbemClassObject> spEnumInst;
hr = spServices->CreateInstanceEnum(CComBSTR("Win32_Share"),
    WBEM_FLAG_SHALLOW, NULL, &spEnumInst);

bool bFinished = false;
while (!bFinished)
{
    // Get the Win32_Share instance
    ULONG uNumOfInstances = 0;
    CComPtr<IWbemClassObject> spInstance;
    HRESULT hrNext = spEnumInst->Next(10000, 1, &spInstance,
        &uNumOfInstances);

    if (hrNext == WBEM_S_FALSE)
        bFinished = true;
    else if (hrNext == WBEM_S_NO_ERROR)
    {
        // Get properties from the Win32_Share instance
        CComVariant varRelPath;
        hr = spInstance->Get(CComBSTR("__RELPATH"), 0,
            &varRelPath, 0, 0);

        CComVariant varPath;
        hr = spInstance->Get(CComBSTR("Path"), 0, &varPath, 0, 0);

        _tprintf(_T(" %ls path is %ls\n"), V_BSTR(&varRelPath),
            V_BSTR(&varPath));
    }
}
```

The CreateInstanceEnum call creates an enumerator that can be used to gain access to every instance of that class. The enumerator is returned in an IEnumWbemClassObject interface and the Next method call provides access to an IWbemClassObject interface (that is, an instance). The code sample specified WBEM_FLAG_SHALLOW, which instructed CreateInstanceEnum to return only an enumerator that has the instances of the Win32_Share class. A quick check is done to determine whether there are more instances to retrieve and if there are, some output is made. Notice that __RELPATH was specified as a property name: _RELPATH is the relative path of the object. WMI has many system-
provided properties for classes and instances; we discuss these in more detail later in the chapter. The output of this on our machine produces the following:

```csharp
    Win32_Share.Name="C$" path is C:\
    Win32_Share.Name="IPC$" path is
    Win32_Share.Name="ADMIN$" path is C:\WINNT
```

A portion of the `IWbemServices` interface looks like this:

```csharp
    interface IWbemServices
    {
    HRESULT CreateInstanceEnum(
        const BSTR strClass, // Class name to enumerate
        LONG lFlags, // WBEM_FLAG_SHALLOW
        IWbemContext *pCtx, // Used in providers
        IEnumWbemClassObject **ppEnum); // Your Enumerator
    }
```

The `strClass` parameter specifies the name of the class you want to enumerate and is returned in the `ppEnum` out-parameter. You can specify how the enumerator should be built based on the `lFlags` parameter. For instance, if the class you want to enumerate is a base (or abstract) class (introduced in Chapter 5), then you might want to use `WBEM_FLAG_DEEP` instead of `WBEM_FLAG_SHALLOW` to build your enumerator. This will cause WMI to provide instances not only of the base class but also of all its derived classes in the enumerator. Let us have a look at an example. Suppose you want to see all security accounts on your system. If you use CIM Studio (see Figure 7.1) to look at the CIMV2 schema, you will see that the `Win32_Account` is a base class for `Win32_SystemAccount`, `Win32_Group`, and `Win32_UserAccount` classes. If you call `CreateInstanceEnum` for the `Win32_Account` class and specify `WBEM_FLAG_DEEP` you will get an enumerator that will contain all instances of `Win32_SystemAccount`, `Win32_Group`, and `Win32_UserAccount`.

To recap, the following call will provide only an enumerator of all the instances of the `Win32_SystemAccount` class.

```csharp
    CComPtr<IEnumWbemClassObject> spEnumInst;
    hr = spServices->CreateInstanceEnum(CComBSTR("Win32_SystemAccount"),
                                        WBEM_FLAG_SHALLOW, NULL, &spEnumInst);
```

Figure 7.1  Win32_Account class hierarchy
This call will provide an enumerator of all the instances of the `Win32_SystemAccount`, `Win32_Group`, and `Win32_UserAccount` classes. Note that `WBEM_FLAG_DEEP` typically is used when all instances of classes derived from a base class are required.

