Sun WBEM SDK Developer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806–6831–10
April, 2001

040210@7940

Contents

Tables

Figures

Examples

Preface

The *Sun WBEM SDK Developer's Guide* describes the Sun WBEM Software Developer's Toolkit, that enables software developers to create standards-based applications that manage WBEM-enabled objects. Developers can also use this software to write providers, programs that communicate with managed objects to access data.

Who Should Use This Book

This book is intended for two types of developers:

- System and Network Application Developers

  Programmers who write applications that manage the information stored in CIM classes and instances will find this guide useful. These developers typically use the Sun WBEM APIs to get and set the properties of predefined CIM instances and classes.

- Instrumentation Engineers

  Instrumentation engineers provide resources such as processors, memory, routers, and other manageable devices. These developers need to communicate device information in a standard CIM format to the CIM Object Manager, typically through a piece of software called a provider. These users use the WBEM APIs to create classes, instances, and properties.

  Instrumentation engineers might work with class and schema designers, who describe new groups of managed resources (CIM classes) or a collection of CIM classes, called schema. Schemas describe the managed objects in a particular system environment, for example, Microsoft Windows 32 or the Solaris operating environment.

Before You Read This Book

This book describes how to use the Sun WBEM SDK components and tools to write management applications.

This book requires knowledge of the following:

- Object-oriented programming concepts
- Java programming
- A solid understanding of Common Information Model (CIM) concepts

If you are unfamiliar with these areas, you might find the following references useful:

- *Java™ How to Program*

  H. M. Deitel and P. J. Deitel, Prentice Hall, ISBN 0–13–263401–5
- *The Java Class Libraries, Second Edition, Volume 1*, Patrick Chan, Rosanna Lee, Douglas Kramer, Addison-Wesley, ISBN 0–201–31002–3
- *CIM Tutorial*, provided by the Distributed Management Task Force

The following Web sites are useful resources when working with WBEM technologies.

- Distributed Management Task Force (DMTF)

  See this site at www.dmtf.org for the latest developments on CIM, information about various working groups, and contact information for extending the CIM Schema.
- Rational Software

  See this site at www.rational.com/uml for documentation on the Unified Modeling Language (UML) and the Rose CASE tool.

How This Book Is Organized

Chapter 1 introduces Web-Based Enterprise Management (WBEM), the Common Information Model (CIM), Sun WBEM SDK, and Solaris WBEM Services.

Chapter 2 describes how to use CIM WorkShop to manipulate CIM classes, instances, methods, and properties.

Chapter 3 provides an overview of the client APIs and examples of how to use them to create and manipulate CIM objects.

Chapter 4 explains how to use the Client APIs to write client applications.

Chapter 5 provides an overview of the provider APIs and explains how to write a provider, classes that mediate between managed objects and the CIM Object Manager.

Chapter 6 This chapter describes the CIM event model and explains how providers generate CIM events and how applications subscribe to be notified of the occurrence of CIM events.

Chapter 7 explains how to run the code examples provided with the Sun WBEM SDK.

Chapter 8 explains error messages returned by Sun WBEM SDK APIs.

Appendix A describes general CIM terms and concepts.

Glossary presents a list of words and phrases found in this book and their definitions.

Related Information

The *Solaris WBEM Services Administrator's Guide* explains Common Information Model (CIM) concepts and describes how to administer Web-based Enterprise Management (WBEM) services in the Solaris™ operating environment.

See the online Javadoc reference pages for the WBEM Application Programming Interfaces in `/usr/sadm/lib/wbem/doc/index.html`. Also see the online Javadoc reference pages for the CIM and Solaris Schema classes in `/usr/sadm/lib/wbem/doc/mofhtml`.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com[SM] Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Overview of WBEM

This chapter provides a detailed description of Web-Based Enterprise Management (WBEM) . The following topics are covered.

- About WBEM
- Common Information Model
- Managed Object Format
- CIM and Solaris

About WBEM

Web-Based Enterprise Management (WBEM) is an initiative and a technology. As an initiative, WBEM includes standards for managing systems, networks, users, and applications by using Internet technology. As a technology, WBEM provides a way for management applications to share management data independently of vendor, protocol, operating system, or management standard. By developing management applications according to WBEM principles, vendors can develop products that work together easily for low cost of development.

The Distributed Management Task Force (DMTF), a group representing corporations in the computer and telecommunications industries, is leading the effort to develop and disseminate standards for management of desktop environments, enterprise-wide systems, and the Internet. The goal of the DMTF is to develop an integrated approach to managing networks across platforms and protocols, resulting in cost-effective products that interoperate as flawlessly as possible. For information about DMTF initiatives and outcomes, see the DMTF web site at www.dmtf.org.

Common Information Model

The Common Information Model (CIM) is an approach to managing systems and networks. CIM provides a common conceptual framework to classify and define the parts of a networked environment and depict how they integrate. The model captures notions that are applicable to all areas of management, independent of technology implementation.

CIM Terminology

The Common Information Model uses a set of terminology specific to the model and the principles of object-oriented programming. For information about CIM terminology and descriptions of what the terms represent, see Appendix A. See the Glossary for an expanded list of terms that have a specialized meaning in CIM.

CIM Structure

The Common Information Model categorizes information from general to specific. Specific information, such as a representation of the Solaris environment, extends the model. CIM consists of the following three layers of information:

- Core Model – A subset of CIM not specific to any platform.
- Common Model – Information model that visually depicts concepts, functionality, and representations of entities related to specific areas of network management, such as systems, devices, and applications.
- Extensions – Information models that support the CIM Schema and represent a very specific platform, protocol, or corporate brand.

Collectively, the Core Model and the Common Model are referred to as the CIM Schema.

The Core Model

The Core Model provides the underlying, general assumptions of the managed environment—for example, that specific, requested data must be contained in a location and distributed to requesting applications or users. These assumptions are conveyed as a set of classes and associations that conceptually form the basis of the managed environment. The Core Model is meant to introduce uniformity across schemas intended to represent specific aspects of the managed environment.

For applications developers, the Core Model provides a set of classes, associations, and properties that can be used as a starting point to describe managed systems and determine how to extend the Common Model. The Core Model establishes a conceptual framework for modeling the rest of the managed environment.

The Core Model provides classes and associations to extend specific information about systems, applications, networks, devices, and other network features through the Common Model and extensions. For information about the system aspects of the Core Model and related classes and associations, see Appendix A.

The Common Model

Areas of network management depicted in the Common Model are independent of a specific technology or implementation but provide the basis for the development of management applications. This model provides a set of base classes for extension into the area of five designated technology-specific schemas: Systems, Devices, Applications, Networks, and Physical.

CIM Extensions

Extension schemas are built into CIM to connect specific technologies into the model. By extending CIM, a specific operating environment such as Solaris can be made available to a greater number of users and administrators. Extension schemas provide classes for software developers to build applications that manage and administer the extended technology.

Managed Object Format

MOF is the standard language used to define elements of the Common Information Model (CIM). The MOF language specifies a syntax for defining CIM classes and instances. MOF provides developers and administrators with a simple and fast technique for modifying the CIM Repository. For more information about MOF, see the DMTF web page at http://www.dmtf.org.

Because MOF can be converted to Java, an application developed in MOF can be run on any system or in any environment that supports Java.

The MOF Syntax

Programmers can use the CIM API to represent CIM objects, developed in MOF, as Java classes. The CIM Object Manager checks and enforces that these CIM objects comply with the CIM 2.1 Specification. In some cases, it is possible to represent something syntactically correct in a MOF file that does not adhere to the CIM specification. The CIM Object Manager returns an error message when such a MOF file is compiled.

For example, if you specify scope in the qualifier definition in a MOF file, CIM Object Manager returns a compilation error because scope can only be specified in the definition of a CIM Qualifier Type. A CIM Qualifier cannot change the scope that was specified in the CIM Qualifier Type.

Schema MOF Files

The installation of Solaris WBEM Services puts MOF files that form the CIM Schema and the Solaris Schema in the directory `/usr/sadm/mof`. These files are automatically compiled and run when the CIM Object Manager starts.

The CIM Schema files, denoted by CIM in the file name, form standard CIM objects. For an explanation of the parts that make up the CIM Schema, see version 2.1 of the CIM Specification, which can be obtained at http://dmtf.org/spec/cims.html.

The Solaris Schema describes Solaris objects by extending the standard CIM Schema. The MOF files that make up the Solaris Schema use the Solaris prefix in the file names, but otherwise follow the same file name conventions as the CIM Schema MOF files. You can view the MOF files that make up the Solaris Schema in a text editor of your choice.