```cpp
CComPtr<IEnumWbemClassObject> spEnumInst;
hr = spServices->CreateInstanceEnum(CComBSTR("Win32_Account"),
                                   WBEM_FLAG_DEEP, NULL, &spEnumInst);
```

Other flags can be specified in the `lFlags` parameter. `WBEM_FLAG_FORWARD_ONLY` is one of them; its use produces a forward-only enumerator. This type of enumerator generally is faster and requires less memory; however, you won’t be able to call the `Clone` or `Reset` methods. If you do not need to use `Clone` or `Reset`, we recommend that you specify this flag on all your calls.

The `WBEM_FLAG_BIDIRECTIONAL` is implied by default (because it has the value zero). This flag means that WMI will retain pointers to instances until the enumerator is released. In practice, the instances are not released immediately. If the class you are enumerating has a very large collection of instances, you may experience `WBEM_E_OUT_OF_MEMORY` being returned by `CreateInstanceEnum`. In this case, you should use the asynchronous version, `CreateInstanceEnumAsync`. (A forward-only enumerator can save some memory, but it typically won’t be enough of a saving if you expect a very large collection of instances.) Other flags will be discussed later in the chapter.

### Creating an Object

When writing applications to create management objects, you need to take into account how, if at all, those objects can be created. This applies equally to how you foresee the creation of your own management objects when designing your schema. Creating a shared directory on a machine is an example of creating an object. There are three ways to look at the creation of objects.

1. Use the standard mechanism provided by WMI through `IWbemServices`.
2. Use a method such as `Create`. The `Win32_Share` class does this.
3. You cannot create instances at all, as in the case of the `Win32_ComputerSystem` class.

### First Point

When creating instances in the standard mechanism, you need to spawn an uncommitted instance based on the class definition, set your properties, and call `PutInstance` on
the IWbemServices interface. (The code sample later in this section illustrates this process.)

Generally, you are required to set all the key properties on the newly spawned instance before passing it to PutInstance. Key properties identify an object explicitly and are required to access it in the future. In some cases, the provider may assign a value to a key property if it discovers that one does not exist, but this is not common. It is considered best practice that the key properties be set when you create your management objects. To see what this means, let us look at the following class:

```csharp
class Sample_Book
{
    [key] string BookTitle;
    [key] string AuthorName;
    string Summary;
    DATETIME DatePublished;
    uint8 NumberOfChapters;
};
```

The only properties that have to be set are BookTitle and AuthorName. This means that the other properties need not be set to create a valid instance. This can pose some danger, especially if you decide that you also require Summary and NumberOfChapters properties. However, this danger is not so bad if you control the source code in the provider. You simply ensure that the remaining two required properties are set; otherwise PutInstance could return something like WBEM_E_INVALID_OBJECT. Of course, this isn't guaranteed with other providers.

WMI classes that support the standard mechanism of creating management objects have the supportsCreate qualifier.

**Second Point**

To clearly state what information is required to create an instance, some classes use a method to create an instance. This is the case with the Win32_Share class, which has a method called Create. You first would need to set up a method call and then execute it. The next time you enumerate the Win32_Share class, the new instance will be listed as part of the collection. Let us look again at the Sample_Book class, this time with a method to create an instance:

```csharp
class Sample_Book
{
    [key] string BookTitle;
    [key] string AuthorName;
    string Summary;
    DATETIME DatePublished;
    uint8 NumberOfChapters;
};
```
As it is clear to see, the `Create` method requires both `Summary` and `NumberOfChapters` values, as well as the key values. Also, notice that the method returns a `boolean` value to indicate whether the method succeeded. (You could instead return a `uint32` if you want to return a HRESULT. However, remember that the method can be called from scripting environments where HRESULTs may not be the most suitable return value.) Here are other reasons that you might want a method for the creation of instances:

1. The key values are not known in advance and are generated by the provider.
2. You may want to have specific flags passed into the method for more context information when creating the instance.
3. You may want to receive one or more out-parameters (other than the return value).

If you find that a method exists for the creation of instances, it may contain the `Constructor` qualifier to signify that the method creates instances.

**Third Point**

Instances cannot be created for some classes, as in the case of the `Win32_ComputerSystem` class. If you think about it, it does not make sense to have more than one instance of this class. You may find with your own schema that you cannot create instances and the only supported operations allowed are to update, enumerate, and perform queries.

**Example**

Let's examine how you might create an object with the standard mechanism. We discuss calling methods on classes later in the chapter.

```csharp
// Get class so we can spawn an instance of it
CComPtr<IWbemClassObject> spClass;
hr = spServices->GetObject(CComBSTR("Sample_Book"), 0,
    NULL, &spClass, NULL);

// Make new object
CComPtr<IWbemClassObject> spInstance;
hr = spClass->SpawnInstance(0, &spInstance);
```
// Set some properties on the object
CComVariant varBookTitle("Developing WMI Solutions");
hr = spInstance->Put(CComBSTR("BookTitle"), 0, &varBookTitle, 0);

CComVariant varAuthor("Gwyn Cole");
hr = spInstance->Put(CComBSTR("AuthorName"), 0, &varAuthor, 0);

CComVariant varSummary("A cool book on Windows Management Instrumentation");
hr = spInstance->Put(CComBSTR("Summary"), 0, &varSummary, 0);

CComVariant varNumOfChapters(int(11));
hr = spInstance->Put(CComBSTR("NumberOfChapters"), 0, &varNumOfChapters, 0);

// Commit to create instance in WMI
hr = spServices->PutInstance(spInstance, WBEM_FLAG_CREATE_ONLY, NULL, NULL);

The first task is to spawn an uncommitted instance based on the class definition. Calling SpawnInstance does this for the Sample_Book class in this case. We call this an uncommitted instance because it has not yet been passed to WMI and normal operations such as setting and getting properties can be performed. Next, we do exactly that, setting all the properties (as long as we include the key properties) of the new instance. After all the properties have been set, use the IWbemServices interface to pass the instance to WMI by calling PutInstance. The provider will extract all the properties from the instance and physically do whatever the provider needs to do to create the instance.

This is what the SpawnInstance method looks like:

interface IWbemClassObject
{
    HRESULT SpawnInstance(
        LONG TFlags,  // must be zero
        IWbemClassObject **ppNewInstance);  // New instance
};

When SpawnInstance returns, the ppNewInstance out-parameter will contain the new instance, ready for properties to be set. The TFlags parameter is documented as having to be 0.

The PutInstance method looks like this:

interface IWbemServices
{
    HRESULT PutInstance(
        IWbemClassObject *pInst,  // Instance to create/update
The uncommitted instance is passed in the pInst parameter. The lFlags parameter specifies the kind of put-operation that must be performed. In the code sample, we used WBEM_FLAG_CREATE_ONLY, which tells WMI that this operation can create only instances. If you already had an instance with identical key properties, the PutInstance call would fail with WBEM_E_ALREADY_EXISTS. If we had used WBEM_FLAG_CREATE_OR_UPDATE instead, the PutInstance call would not have failed and an update would have been attempted. Use WBEM_FLAG_CREATE_ONLY when you do not intend for the instance to be updated accidentally. We cover updating instances such as WBEM_FLAG_UPDATE_ONLY in the next section.

If the lFlags parameter has WBEM_FLAG_RETURN_IMMEDIATELY set, then you will receive an IWbemCallResult interface pointer from the ppCallResult out-parameter. This allows you to perform semisynchronous calls. We discuss this later in the chapter.

An example of creating an object with a method will be demonstrated later in the chapter when we discuss WMI method calls.

## Updating Objects

Updating existing management objects with new property values is a common requirement when writing applications. As with creating instances, there are three options: to use the standard mechanism to update, to use a method to update, and to be unable to update at all. The standard mechanism for updating instance property values is very similar to creating a new instance because you still make a call to PutInstance. The general process is to get your instance, change the properties, and call the PutInstance method. The following code sample changes a property value on the book instance we created earlier:

```c
// Generate the object path
CComBSTR bstrPath = _T("Sample_Book.BookTitle=" Developing WMI Solutions", AuthorName="Gwyn Cole\"\"");

// Get the Sample_Book object we want
CComPtr<IWbemClassObject> spInstance;
hr = spServices->GetObject(bstrPath, 0, NULL, &spInstance, NULL));