CIM and Solaris

Sun Microsystems, Inc. extends CIM principles and classes in the Solaris operating environment. Developed in Java to run on any Java-enabled platform, the Sun implementation consists of two products: the Sun WBEM SDK and Solaris WBEM Services.

Sun WBEM SDK

The Sun WBEM Software Development Kit (SDK) contains the components required to write management applications that can communicate with any WBEM-enabled management device. Developers can also use this tool kit to write providers, programs that communicate with managed objects to access data. All management applications developed using Sun WBEM SDK run on the Java platform. .

The Sun WBEM SDK installs and runs in any Java environment. It may be used as a standalone application or with Solaris WBEM Services.

Solaris WBEM Services

Solaris WBEM Services provides routing and security services. The CIM Object Manager routes data about objects and events between components. Sun WBEM User Manager is an application with a graphical user interface in which you can set user permissions to specific work areas. For information about Solaris WBEM Services and components, see the *Solaris WBEM Services Administrator's Guide*.

CHAPTER 2

CIM WorkShop

This chapter explains how to use CIM WorkShop to add new properties, methods, and qualifiers to the classes and instances that you create, and to set the scope and flavor of new qualifiers for new classes and instances. The following topics are covered:

- About CIM WorkShop
- Starting CIM WorkShop
- Navigating in CIM WorkShop
- Viewing Class Characteristics
- Working in Namespaces
- Working with Classes
- Adding a Class
- Deleting Classes and Their Attributes
- Working with Instances
- Invoking Methods
- Reference: CIM WorkShop Windows and Dialog Boxes

About CIM WorkShop

CIM WorkShop provides a graphical user interface in which you can view and create classes and instances. In CIM WorkShop, you can complete any of the following tasks:

- View and select namespaces
- Add namespaces
- View and create classes
- Add properties, qualifiers, and methods to new classes
- View and create instances
- View and modify instance values

Note – CIM guidelines prevent you from modifying or editing the properties, methods, or qualifiers of CIM Schema or Solaris Schema classes. However, you can create new classes and instances of classes. When you create a new class or instance, you can add or delete properties, methods, and qualifiers. You can also change the values, including the scope and flavor, of new qualifiers that you create for a new class, instance, property, or method. You cannot change the values of inherited properties, methods, or qualifiers.

Starting CIM WorkShop

CIM WorkShop is available as part of the Sun WBEM SDK.

The CIM Object Manager must be installed to run CIM WorkShop. During an installation of Solaris WBEM Services in the Solaris operating environment, the CIM Object Manager runs on the local host. If you install only the Sun WBEM SDK, you must point to a host on which the CIM Object Manager already has started. You can enter this information in the Host field of the Login dialog box that is displayed when you start CIM WorkShop. For information about the CIM WorkShop dialog boxes and fields, see "Reference: CIM WorkShop Window and Dialogs" on page 43.

▼ How to Start CIM WorkShop

1. **Start the CIM WorkShop:**

   - Type the following command at the system prompt:

     ```
     % /usr/sadm/bin/cimworkshop
     ```

   The CIM WorkShop window is displayed, followed by the Login dialog box. The Login dialog box shows the name of the host computer on which CIM Workshop is installed and the path of the default namespace, `root\cimv2`. Context Help, information about how to complete the dialog box, is displayed on the left side of the Login dialog box. When you click a field, the help content changes to reflect how to enter information into the field and the meaning of the field.

2. **In the CIM WorkShop login dialog box, do the following:**

   - In the Host Name field, type the name of a host running the CIM Object Manager.

Note – By default, CIM WorkShop connects to the CIM Object Manager on the local host, in the default namespace, `root\cimv2`. If you start CIM WorkShop as part of the WBEM SDK in the Solaris operating environment or in the Microsoft Windows environment, you need to provide the name of a host that is already running a CIM Object Manager.

- In the Namespace field, click in the field and type the name of the namespace that you want to use, or retain the name of the default namespace.
- In the User Name field, type the user name you generally use for system and networking privileges.
- In the Password field, type the password you generally use for system and networking privileges.

Note – If you do not specify a user name and password, you can log in using the default user account, *guest*. Guest privileges are read-only. Your CIM Object Manager administrator can set up write privileges associated with your user name and password.

- By default, CIM WorkShop uses the RMI protocol to connect to the CIM Object Manager on the local host, in the default namespace, `root\cimv2`. You can select HTTP if you want to communicate to a CIM Object Manager using the standard XML/HTTP protocol from the Desktop Management Task Force.

3. **Click OK.**

   A message is displayed to show that the classes in the class inheritance tree are being enumerated. In the left side of the CIM WorkShop window, CIM classes are displayed.

Navigating in CIM WorkShop

When you first start CIM WorkShop, the classes of the CIM Schema display hierarchically in the left side of the CIM WorkShop window. This arrangement of classes is referred to as the class inheritance tree. When you select a class, its associated properties are listed in the right side of the window. In the following illustration, the properties of the class `Solaris_ComputerSystem` are listed in the right side of the CIM WorkShop window.

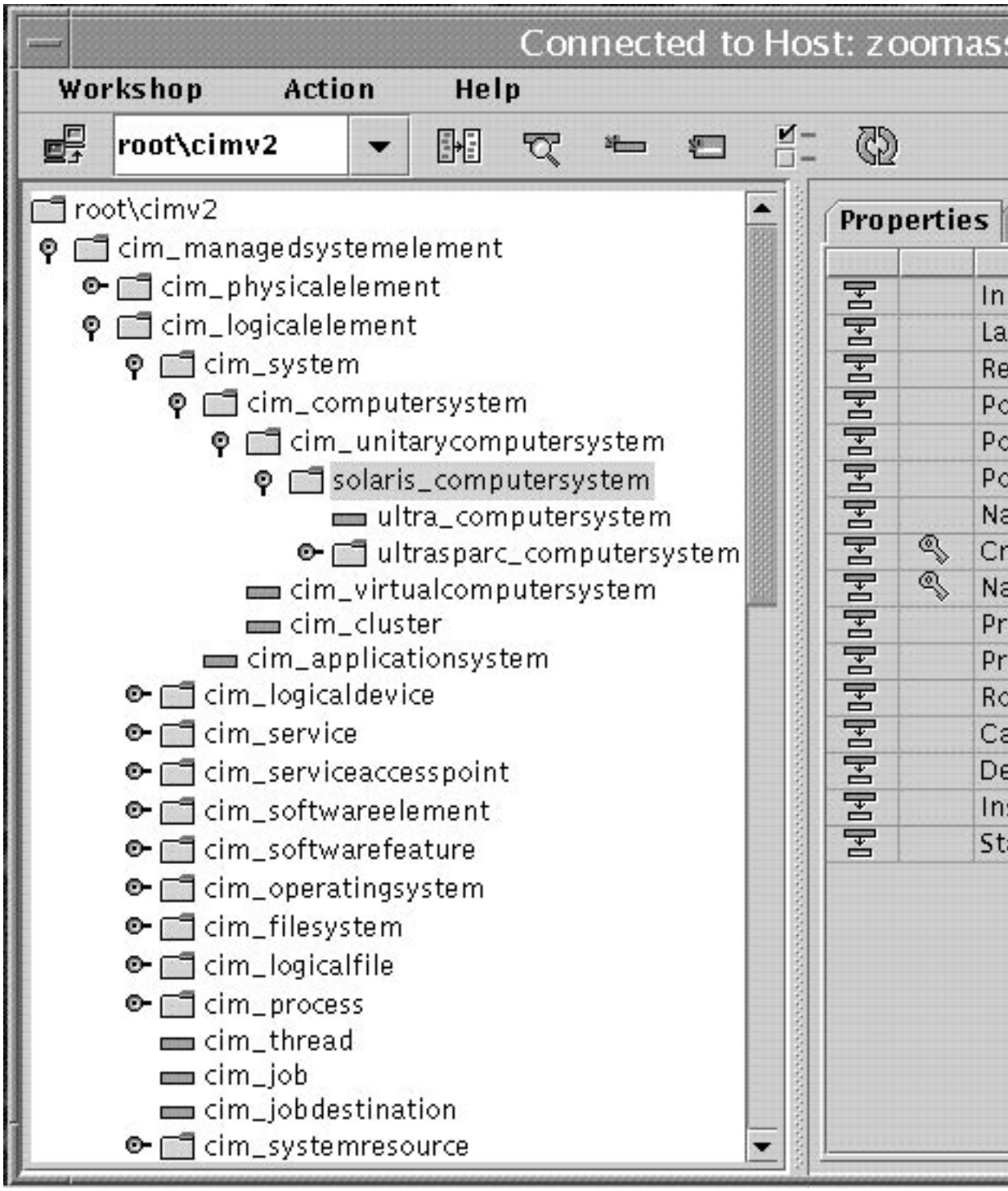
Connected to Host: zoomass
Workshop
Action
Help
root\cimv2
root\cimv2
cim_managedsystemelement
cim_physicalelement
cim_logicalelement
cim_system
cim_computersystem
cim_unitarycomputersystem
solaris_computersystem
ultra_computersystem
ultrasparc_computersystem
cim_virtualcomputersystem
cim_cluster
cim_applicationsystem
cim_logicaldevice
cim_service
cim_serviceaccesspoint
cim_softwareelement
cim_softwarefeature
cim_operatingsystem
cim_filesystem
cim_logicalfile
cim_process
cim_thread
cim_job
cim_jobdestination
cim_systemresource
Properties

FIGURE 2–1 Class Inheritance Tree in CIM WorkShop Window

For information about the toolbar, menus, and layout of the CIM WorkShop window, see "Reference: CIM WorkShop Window and Dialogs" on page 43.