// Change the number of chapters
CComVariant varNumOfChapters(int(12));
hr = spInstance->Put(CComBSTR("NumberOfChapters"), 0, &varNumOfChapters, 0);
```
// Commit to update instance in WMI
hr = spServices->PutInstance(spInstance, WBEM_FLAG_UPDATE_ONLY,
    NULL, NULL);

As you can see, updating an instance is straightforward. We changed only the number of chapters property for this book; all the other property values remained unaffected. However, notice the flags parameter in the PutInstance call: This time WBEM_FLAG_UPDATE_ONLY was specified. The WBEM_FLAG_UPDATE_ONLY will allow only for an update to an already existing instance. If the Sample_Book instance did not exist, the PutInstance method would return WBEM_E_NOT_FOUND. Using WBEM_FLAG_CREATE_OR_UPDATE instead would have allowed the provider to do whatever is necessary to update the instance, even if it had to create it. Use WBEM_FLAG_UPDATE_ONLY in situations where you do not want accidental creation of instances.

Some classes use methods to update instances with very similar reasons to those described in the previous section. For example, the Win32_Share class has a method called SetShareInfo in which the description of the shared directory, together with some other properties, can be changed. The Win32_Service class is another example: It has two methods to update an instance, Change and ChangeStartMode.

WMI classes that support the standard mechanism of updating management objects have the supportsUpdate qualifier.

### Deleting an Object

When writing applications, you will occasionally want to delete management objects. For instance, you might want to remove access to an already existing shared directory, which would involve deleting the instance of the shared directory management object. As with creating and updating instances, you have three options: to use the standard mechanism to delete, to use a method to delete, and to be unable to delete at all.

For some classes, such as the Win32_VideoController class, it makes no sense to delete instances. This is a hardware device, and to remove the instance would involve physically removing the video controller from your computer.

The easiest method of removing instances is to use the standard mechanism to delete an object. All you need is a valid object path and to make a call to DeleteInstance. WMI classes that support the standard mechanism of deleting management objects have the supportsDelete qualifier.

In some classes, you may find that a deletion method is available. Here are some reasons that you might want a method for the deletion of instances:

1. You may want to have specific flags passed into the method for more context information when deleting the instance.
2. You may want to receive one or more out-parameters other than the return value.
If you find that a method exists for the deletion of instances, then you can expect the standard mechanism to fail. This depends on the provider.

Let us look again at the Sample_Book class, this time with a method to delete an instance:

```csharp
class Sample_Book
{
    [key] string BookTitle;
    [key] string AuthorName;
    string Summary;
    DATETIME DatePublished;
    uint8 NumberOfChapters;

    [static, implemented] boolean Create([in] string BookTitle,
            [in] string AuthorName, [in] string Summary,
            [in] uint8 NumberOfChapters);

    [implemented] boolean Delete();
};
```

Did you observe that the `Delete` method does not include the key properties of the class? How will the `Delete` method know which book to delete? The key reason is that the `Delete` method is an object method and is relative to the management object. The `Create` method was defined with the `static` qualifier, which defines it as a class method. This effectively means the method can be called without reference to any instances of the class. The `static` qualifier is the same concept as in object-oriented programming. The `Delete` method can be called only relative to the management object.

Deleting an instance is a very easy operation with the standard mechanism. You need only to pass an object path to `DeleteInstance` and the provider will attempt to remove the instance. Let us remove the `Sample_Book` instance that we created earlier:

```csharp
// Generate the object path
CComBSTR bstrPath = _T("Sample_Book.BookTitle=" Developing WMI Solutions", AuthorName="Gwyn Cole");

// Delete Sample_Book
hr = spServices->DeleteInstance(bstrPath, 0, NULL, NULL));
```

The `DeleteInstance` method looks like this:

```csharp
interface IWbemServices
{
    HRESULT DeleteInstance(
        const BSTR strObjectPath, // Object path to DELETE
        LONG lFlags, // Usually zero
        IWbemContext *pCtx, // Used in providers
        IWbemCallResult **ppCallResult); // Not needed for now
};
```
The `strObjectPath` parameter specifies the object path of the management object to delete. When `DeleteInstance` is called, the provider will verify that the object path does reference a valid instance and then it will attempt to remove it permanently. When writing your applications, be careful not to hold onto an object (an `IWBemClassObject` pointer) and later delete the object while still holding a reference to the instance. This will result in unpredictable behavior.

The `lFlags` parameter allows just one flag, `WBEM_FLAG_RETURN_IMMEDIATELY`, which is used when executing semisynchronous calls.

### Performing Queries

Performing queries is one of the most popular mechanisms of gaining access to managed objects. The reason is that it allows great flexibility in specifying exactly what you want. When writing your own applications, you will inevitably find that you must perform queries that fall into one of two categories:

1. You need context-relative information.
2. You need to improve performance.

For an example of context-relative information, suppose that a task requires a security principle (that is, a user) from a particular domain. A user interface application could formulate the following WQL query:

```
SELECT * FROM Win32_UserAccount WHERE Domain="DOMAIN_A"
```

This query would provide you with a full list of all the users in DOMAIN_A. You can also use a query to improve performance by specifying the properties you require. The above example asked for all properties from the `Win32_UserAccount` class (that is, the `SELECT *`). You do not need all the properties if you simply want to display a list of users and use the SID (security identifier) to perform some task. You can revise the query to ask specifically for the display name and SID.