Browsing the Class Inheritance Tree

Each class that has classes is denoted by two icons: a folder icon and an enabler icon. The enabler icon looks like a small key to the left of the folder icon.

The folder icon indicates that the class serves as a container for the classes it contains. The enabler icon serves as a navigation aid.

When an enabler icon is displayed horizontally, the class folder is closed and classes are contained. When you click the enabler icon, the class folder opens and classes are revealed. When the enabler icon is displayed vertically, it indicates that the class folder is open.

▼ How To Display the Contents of a Class

- **Click the enabler icon of the desired class to view its contents.**

▼ How to Display the Properties and Methods of a Class

- **Click the class folder icon of the class.**

  The properties and methods of the class are displayed in the right frame of the CIM WorkShop window.

Finding a Class

CIM WorkShop enables you to quickly find a specific class.

▼ How to Find a Class

1. **In the toolbar, click the Find Class icon.**
2. **In the Find Class dialog box, type the name of the class you want to find and click OK.**

   When the specified class is found, its details are displayed in the right frame of the CIM WorkShop window.

Viewing Class Characteristics

When you select a class from the class inheritance tree—by clicking its folder icon—two tabs, indicating the properties and methods of the class, are displayed in the right side of the CIM WorkShop window.

Selecting a Class

In the class inheritance tree, classes that contain classes are indicated by folder icons. Classes that do not contain classes are indicated by purple rectangles. Select a class by clicking the folder or page icon of the class in the class inheritance tree.

Viewing Class Properties

By default, when the CIM WorkShop window is displayed, the Properties tab appears in the right side of the CIM WorkShop window. In the left side of the CIM WorkShop window, you can select a class from the class inheritance tree; you can view all properties of the class in the Properties tab. Inherited properties are indicated by an icon that consists of a purple rectangle with a black arrow pointing to a white rectangle. Properties that have an assigned key qualifier are indicated by a gold key icon. For information about the presentation of properties in the Properties tab, see "The Properties Tab" on page 46.

Viewing Class Methods

After you select a class in the class inheritance tree, you can click the Methods tab to display the methods associated with the class. For information about how the methods are displayed in the methods tab, see "The Methods Tab" on page 46.

Viewing Qualifiers

In CIM, qualifiers are attributes of classes, instances, properties, and methods. In CIM Workshop, you can view the qualifiers by right-clicking a class, property, or method and clicking Qualifiers in the pop-up menu. Clicking Qualifiers causes the Qualifiers dialog box to be displayed. For information about the presentation of qualifier information in the Qualifiers dialog box, see "Qualifiers Dialog Box" on page 51.

Viewing the Scope of a Qualifier

When you click the Scope button in the Qualifiers dialog box, the Scope dialog box is displayed. In the Scope dialog box, you can view the scope of a qualifier. For information about the Scope dialog box, see "Scope Dialog Box" on page 51.

Viewing the Flavor of a Qualifier

When you click the Flavor button in the Qualifiers dialog box, the Flavors dialog box is displayed. In the Flavors dialog box, you can view the flavor of a qualifier. For information about the Flavors dialog box, see "Flavors Dialog Box" on page 51.

Working in Namespaces

A namespace is a logical entity, an abstraction of a managed object into which classes and instances can be stored. A namespace can be implemented in various forms including a directory structure, a database, or a folder. By default, CIM WorkShop connects to the CIM Object Manager on the local host, in the default namespace `root\cimv2`. All classes contained in the default namespace are displayed in the left side of the CIM WorkShop window. The name of the current namespace is listed in the toolbar of the CIM WorkShop window. In CIM WorkShop, you can browse the classes of namespaces on different hosts and you can change location to new namespaces.

When you want to set user privileges for a particular namespace, use the Sun WBEM User Manager. For information about the Sun WBEM User Manager tool, see "Administering Security" in *Solaris WBEM Services Administrator's Guide*.

This section describes how to:

- Create a namespace
- Change to a namespace
- Change to a host
- Refresh the class inheritance tree of the namespace

Creating a Namespace

You can create one or more namespaces within an existing namespace.

▼ How to Create a Namespace

1. **Select Change Namepace from the Workshop menu in the main CIM WorkShop window.**

2. **Right click on the namespace in which you want to create the new namespace, and then select Add Namespace.**

3. **In the Input dialog box, type the name of the new namespace and click OK.**

Changing Namespaces

In the Sun WBEM SDK, the default namespace is `root\cimv2`. You can change to any other namespace.

▼ How to Change Namespaces

1. **In the CIM WorkShop window, click Workshop->Change Namespace.**

2. **In the Change Namespace dialog box, click the icon of the namespace you want to use. Click OK.**

   The namespace you have selected becomes the current namespace.

Changing Hosts

You can change to another host to view namespaces or processes.

▼ How to Change Hosts

1. **Click Workshop->Change Host or click the Change Hosts icon in the CIM WorkShop toolbar.**

2. **In the Host Name field, type the name of the host on which the namespace you want to view is located.**

3. **Type your user name and password in the User Name and Password fields, respectively.**

4. **Click OK.**

Refreshing Classes and Namespaces

You can refresh the display of the class inheritance tree in the namespace to reflect current changes made by other users who work in the namespace.

▼ How to Refresh a Class Inheritance Tree

1. **In the class inheritance tree, click the folder of the class you want to refresh.**

2. **Click Action->Refresh or click the Refresh Selected Class icon in the CIM WorkShop toolbar.**

Working with Classes

Classes are the building blocks of applications. When you start CIM WorkShop, it becomes populated with the classes that make up the CIM and Solaris Schemas. These classes adhere to the Distributed Management Task Force standards. Their unique properties, methods, and qualifier values cannot be changed.

To set new values for an existing class, you can create a new instance or class. The CIM and Solaris Schema classes serve as templates. When you create a new instance or class, you produce a copy of the selected class in which you can add new properties, methods, and qualifier values. In this way, you build your own extensions into the CIM or Solaris Schemas.

Note – You cannot modify the values of inherited properties, methods, or qualifiers.

For information about how to create an instance, see "Working with Instances" on page 40. For information about how to create a class, see the following section.

Adding a Class

Adding a class to an existing class involves the following tasks:

- Selecting the class
- Creating a new class
- Adding new qualifiers to the class
- Adding new properties to the class
- Adding new qualifiers to the properties
- Setting qualifier values: scope and flavor

Creating a Class

The first step in creating a class is to specify a name for the class. In CIM WorkShop, class names are displayed using standard CIM syntax: *SchemaIndicator_ClassName*. If you create a class of a CIM Schema class, the acronym *CIM* is used before the class name. If you create a class of a Solaris Schema class, the name *Solaris* is used before the class name. The underscore character (_) is required in the name of all classes that inherit a Key qualifier.

▼ How to Add a class

1. **In the class inheritance tree of the CIM WorkShop window, select the class from which to create a class.**
2. **Choose one of the following procedures for creating a class:**
   - Click Action->Add Class.

     or
   - Click the Add New Class icon in the toolbar of the CIM WorkShop window.

     or
   - Right-click the selected class and click Add Class.

     The New Class dialog box is displayed.
3. **In the Class Name field, type the name of the new class.**

   For example, you can create a class from the class `Solaris_ComputerSystem` titled `Ultra1_ComputerSystem`.
4. **To retain inherited properties and methods of the class, click OK. To add new properties, click Add Property.**

   If you click OK, a class is created that uses inherited properties, methods, qualifiers, and their values. If you click Add Property, the Add Property dialog box is displayed, in which you can specify properties to add to the class. For information about how to add properties to a class, see "Adding New Properties to a Class" on page 37.