```
SELECT FullName, SID FROM Win32_UserAccount WHERE Domain="DOMAIN_A"
```

This query would return only the `FullName` and SID property values for each instance that matches the query (where `Domain="DOMAIN_A"`). However, the performance improvement can vary from provider to provider. WMI providers do not have to support query optimization, in which case WMI calls `CreateInstanceEnumAsync` for a complete list of all the instances together with all the property values set. WMI then processes each instance and if the instance matches the query, WMI copies the instance and the required properties to the query enumerator that is output to the caller.
The bottom line is that the speed of queries is dependent on whether providers support query optimization in their implementation.

Performing queries is in some ways similar to enumeration. When we dealt with enumeration earlier in the chapter, you learned that you could get all of the instances of a specific class and access the instances through an enumerator (IEnumWbemClassObject). When performing queries, you still end up with an enumerator that allows you to enumerate all of the instances that match your query.

For more information on all the different types of queries that can be performed, review Chapter 3 and the Platform SDK documentation.

The following code sample illustrates how to obtain a list of Windows Services that have been started:

```c++
CComBSTR bstrQuery("SELECT * FROM Win32_Service WHERE Started=true");

// Execute query
CComPtr<IEnumWbemClassObject> spEnumInst;
hr = spServices->ExecQuery(CComBSTR(_T("WQL")), bstrQuery,
    WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY, NULL,
    &spEnumInst);

bool bFinished = false;
while (!bFinished)
{
    // Get the instance
    ULONG uNumOfInstances = 0;
    CComPtr<IWbemClassObject> spInstance;
    HRESULT hrNext = spEnumInst->Next(10000, 1, &spInstance,
        &uNumOfInstances);

    if (hrNext == WBEM_S_FALSE)
    {
        bFinished = true;
    } else if (hrNext == WBEM_S_NO_ERROR)
    {
        // Get properties from the object
        CComVariant varName;
        hr = spInstance->Get(CComBSTR(_T("DisplayName")), 0, &varName, 0, 0);
        _tprintf(_T(" %ls\n"), V_BSTR(&varName));
    }
}
```

In this code sample, the `ExecQuery` call performs the query (if it is valid) and creates an enumerator that can be used to gain access to every instance that matches the query. The enumerator is returned in an IEnumWbemClassObject interface. It does a quick check to ensure that there are more instances to retrieve, and if there are, some output (the display name of the service) is made.
The `ExecQuery` method looks like this:

```c
interface IWbemServices
{
    HRESULT ExecQuery(
        const BSTR strQueryLanguage,          // Must be 'WQL'
        const BSTR strQuery,                  // The WQL query to perform
        LONG lFlags,                         // How to perform the query
        IWbemContext *pCtx,                  // Used in providers
        IEnumWbemClassObject **ppEnum);      // Results from query
};
```

The `strQueryLanguage` parameter specifies the type of query to perform. The current implementation of WMI supports only “WQL”.

The `strQuery` parameter is the query that you wish to perform. If the query is successful, you will get an enumerator in the `ppEnum` out-parameter that you can use to gain access to each management object. If the query is invalid or the class specified is not found, `ExecQuery` will return `WBEM_E_INVALID_QUERY` or `WBEM_E_NOT_FOUND`, respectively.

Several flags can be specified in the `lFlags` parameter. `WBEM_FLAG_FORWARD_ONLY`, which produces a forward-only enumerator, is one of them.

If the query you are processing has a very large collection of instances, `WBEM_E_OUT_OF_MEMORY` may be returned by `ExecQuery`. In this case, you must use the asynchronous version, `ExecQueryAsync`.

As mentioned earlier, WMI has some system-provided properties that are automatically included in every object. These properties are `__SUPERCLASS`, `__DYNASTY`, `__RELPATH`, `__DERIVATION`, `__SERVER`, `__NAMESPACE`, and `__PATH`. The meaning and purpose of these properties are discussed in Chapter 3. We mention these now because, depending on the type of query you perform, these system-provided properties may or may not be set.

If you perform any query that starts with `SELECT *`, you can be assured that all the system-provided properties will be available for you, as in the following case:

```sql
SELECT * FROM Win32_UserAccount WHERE Domain="DOMAIN_A"
```

If you perform any query that specifies specific properties in the `SELECT` statement, then the system-provided properties will not be available, as in the following case:

```sql
SELECT FullName, SID FROM Win32_UserAccount WHERE Domain="DOMAIN_A"
```

All is not lost; you can instruct WMI to include the system-provided properties that can identify the location of a management object. To do this, you need to specify `WBEM_FLAG_ENSURE_LOCATABLE` when calling `ExecQuery`. The resulting instances in the
returned enumerator will include __RELPATH, __SERVER, __NAMESPACE, and __PATH.¹

There are occasions when you depend on the system-provided properties being available. The MMC snap-in in Chapter 9 will demonstrate the use of system-provided properties when creating and updating management objects in the user interface.

Microsoft Systems Management Server implements an extended version of WQL called Extended WQL. Extended WQL offers additional SELECT clauses such as DISTINCT, JOIN, and DATEPART. The WBEM_FLAG_PROTOTYPE allows you to obtain a class definition in the result set that uses the JOIN clause.

### Making Method Calls

So far in this chapter, you have seen a few occasions in which you might want or need to make method calls. For instance, you saw that we could have called methods for the creation and deletion of our Sample_Book class. Other examples include changing the configuration of shared directories and Windows Services (as well as starting and stopping them). You may discover in your own schema designs that you need to add behavior to your classes, and in this case methods are the answer. Although we see methods for creating, changing, and deleting instances, we urge you to use the standard mechanisms provided by WMI for these types of operations where possible. However, exceptions are appropriate when you find yourself in one or more of the following situations:

1. You may want to have specific flags passed into the method for more context information when creating and deleting instances.
2. You may want to receive one or more out-parameters other than the return value.
3. None of the WMI HRESULTs accommodate the types of errors you’d like to return. You can define the method return value so that you can return an appropriate data type for the errors you would like to pass back to the caller.
4. You may want to create multiple instances at the same time.

Calling methods fall into two categories:

1. Calling static methods on classes
2. Calling methods on objects

¹ Note that you can achieve the same result by specifying the system properties in the SELECT statement, although it is much easier simply to use the WBEM_FLAG_ENSURE_LOCATABLE flag.
To illustrate making method calls in detail, we shall use a fictitious class with a method called \( \text{foo} \) and decorate the method with different parameters so that you can gain a clear understanding. Let us look at this declaration:

```csharp
class Sample_Class
{
    [key] string Name;
    [static, implemented] void foo();
};
```

The class has a static method that takes no parameters and returns \( \text{void} \) (effectively meaning no return value). So let us see what code is needed to make this method call:

```csharp
hr = spServices->ExecMethod(CComBSTR("Sample_Class"), CComBSTR("foo"), 0, NULL, NULL, NULL, NULL);
```

That's it! The first parameter provides a class or object context of the method you would like to call. In the above case, we passed a class reference, \( \text{Sample}_\text{Class} \). The second parameter is the name of the method you want to call—in this case, \( \text{foo} \). Let us revise our method declaration:

```csharp
class Sample_Class
{
    [key] string Name;
    [implemented] string foo();
};
```

Not much difference. The only change we made was the removal of the \text{static} qualifier. The \text{static} qualifier meant that the method could be called without reference to any instances of the class. This is exactly what happened in the code sample: We passed a class reference. With the above change to \( \text{foo} \), we must now use an object reference. Let us look at the following code:

```csharp
hr = spServices->ExecMethod(CComBSTR("Sample_Class.Name=""Gwyn""), CComBSTR("foo"), 0, NULL, NULL, NULL, NULL);
```

Simple. As you can see, an object reference was passed instead. Unfortunately, methods that take no parameters and have no return value are not practical for most purposes. So let's extend \( \text{foo} \) to return a value:

```csharp
class Sample_Class
{
    [key] string Name;
    [implemented] string foo();
};
```
From a code perspective, things start getting more involved. We now need to obtain the return value from the `ExecMethod` call. This is the purpose of the sixth parameter, which returns an `IWbemClassObject` interface. But, you’re probably asking yourself, isn’t the `IWbemClassObject` interface used for representing object instances? The best way to visualize the `IWbemClassObject` interface is as a container for useful information. All method calls use this interface to pass back the method out-parameters as well as the return value. The return value exists in a property called `ReturnValue`, hence none of your method out-parameters can use this name. Let’s see some code that retrieves our string return value on `foo`:

```cpp
CComBSTR bstrMethod("foo");
CComBSTR bstrObject("Sample_Class.Name="Gwyn"");

CComPtr<IWbemClassObject> spOutParams;
hr = spServices->ExecMethod(bstrObject, bstrMethod, 0, NULL, NULL, &spOutParams, NULL);

// Inspect out-parameters for return value
CComVariant varReturnValue;
hr = spOutParams->Get(L"ReturnValue", 0, &varReturnValue, 0, 0);
_tprintf(_T("%ls::%ls returned %ls\n"), bstrObject, bstrMethod, V_BSTR(&varReturnValue));
```

The above code now has all the information you will need to make the `Delete` method call on the `Win32_Share` class. Finally, let’s make our `foo` method take an in-and out-parameter.