Adding Qualifiers

You can add qualifiers to a new class. You cannot change or reset the values of inherited qualifiers that modify the class. Also, you cannot delete inherited qualifiers.

▼ How to Add Qualifiers

1. **In the New Class dialog box, after you provide a name for the new class, click Class Qualifiers.**

2. **In the Qualifiers dialog box, right-click the Qualifier for which you want to set new values and click Add Qualifier.**
3. **In the Add Qualifier dialog box, select the name of a qualifier in the list and click OK.**
4. **To set the scope of the qualifier:**
   a. **Click Scope.**
   b. **In the Scope dialog box, select the scope of the qualifier and click OK.**
5. **To set the flavor of the qualifier:**
   a. **Click Flavors.**
   b. **In the Flavors dialog box, select the flavor of the qualifier and click OK.**
6. **Click OK in the Qualifiers dialog box to close it.**

Adding New Properties to a Class

You can add new properties to a class and modify their values. You cannot change the values of inherited properties, and you cannot delete inherited properties.

▼ How to Add a New Property to a Class

1. **After specifying a name for the new class, click Add Property in the New Class dialog box.**
   The Add Property dialog box is displayed.
2. **In the Name field, type the name of the new property.**
3. **Select a property type from the Type field and click OK.**
   The new property is displayed in the Properties tab of the New Class dialog box. If the list of properties is long, click the scroll bar to view the newly added property.
4. **Click OK in the New Class dialog box.**
   For information about how to add new qualifiers or set qualifier values for a new property or class, see the following sections.

Adding Qualifiers to a New Property

You can set the values of qualifiers for new properties of the class. You cannot change or reset the values of qualifiers that modify inherited properties or methods. You cannot delete inherited qualifiers.

▼ How to Add Qualifiers to a New Property

1. **In the New Class dialog box, click the new property you created and click Property Qualifiers.**
   The Qualifiers dialog box is displayed for the property that you created.
2. **Click Add Qualifier.**
3. **In the Name field of the Add Qualifier dialog box, select a qualifier and click OK.**
4. **Click OK in the Qualifiers dialog box and in the New Class dialog box.**
   The qualifier and qualifier type are set for the selected property.

Deleting Classes and Their Attributes

CIM WorkShop provides a way to delete classes, properties, methods, and qualifiers that you no longer need or use.

Note – When you delete a class, you delete all subclasses it contains. You also delete all associated properties, methods, and qualifiers of the class and its subclasses.

Deleting a Class

Use the following procedure to delete a class from the class inheritance tree.

▼ How to Delete a Class

1. **Select the class that you want to delete.**
2. **Click Action->Delete Class and click OK in the dialog box that asks you to confirm your decision to delete a class.**
   The class is deleted.

Deleting a Property of a Class

You can delete only a property that you created in a new class. Otherwise, you can view but not modify or delete properties of classes. You cannot delete an inherited property in a subclass. However, when you create a new class, you can delete any new properties added to the class. For information about how to create a class, see "Adding a Class" on page 35.

▼ How to Delete a Property of a Class

- **In the Properties tab of the New Class dialog box, select the property name or type and click Delete Property.**

Deleting Qualifiers

When you create a new class, you can delete qualifiers of properties or methods inherited from the parent class. For information about how to create a class, see "Adding a Class" on page 35.

▼ How to Delete a Qualifier of a Property

1. **In the Properties tab of the New Class dialog box, select the property that you want to delete.**
2. **Click Property Qualfiers.**
3. **In the Qualifiers dialog box, select the qualifier to delete, Sclick Delete Qualifier, then click OK.**

   The selected qualifier is deleted. The New Class dialog box is displayed.

▼ How to Delete a Qualifier of a Method

1. **In the Methods tab of the New Class dialog box, right-click the method with the qualifier you want to delete.**
2. **Click Qualifiers in the pop-up menu.**
3. **Select the qualifier you want to delete.**
4. **Right click the qualifier and select Delete Qualifier or select the qualifier to delete and click the Delete Qualifier button.**
5. **Select Qualifier to list the qualifiers for this method.**

▼ How to Add a Qualifier to a Method

1. **In the Methods tab of the New Class dialog box, right-click the method to add a qualifier to and then select Qualifiers.**
2. **Click the Add Qualifier button in the Qualifiers dialog box.**
3. **Select the name of the qualifier type to add and then click OK. Click OK to close the Qualifiers dialog box.**

Working with Instances

In CIM WorkShop, you can create instances of classes. Instances inherit the characteristics of the class. You can then change the attributes of a new instance to create a unique instance of a class.

Displaying Instances

Before you create a new instance of a class, it is useful to view the instances of the class to see what properties and methods they contain.

▼ How to Display Instances of an Existing Class

1. **In the class inheritance tree of the CIM WorkShop window, select the class for which you want to view instances.**
2. **To display the Instances window:**
   - Click Action->Instances->Deep Enumeration or Shallow Enumeration.

     or
   - Click the Show Instances icon on the CIM WorkShop toolbar.

     or
   - Right click a class and select either Deep Enumeration or Shallow Enumeration from the pop-up dialog box.

The Instances window is displayed. If the selected class has instances, the instances are displayed in the left frame of the Instances window. If the selected class does not have instances, the left frame of the Instances window is empty.

Adding Instances

Add instances to a class when you want to modify the inherited qualities of objects.

▼ How to Add an Instance to a Class

1. **In the CIM WorkShop window:**
   - Click Action->Instances->Deep Enumeration to list the instances of the current class and all its descendant classes.

     or
   - Click Action->Instances->Shallow Enumeration to list the instances of the current class.

     The Instances window is displayed. All instances of the class are displayed in the left side of the window.
2. **Right-click an instance listed in the Instances window.**

   The Add Instances dialog box is displayed with options to Refresh, Add Instance, or Delete Instance.
3. **Click Add Instance.**
4. **To modify the inherited properties of an instance:**
   a. **Click in the value field to be changed.**

      A dialog box is displayed in which you can provide a value for the property. The dialog box displayed varies depending on the type of the selected property. For example, if you select a property that has a type `string`, the String dialog box will display. The Value field of this dialog box accepts only character strings.

   b. **In the Value field of the dialog box, type the required value.**
5. **Click OK to close the Add Instances window.**

Deleting Instances

You can delete an instance that you no longer use.

▼ How to Delete an Instance

1. **In the left frame of the CIM WorkShop window, right-click the class from which you want to delete an instance.**
2. **In the pop-up menu, click Instances->Deep Enumeration to list instances of the selected class and its subclasses. Click Instances->Shallow Enumeration ot list**

instances of the selected class.

3. **In the Instance window, right-click the instance you want to delete and click Delete Instance in the pop-up menu.**
   The instance is deleted.

Invoking Methods

In CIM WorkShop, you can set input values for a parameter of a method and invoke the method. Input parameters feed set values, such as a character string, Boolean expression, or integer to a function of the method to enable the function to complete its operations. Invoking the method returns additional data in the form of output parameters.

For information about the dialog boxes you use to set parameter values and invoke methods, see "Value Type Dialog Boxes" on page 52 and "Invoke Methods Dialog Box" on page 57.

▼ How to Invoke a Method

1. **In the CIM WorkShop window.**
2. **Select Action->Instances->Deep Enumeration to list the instances of the selected class and is subclasses. Select Action->Instances->Shallow Enumeration to list the instances of the selected class.**
3. **Click the Methods Tab.**
4. **Right-click the method to invoke and select Invoke Method.**
5. **In the Input Value column of the Invoke Method dialog box, click the cell of the value you want to add.**
   A Cell Value dialog box opens.
6. **Type an input value in the cell for the parameter and click OK.**
7. **Click Invoke Method.**
   All output values and the return value are filled in automatically.
8. **To add another input value, click and type a value in the corresponding cell of the Input Value column.**
9. **When you have finished adding new input values and invoking the method, click Close.**

Reference: CIM WorkShop Window and Dialogs

The following section provides descriptions of the frames, toolbar icons, and fields that comprise the CIM WorkShop window. It also describes CIM WorkShop dialogs.

The CIM WorkShop Window

The CIM WorkShop window is divided into two main frames. In the left frame, you can view the class inheritance tree of the current host. In the right frame, you can view the properties and methods of a selected class.

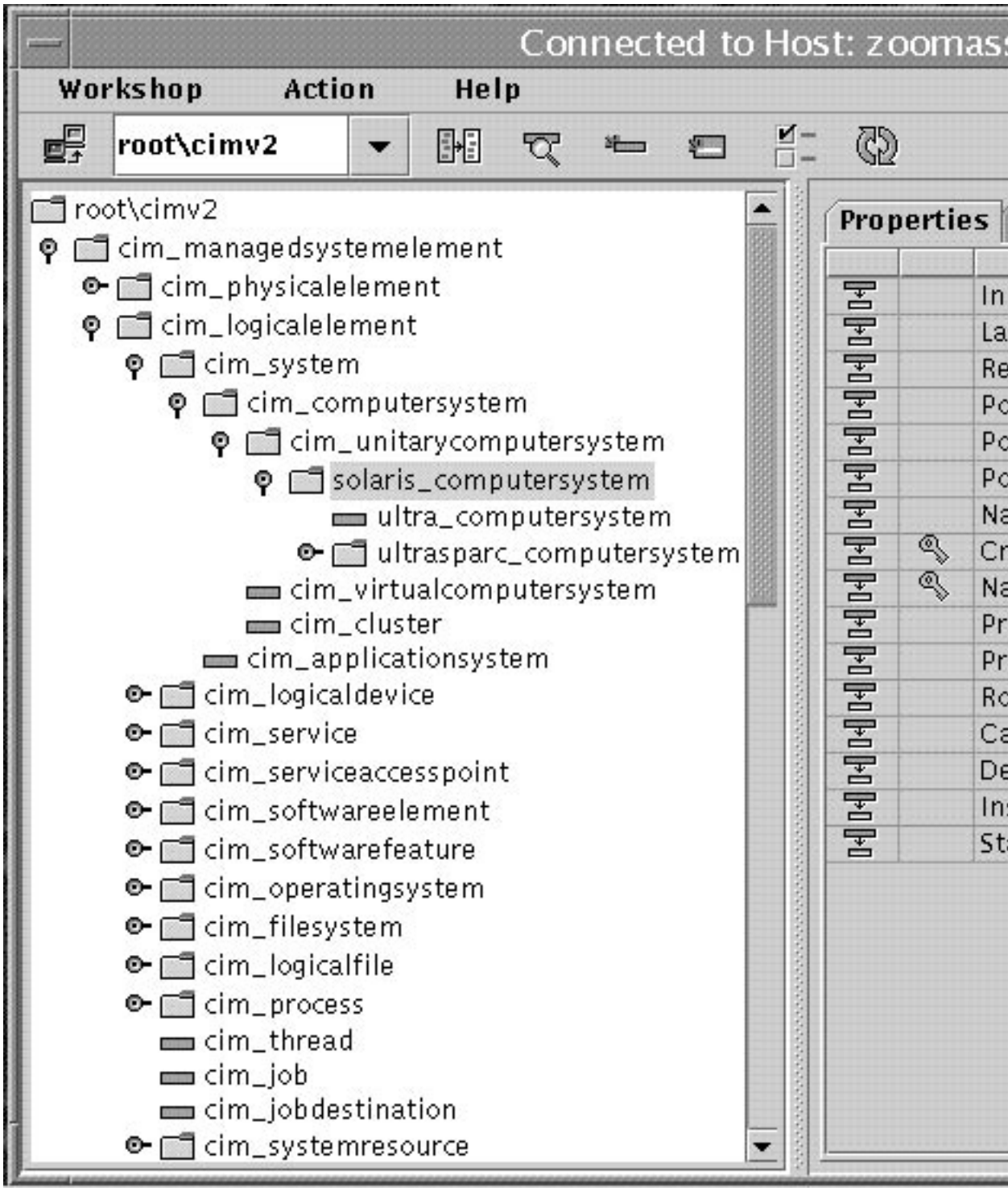
Connected to Host: zoomass
Workshop
Action
Help
root\cimv2
root\cimv2
cim_managedsystemelement
cim_physicalelement
cim_logicalelement
cim_system
cim_computersystem
cim_unitarycomputersystem
solaris_computersystem
ultra_computersystem
ultrasparc_computersystem
cim_virtualcomputersystem
cim_cluster
cim_applicationsystem
cim_logicaldevice
cim_service
cim_serviceaccesspoint
cim_softwareelement
cim_softwarefeature
cim_operatingsystem
cim_filesystem
cim_logicalfile
cim_process
cim_thread
cim_job
cim_jobdestination
cim_systemresource
Properties

FIGURE 2–2 The CIM WorkShop Window

TABLE 2–1 Frames of the CIM WorkShop Window

Frame	Description
Left frame	Displays classes and instances contained in the namespace of the current host. The left frame in the CIM WorkShop shows the contents of the selected namespace. The classes that belong to the namespace are displayed hierarchically. This organization of classes is known as a class inheritance tree. Classes that contain subclasses are represented as a key icon and a folder. Clicking the key or double-clicking the folder causes the list of subclasses to display. Classes that do not contain subclasses are represented by page icons.
Right frame	Provides a Properties tab and a Methods tab from which you can view the values of properties and methods of a class. You can view attributes and values of qualifiers and flavors by right-clicking on a property or method.
Toolbar	Provides icons that enable you to change hosts, change location to a namespace within the default namespace `root\cimv2`, find a class in the class inheritance tree, create a subclass, and show instances and qualifiers of a selected class and refresh selected class.
Title bar	Posts the title of the CIM WorkShop window

CIM WorkShop Toolbar Icons

The icons provided in the CIM WorkShop toolbar enable you to display and change namespaces and search for classes and instances.

FIGURE 2–3 The CIM WorkShop Toolbar

TABLE 2–2 Icons of the CIM WorkShop Toolbar

Icon	Description
Change Hosts	Enables you to connect to a different host or name space and to log in with a different user name and password, and set the transfer protocol.
Change Namespace	Causes the Change Namespace dialog box to display in which you can select another name space to view.

TABLE 2–2 Icons of the CIM WorkShop Toolbar *(Continued)*

Icon	Description
Find Class	Enables you to search for a specific class in the name space.
Add New Class	Causes the New Class dialog box to display in which you can create a new subclass of a selected class.
Show Instances	Causes the Show Instances dialog box to display in which you can view instances of a selected class.
Show Qualifiers	Causes the Qualifiers dialog box to display in which you can view the qualifiers of a selected class.
Refresh Selected Class	Resets the display of the class hierarchy tree. Open class folders are closed and the tree is returned to the state it was in when it was first displayed.

The Properties Tab

The Properties tab shows information about a selected property. An icon resembling a folder with an arrow indicates that the property is inherited from a superclass. An icon resembling a gold key indicates that the property is a Key. Key properties provide unique identifiers for an instance of the domain class. The unique instance is indicated by a key qualifier.

In the Properties tab, the Name, Type, and Value of the property are displayed. You can change the value of a property when you create a new class of the domain class.

The Methods Tab

A method is a function that describes the behavior of a class. Examples of methods are behaviors such as start service, stop service, format disk, and so on. By selecting the Methods tab, you can view all methods of the class. Methods are listed consecutively.

Methods have two parts, a signature and a body. The signature consists of the method name, the parameters names, types, and their order, and the method return type. The method body consists of a sequence of instructions.

Reading from left to right horizontally, the method contains three parts:

- Return data type — The data type of the return value for this method.
- Name of the method
- Parameters — A comma-separated list of parameters enclosed within parentheses. Each parameter has a name and data type. Parameters that are input to the method are preceded with [IN]; parameters that are output from the method are preceded with [OUT]. A parameter can have one or more qualifiers.

In the following example, the method `SetDateTime` takes the input parameter `Time`, which is of type `datetime` and returns a `boolean`.

```
boolean SetDateTime([IN(true)] datetime Time);
```

CIM WorkShop Menus

The following table describes the CIM WorkShop menus and menu items.

TABLE 2–3 CIM WorkShop Menus and Menu Items

Menu	Menu Item	Description
Workshop	Change Host	Causes the Login Dialog box to display, in which you can change the host and namespace.
	Change Namespace	Causes the Change Namespace dialog box to display, in which you can change to a location other than the default namespace in the `root\cimv2` namespace.
	Exit	Enables you to exit CIM Workshop.

TABLE 2–3 CIM WorkShop Menus and Menu Items *(Continued)*

Menu	Menu Item	Description
Action	Add Class	Causes the New Class dialog box to display in which you can create a subclass for a selected class.
	Delete Class	Deletes a selected class.
	Find Class	Enables you to specify a class to find in the class inheritance tree.
	Instances	Causes the Instances dialog box to display for the selected class. In this dialog box, you can view all instances of the class, add new instances, and delete instances.
	Qualifiers	Causes the Qualifiers dialog box to display. In this dialog box, you can view qualifier values, scope, and flavor of a selected class. You can select either deep or shallow enumeration of instances.
	Association Traversal	Causes the Association Traversal dialog box to display, in which you can view and traverse the associations of a class.
	Refresh	Causes the latest changes to be retrieved from the CIM Object Manager and displayed in CIM WorkShop for a selected class or namespace.