```cpp
class Sample_Class
{
    [key] string Name;
    [implemented] string foo([ in] uint32 inparam, [ out] boolean outparam);
};
```

Developing code to pass method in-parameters is significantly more involved. To prepare the method’s in-parameters requires access to the method declaration (made from the class definition). You will need to spawn a brand new instance of the method in-parameter definition so that parameter values can be set. Finally, the in-parameters are passed to the `ExecMethod` call and the provider will extract the parameters for the operation. Let’s see some code:

```cpp
CComBSTR bstrMethod("foo");
CComBSTR bstrObject("Sample_Class.Name="Gwyn"");
CComBSTR bstrClass("Sample_Class");
```
// Get the class definition so we can get access to the method in-parameters
CComPtr<IWbemClassObject> spClass;
hr = spServices->GetObject(bstrClass, 0, NULL, &spClass, NULL);

// Get the methods in-parameters
CComPtr<IWbemClassObject> spInParamsDefinition;
hr = spClass->GetMethod(bstrMethod, 0, &spInParamsDefinition, NULL);

// Spawn an instance of the in-parameters for our use
CComPtr<IWbemClassObject> spInParams;
hr = spInParamsDefinition->SpawnInstance(0, &spInParams);

// Setup required in-parameters for method
CComVariant varInParam(int(12345));
hr = spInParams->Put(L"inparam", 0, &varInParam, 0);

// Execute method
CComPtr<IWbemClassObject> spOutParams;
hr = spServices->ExecMethod(bstrClass, bstrMethod, 0, NULL, spInParams,
&spOutParams, NULL);

// Inspect out-parameters for return value
CComVariant varReturnValue;
hr = spOutParams->Get(L"ReturnValue", 0, &varReturnValue, 0, 0);

// ReturnValue contained in V_BSTR(&varReturnValue)

// Inspect the methods out-parameters (other than the method return value)
CComVariant varOutParam;
hr = spOutParams->Get(L"outparam", 0, &varOutParam, 0, 0);

// outparam value contained in V_BOOL(&varOutParam)

As with out-parameters, all in-parameters are contained within an instance of the
IWbemClassObject interface. Each property corresponds to a method parameter, so if
the method you plan to call has six parameters, then you will have to set six properties
(assuming that all the parameters are required and are not optional).

To provide a more serious code sample and introduce some other issues that you
most likely will have to deal with, we shall create a shared directory with the Win32_
Share class. Here is an almost complete declaration of the Win32_Share class:

class Win32_Share: CIM_LogicalElement
{
    [read] string Name;
    [read] boolean AllowMaximum;
    [read] uint32 MaximumAllowed;
    [read] string Path;
    ...
Let's focus on the `Create` method. It has seven in-parameters, `Path`, `Name`, `Type`, `MaximumAllowed`, `Description`, `Password`, and `Access`. Each of these parameters should be straightforward to understand; if you look at the sharing property page for a directory, you will see the user interface that maps to these parameters. Let's discuss the `Create` method's parameter types and declarations in detail.

First, you will see that most parameters take standard types that you already recognize, such as `string` and `uint32`. The `Access` parameter is an exception: It passes a `Win32_SecurityDescriptor` object. Classes intended for use in method parameters usually are inherited from `Win32_MethodParameterClass`.

All parameter declarations use qualifiers to provide more context information. The `[in]` qualifier specifies that the parameter is an in-parameter and `[optional]` means that it is not required. The `ValueMap` qualifier specifies the values that are valid for the property or parameter. The full `ValueMap` declaration that follows shows various integers' values that are valid for the `Type` parameter.

```plaintext
ValueMap
{
  "0", "1", "2", "3",
  "2147483648", "2147483649", "2147483650", "2147483651"
}
```

What do these values mean or represent? That is the role of the `Values` qualifier, which provides a textual description of the values in the `ValueMap`. Here is the full `Values` declaration:

```plaintext
Values
{
  "Disk Drive", "Print Queue", "Device", "IPC",
  "Disk Drive Admin", "Print Queue Admin", "Device Admin", "IPC Admin"
}
```
Using the Values and ValueMap qualifiers, it is possible to work out what the Type parameter means or should be. The parameter is 0 (in integer form), specifies “Disk Drive” and 1 specifies “Print Queue” and so on. You should use these types of qualifiers in your own schema to aid the understanding of your method parameters and class properties.

The code that follows calls a static method to create a shared directory:

```c
CComBSTR bstrMethod("Create"); // Method to call
CComBSTR bstrClass("Win32_Share"); // The class the method belongs to

// Get the class definition so we can get access to the method from the class
CComPtr<IWbemClassObject> spClass;
hr = spServices->GetObject(bstrClass, 0, NULL, &spClass, NULL);

// Get the methods in-parameters
CComPtr<IWbemClassObject> spInParamsDefinition;
hr = spClass->GetMethod(bstrMethod, 0, &spInParamsDefinition, NULL);

// Spawn an instance of the in-parameters for our use
CComPtr<IWbemClassObject> spInParams;
hr = spInParamsDefinition->SpawnInstance(0, &spInParams);

// Setup required in-parameters for method
CComVariant varName("Temp");
hr = spInParams->Put(L"Name", 0, &varName, 0);

CComVariant varPath("C:\Temp");
hr = spInParams->Put(L"Path", 0, &varPath, 0);

CComVariant varType(int(0));
hr = spInParams->Put(L"Type", 0, &varType, 0);

// Execute method
CComPtr<IWbemClassObject> spOutParams;
hr = spServices->ExecMethod(bstrClass, bstrMethod, 0, NULL,
spInParams, &spOutParams, NULL);

// Inspect out-parameters for return value
CComVariant varReturnValue;
hr = spOutParams->Get(L"ReturnValue", 0, &varReturnValue, 0, 0);

_tprintf(_T("%ls::%ls returned %d\n"), bstrClass, bstrMethod, V_I4(&varReturnValue));
```
Most of the details were discussed in earlier code, so we will focus on the new information. We completed only the required parameters (that is, parameters that don’t have the optional attribute set) for two reasons: first, to keep the code sample small and second, we needed to fill in only the required parameters to create a shared directory. The Type parameter has a value of 0 (integer), if you recall, and this, according to the Values qualifier, will specify “Disk Drive”. We finally pass the in-parameters to the ExecMethod call and we get the out-parameters through \texttt{spOutParams}. If a method being called returns a value other than \texttt{void}, then you will always get an out-parameter named “ReturnValue”.

Let’s look at the \texttt{GetMethod} method call:

```c
interface IWbemClassObject
{
HRESULT GetMethod(
    LPCWSTR wszName,                    // Name of the method
    LONG lFlags,                        // Must be zero
    IWbemClassObject **ppInSignature,   // In-params definition
    IWbemClassObject **ppOutSignature); // Out-params definition
};
```

A key point to notice is that \texttt{GetMethod} can be called only if the \texttt{IWbemClassObject} instance represents the definition of a class. If this were an object, \texttt{GetMethod} would return a \texttt{WBEM E_ILLEGAL_OPERATION} error. Ultimately, \texttt{GetMethod} allows you to obtain a definition of both the in-parameters and the out-parameters of a method. The \texttt{wszName} parameter specifies the name of the method. Both the \texttt{ppInSignature} and \texttt{ppOutSignature} out-parameters contain the parameter definitions. If you are not interested in either \texttt{ppInSignature} or \texttt{ppOutSignature} then you can simply pass \texttt{NULL} to ignore the parameter. In the code sample above, we ignored \texttt{ppOutSignature}. The \texttt{lFlags} parameter is documented as having to be 0.

Next, the \texttt{ExecMethod} method looks like this:

```c
interface IWbemServices
{
HRESULT ExecMethod(
    const BSTR strObjectPath,        // The class or object ref
    const BSTR MethodName,           // The name of the method
    long lFlags,                     // How to make the call
    IWbemContext *pCtx,              // Used in providers
    IWbemClassObject *pInParams,     // The in-parameters
    IWbemClassObject **ppOutParams,  // The out-parameters
    IWbemCallResult **ppCallResult); // Not used for now...
};
```
As mentioned already, the \texttt{strObjectPath} parameter is either a class reference for static methods or an object reference for object methods.

The \texttt{lFlags} parameter allows only one flag, \texttt{WBEM\_FLAG\_RETURN\_IMMEDIATELY}, which you use when you want to execute a semisynchronous call. We discuss semisynchronous calls later in the chapter.

If a method has in-parameters, you must supply your in-parameters (as already discussed) in the \texttt{pInParams} parameter. If the method does not have any in-parameters, then you can simply pass NULL.

If a method has out-parameters, you will receive them through \texttt{ppOutParams} when \texttt{ExecMethod} returns. If you wish to ignore the out-parameters, you can simply pass NULL.