Login Dialog Box

The login dialog box is displayed when you first encounter CIM WorkShop. In the login dialog box, you specify the following:

- Host on which the CIM Object Manager is running and which contains the namespace you want to use
- Namespace in which you want to work
- Your user name
- Your password
- Transfer protocol

If you do not specify a user name and password, you log in to CIM WorkShop as a guest. Guest privileges are read-only.

By default, CIM WorkShop uses the RMI protocol to connect to the CIM Object Manager on the local host, in the default namespace, `root\cimv2`. You can select HTTP if you want to communicate to a CIM Object Manager using the standard XML/HTTP protocol from the Desktop Management Task Force. When a connection is established, all classes contained in the default namespace are displayed in the left side of the CIM WorkShop window.

New Class Dialog Box

In the New Class dialog box, you can create a new class.

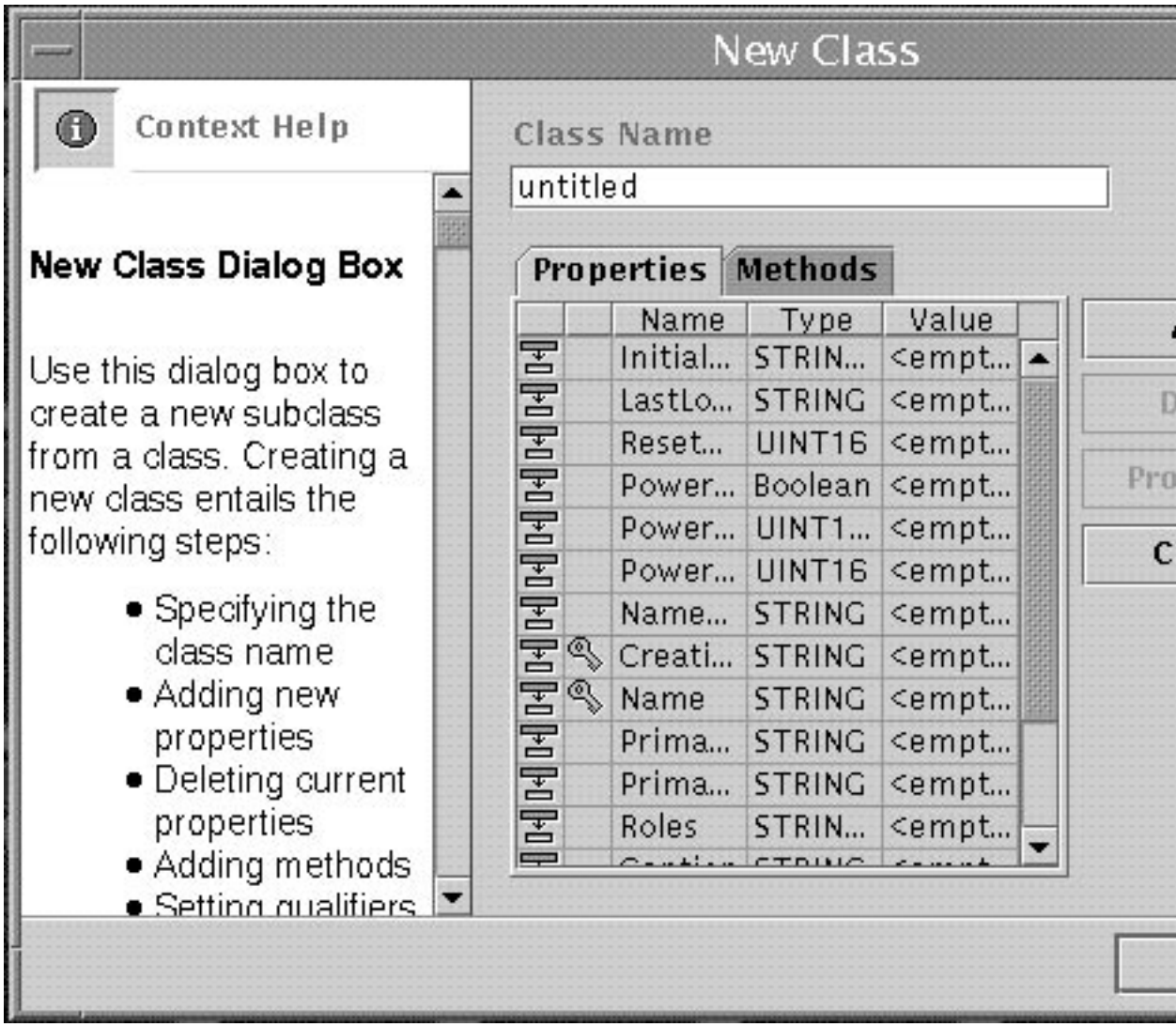

FIGURE 2–4 The New Class Dialog Box

Add Property Dialog Box

In the Add Property dialog box, you can add new properties to a class as you create it. In the Name field, you specify the name of the property. In the type field, select a type and click OK.

Qualifiers Dialog Box

In the Qualifiers dialog box, you can view qualifiers for a selected class, property, or method. When you create a new class, you can add qualifiers to the class or modify qualifiers of the class, its properties, or its methods in the Qualifiers dialog box. The title bar of the Qualifiers dialog box indicates the name of the class for which you view qualifiers or the class for which you add or modify qualifiers.

The following table describes the fields of the Qualifiers dialog box.

TABLE 2–4 Qualifiers Dialog Box Fields

Name of Field	Description	Example
Name	Shows the name of the qualifier	Provider
Type	Shows the type of value that the qualifier provides	`string`
Value	Shows the value of the qualifier	`Solaris`

The following table describes the buttons of the Qualifiers dialog box.

TABLE 2–5 Qualifiers Dialog Box Buttons

Name of Button	Description
Scope	Causes the Scope dialog box to display, in which you can view the scope of a selected qualifier
Flavors	Causes the Flavors dialog box to display, in which you can view the flavor of a selected qualifier
Add Qualifier	Causes the Add Qualifier dialog box to display, in which you can select a qualifier to add for a new subclass, property, or method
Delete Qualifier	Causes a selected qualifier to be deleted from the Qualifiers dialog box

Scope Dialog Box

In the Scope dialog box, you can view the scope of a qualifier that modifies an existing class, property, or method.

Flavors Dialog Box

In the Flavors dialog box, you can view the flavor of a qualifier.

Value Type Dialog Boxes

When you create new properties for a class or set input parameters for a method, you can use any of the dialog boxes that CIM WorkShop provides for specifying values of a particular type. These dialog boxes are configured to accept only a value of the appropriate type. The dialog boxes include the following:

- Real Integer dialog box
- Signed Integer dialog box
- Unsigned Integer dialog box
- String dialog box
- Array dialog box
- Boolean dialog box
- Date/Time dialog box

Real Integer Dialog Box

The Values field of this dialog box accepts only a real integer. A real integer can be a negative or positive number including a decimal point. When you create a property that is of the type Real Integer, type a real integer in the Values field of this dialog box.

Signed Integer Dialog Box

The Values field of this dialog box accepts only a signed integer of a specified size. A signed integer can be a negative or positive whole number. CIM properties that have values which are signed integers can be 8 bits, 16 bits, 32 bits, or 64 bits in size. Depending on the size of the signed integer that makes up the value of the property, type the following in the Values field of this dialog box:

- For a property that is an 8–bit signed integer, type a positive or negative numeric value equivalent to 8 bits.
- For a property that is a 16–bit signed integer, type a positive or negative numeric value equivalent to 16 bits.
- For a property that is a 32–bit signed integer, type a positive or negative numeric value equivalent to 32 bits.
- For a property that is a 64–bit signed integer, type a positive or negative numeric value equivalent to 64 bits.

Unsigned Integer Dialog Box

The Values field of this dialog box accepts only an unsigned integer of a specified size. An unsigned integer can be only a positive whole number. CIM properties that have values which are unsigned integers can be 8 bits, 16 bits, 32 bits, or 64 bits in size. Depending on the size of the unsigned integer that makes up the value of the property, type the following in the Values field of this dialog box:

- For a property that is an 8–bit unsigned integer, type a positive numeric value equivalent to 8 bits.
- For a property that is a 16–bit unsigned integer, type a positive numeric value equivalent to 16 bits.
- For a property that is a 32–bit unsigned integer, type a positive numeric value equivalent to 32 bits.
- For a property that is a 64–bit unsigned integer, type a positive numeric value equivalent to 64 bits.

String Dialog Box

The Values field of this dialog box accepts alphanumeric characters. When you specify the value of a property that is a character string, you must enter a character string, such as Processor_Type, in the Values field of this dialog box. Character strings may not contain integers.

Array Dialog Boxes

In the Array dialog boxes, you can specify an array as a value for a property. The following array dialog boxes are available to return arrays:

- 8–Bit Unsigned Integer Array Dialog Box – returns a collection of positive integers equivalent to 8 bits in size
- 16–Bit Unsigned Integer Array Dialog Box – returns a collection of positive integers equivalent to 16 bits in size
- 32–Bit Unsigned Integer Array Dialog Box – returns a collection of positive integers equivalent to 32 bits in size
- 64–Bit Unsigned Integer Array Dialog Box – returns a collection of positive integers equivalent to 64 bits in size
- 8–Bit Signed Integer Array Dialog Box – returns a collection of positive or negative integers equivalent to 8 bits in size
- 16–Bit Signed Integer Array Dialog Box – returns a collection of positive or negative integers equivalent to 16 bits in size
- 32–Bit Signed Integer Array Dialog Box – returns a collection of positive or negative integers equivalent to 32 bits in size
- 64–Bit Signed Integer Array Dialog Box – returns a collection of positive or negative integers equivalent to 64 bits in size
- String Array Dialog Box – returns a collection of alphanumeric character strings
- Boolean Array Dialog Box – returns a collection of Boolean expressions, True or False
- 32–Bit Real Array Dialog Box – returns a collection of positive or negative real numbers, with or without a decimal point, equivalent to 32 bits in size

- 64–Bit Real Array Dialog Box – returns a collection of positive or negative real numbers, with or without a decimal point, equivalent to 64 bits in size
- 16–Bit Character Array Dialog Box – returns a collection of alphabetic and numeric character strings equivalent to 16 bits in size
- Date/Time Array Dialog Box – returns a collection of dates in the format *mm-dd-yy* and times in the format *hh:mm:ss*

Boolean Dialog Box

In the Boolean dialog box, you can specify True or False as the value of a selected property.

Date/Time Dialog Box

The DateTime dialog box enables you to set the date and time value as specified in the CIM Specification for a new property or method. The DateTime dialog box is displayed from:

- New Class dialog box when you select the value of a new property that uses the DateTime value
- Invoke Methods dialog box when you select the Input Value of a method's parameter

A Set Value dialog box is displayed. Click Add in the Set Value dialog box to display the Date/Time dialog box.

In the fields of the DateTime dialog box, type the date and time value in the following manner:

- Year *yyyy* is a 4-digit year
- Month *mm* is the month
- Day *dd* is the day
- Hour *hh* is the hour, on a 24-hour clock
- Minutes *mm* is the minute
- Seconds *ss* is the second
- Microseconds*mmmmmm* is the number of microseconds
- Universal Coordinate Time *s* is a positive (+) or negative (-) sign that indicates the Universal Coordinated Time, or a (:)

Note – If a *(:)* is provided, the value is interpreted as a time interval, and *yyyymm* are interpreted as days.

- *utc* is the offset from UTC in minutes

For example, Monday June 7, 1999 at 1:30:15 PM EST would be represented as: *19990607133015.000000-300*

Instance Window

The Instance window lists all instances for a selected class. You can also view the properties, methods, and qualifiers associated with each instance.

You display the Instance window by:

- Right-clicking a class in the CIM WorkShop window and clicking Instances in the pop-up menu
- Click Action->Instances->Deep Enumeration or Shallow Enumeration.

Frames of the Instances Window

If the selected class has instances, the instances are listed in the left frame of the Instances window. Each instance is displayed with its Name, CreationClassName, and TargetOperatingSystem. If the selected class does not have instances, a message is displayed.

Like the CIM WorkShop window, the right frame of the Instances window contains two tabs: Properties tab and Methods tab. All properties of the selected instance are displayed in the table of the Properties tab. The Inherited Properties icon—which appears in the left column of the Properties table as a purple rectangle with an arrow pointing to a white rectangle—indicates that the property is inherited from the class used to create the instance. The Key Qualifiers icon—which appears as a gold key—indicates that a property has an inherited Key qualifier.

Instances Window Toolbar Icons

The Instances window contains the following icons in the toolbar:

TABLE 2–6 Instance Window Toolbar Icons

Icon Name	Description
Add New Instance	Causes the Add Instance dialog box to display in which you can create a new instance to add to the class inheritance tree
Delete Selected Instance	Enables you to delete a selected instance

TABLE 2–6 Instance Window Toolbar Icons *(Continued)*

Save Current Instance Property Values	Updates the property values of the current instance.
Refresh Instance List	Updates the list of instances in the left side of the Instances window with the newest instances and the latest changes to instances

Menus of the Instances Window

The instances window contains the following menus and menu items:

TABLE 2–7 Menus of the Instances Window

Menu Name	Menu Item	Description
Instance Editor	Exit	Causes the Instances window to close
Action	Add Instance	Causes the Add Instances dialog box to display, in which you can create new instances to add to the class inheritance tree
	Delete Instance	Enables you to delete a selected instance
	Association Traversal	Displays the Reference Traversal dialog box, in which you can view and traverse all the associations of a class.
	Refresh	Causes the latest changes to be retrieved from the CIM Object Manager and displayed in CIM WorkShop for a selected class or namespace.
Help		Displays CIM WorkShop Copyright information.

Add Instance Dialog Box

In the Add Instance dialog box, if a property can be changed, you can click in the value field to open a Reference Edit dialog box. In the Reference Edit dialog box, click the value field to open a value dialog box. You cannot change the values of inherited properties.

Invoke Methods Dialog Box

The Invoke Methods dialog box is displayed from the Methods tab of an instance of a class that contains methods. Right-click the method and select Invoke Method from the menu. In the Invoke Methods dialog box, you can set the input values for the variables, or parameters, of a method and invoke the method. An example of a parameter value is a signed integer or a Date/Time value.

The Invoke Methods dialog box supports three types of parameters:

- Input parameters specify data values that are passed to a function and executed
- Output parameters specify data values that are returned by a function to the screen of an application or a printer
- Input/Output parameters take data to complete functions and return values

The Parameter Type column of the Invoke Methods dialog box indicates whether the parameter of the method is input, output, or input/output. The value of an input parameter is displayed in the Input Value column. The value of an output parameter is displayed in the Output Value column.

For information about how to invoke a method, see “Invoking Methods” on page 42.

CHAPTER 3

Application Programming Interfaces

The Sun WBEM SDK applications request information or services from the Common Information Model (CIM) Object Manager through the application programming interfaces (APIs). This chapter describes the following topics.

- About the APIs
- The API Packages

For detailed information on the CIM, Client, and Provider APIs, see the Javadoc reference pages.

About the APIs

The APIs represent and manipulate CIM objects. These APIs represent CIM objects as Java classes. An object is a computer representation or model of a managed resource, such as a printer, disk drive, or CPU. Because the CIM Object Manager enforces the Common Information Model (CIM) 2.2 Specification, the objects you model using the APIs conform to standard CIM objects.

Programmers can use these interfaces to describe managed objects and retrieve information about managed objects in a particular system environment. The advantage of modeling managed resources using CIM is that those objects can be shared across any system that is CIM compliant.

The API Packages

The API can be grouped into three categories:

- CIM API – Common classes and methods that applications use to represent all basic CIM elements. The CIM APIs create objects on the local system.
- Client API – Methods that applications use to transfer data to and from the CIM Object Manager. The Client APIs transfer objects that have been created on the local system to the CIM Object Manager.
- Provider API – Interfaces that the CIM Object Manager and object provider use to communicate with each other

CIM API Package (`com.sun.wbem.cim`)

The following table describes the interfaces in the CIM API package.

TABLE 3–1 CIM Classes

Class	Description
`CIMClass`	A CIM class, an object representing a collection of CIM instances, all of which support a common type (for example, a set of properties and methods). This interface creates a template that fills in the required CIM values for the group of objects you are creating.
`CIMDataType`	The CIM data types (as defined by the CIM specification).
`CIMDateTime`	The CIM date and time representation.
`CIMElement`	A CIM element. This is the base class for managed system elements.
`CIMFlavor`	A CIM qualifier flavor, a characteristic of a qualifier that describes rules that specify whether a qualifier can be propagated to derived classes and instances, and whether or not a derived class or instance can override the qualifier's original value.
`CIMInstance`	A unit of CIM data. Use this interface to describe a managed object that belongs to a particular class. Instances contain actual data.

TABLE 3–1 CIM Classes *(Continued)*

Class	Description
`CIMMethod`	A declaration containing the method name, return type, and parameters.
`CIMNameSpace`	A CIM namespace, a directory-like structure, that can contain other namespaces, classes, instances, qualifier types, and qualifiers.
`CIMObjectPath`	A path name of a CIM object. The object names have two parts: namespace and a model path. The model path uniquely identifies an object in the namespace.
`CIMParameter`	A CIM parameter, a value passed to a CIM method from a calling method.
`CIMProperty`	A value that characterizes an instance of a CIM class. Properties can be thought of as a pair of functions, one to set the property value, and one to return the property value. A property has a name and one domain, the class that owns the property.
`CIMQualifier`	A modifier that describes a class, instance, method, or property. Use this class to modify an attribute of a managed object, for example, add read-only access to a disk. There are two categories of qualifiers: those defined by the Common Information Model (CIM) and those defined by developers.
`CIMQualifierType`	A CIM qualifier type, a template for creating CIM qualifiers.
`CIMScope`	A CIM scope, a qualifier attribute that indicates the CIM objects with which the qualifier can be used. For example, the qualifier ABSTRACT has Scope(Class Association Indication), meaning that it can only be used with classes, associations, and indications.
`CIMValue`	A CIM value, a value that can be assigned to properties, references, and qualifiers. CIM values have a data type (`CIMDataType`) and the actual value(s).
`UnsignedInt8`	An unsigned 8-bit integer.
`UnsignedInt16`	An unsigned 16-bit integer.
`UnsignedInt32`	An unsigned 32-bit integer.
`UnsignedInt64`	An unsigned 64-bit integer.

Exception Classes

The Exception classes represent the error conditions that can occur in Sun WBEM SDK classes. The `CIMException` class is the base class for CIM exceptions. All other CIM exception classes extend from the `CIMException` class.

The following table describes the CIM exception classes.

TABLE 3–2 Exception Classes

Class	Description
`CIMClassException`	A semantic exception that occurs in a CIM class. The MOF Compiler (`mofc`) uses this class to handle semantic errors found during compilation.
`CIMException`	Exceptional CIM conditions. This is the base class for CIM exceptions.
`CIMInstanceException`	Semantic exceptions that occur in a CIM instance.
`CIMMethodException`	Semantic exceptions that occur in a CIM method.
`CIMNameSpaceException`	Semantic exceptions that occur in a CIM namespace.
`CIMPropertyException`	Semantic exceptions that occur in a CIM property.
`CIMProviderException`	Exceptional conditions that can occur in the CIM Object Manager's providers.
`CIMQualifierTypeException`	Exceptional conditions that can occur in a CIM qualifier type.
`CIMRepositoryException`	Exceptional conditions that can occur in the CIM repository.
`CIMSemanticException`	Semantic exceptions that can occur in a CIM element. These exceptions are generally thrown when the CIM Object Manager tries to add, modify, or delete a CIM element and encounters situations that are illegal according to the CIM Specification.
`CIMTransportException`	Exceptional conditions that occur in the CIM transport interfaces (RMI and XML).

Client API Package (`com.sun.wbem.client`)

The Client API package contains classes and methods that transfer data between applications and the CIM Object Manager. Applications use the `CIMClient` class to connect to the CIM Object Manager, and they use the methods in the `CIMClient` class that are listed in the following table to transfer data to and from the CIM Object Manager.

TABLE 3–3 Client Methods

Method	Description
`associators`	Gets the CIM classes or instances that are associated with the specified CIM class or instance.
`associatorNames`	Gets the names of the CIM classes or instances that are associated with the specified CIM class or instance.
`close`	Close the client connection to the CIM Object Manager. This interface frees resources used for the client session.
`createClass`	Adds the CIM class to the specified namespace.
`createInstance`	Creates the specified instance if it does not exist. If the CIM instance already exists, throws `CIMInstanceException` with ID `CIM_ERR_ALREADY_EXISTS`.
`createNameSpace`	Creates a CIM namespace, a directory containing classes and instances. When a client application connects to the CIM Object Manager, it specifies a namespace. All subsequent operations occur within that namespace on the CIM Object Manager host
`createQualifierType`	Adds the specified CIM qualifier type to the specified namespace.
`deleteNameSpace`	Deletes the specified namespace on the specified host.
`deleteClass`	Deletes the specified class.
`deleteInstance`	Deletes the specified instance.

TABLE 3–3 Client Methods *(Continued)*

Method	Description
`enumClass(CIMObjectPath path, boolean deep)`	Enumerates the class specified by *Path*. This method returns the *name* of each class, not the class contents, as an enumeration of `CIMObjectPath` objects. If set to *deep*, the enumeration contains the names of all classes derived from the enumerated class. If set to *shallow*, the enumeration contains only the names of the first-level childeren of the enumerated class.
`enumClass(CIMObjectPath path, boolean deep, boolean localonly)`	Enumerates the class specified by *Path*. This method returns the entire class contents, not just the class name, as an enumeration of `CIMClass` objects. If set to *deep*, returns the classes derived from the enumerated class. If set to *shallow*, returns only the first-level children of the enumerated class. If *localOnly* is true, only the non-inherited properties and methods are returned. Otherwise, all proeprties and methods are returned.
`enumInstances(CIMObjectPath path, boolean deep)`	Enumerates the instances in the class specified by *path*. Returns the *names* of the instances for the class specified by `path` as an enumeration of `CIMObjectPath` objects. If *deep* is true, returns the names of all instances of the specified class and all classes derived from the class. Otherwise, returns only the names of instances belonging to the specified class.
`enumInstances(CIMObjectPath path, boolean deep, boolean localOnly)`	Enumerates the instances in the class specified by *path*. Returns the instances (the entire instance not just the name of the instance) for the specified class as an enumeration of `CIMInstance` objects.
`enumNameSpace`	Gets a list of namespaces.
`enumQualifierTypes`	Gets a set of qualifier types for the specified class or classes.
`execQuery(CIMObjectPath relNS, java.lang.String query, int ql)`	Returns an enumeration of instances containing properties that match the property value specified in the query, using the query language specified by ql. Currently, only WBEM Query Language (WQL) is supported. The query language maps the CIM object model to SQL tables.

TABLE 3–3 Client Methods *(Continued)*

Method	Description
`getClass`	Gets the CIM class for the specified CIM object path.
`getInstance`	Gets the CIM instance for the specified CIM object path.
`getProperty`	Returns the value of the specified property.
`getQualifierType`	Gets the qualifier type for the specified CIM object path.
`invokeMethod`	Executes the specified method on the specified object. A method is a declaration containing the method name, return type, and parameters in the method.
`references`	Gets the associations that refer to the specified CIM class or instance.
`referenceNames`	Gets the names of the associations that refer to the specified CIM class or instance.
`setClass`	Updates the specified CIM class if it exists. Returns an error if the CIM class does not exist.
`setInstance`	Updates the specified CIM instance if it exists. Returns an error if the CIM instance does not exist.
`setProperty`	Sets the specified property to the specified value.

The following table describes the interfaces provided in the `com.sun.wbem.client package`.

TABLE 3–4 Interfaces in the `com.sun.wbem.client` Package

Interface	Description
`CIMOMHandle`	Provides to clients a reference to the CIM Object Manager. This interface contains methods that clients can use to transfer data to and from the CIM Object Manager.
`ProviderCIMOMHandle`	Provides to providers a reference to the CIM Object Manager. This interface contains methods that providers can use to transfer data to and from the CIM Object Manager.

The following table describes the methods in the `ProviderCIMOMHandle` Interface.

TABLE 3–5 Methods in the `ProviderCIMOMHandle` Interface

Method	Description
`decryptData`	Decrypts the specified string value using the authentication sessionkey, if the value is encrypted.
`getCurrentAuditId`	Returns a usually unique identifier for the session to be used in auditing records to identify the remote client connection
`getCurrentRole`	Returns the current role assumed by the current authenticated user.
`getCurrentUser`	Returns the current user on whose behalf the provider has been invoked.
`getInternalProvider`	Returns a reference to an internal instance provider, which can be used to store static instance information for the provider.

Provider API Package

The Provider API packages (`com.sun.wbem.provider`) and (`com.sun.wbem.provider20`) contain the provider interfaces that the CIM Object Manager and object providers use to communicate with each other. Providers can use these interfaces to provide the CIM Object Manager dynamic data.

When an application requests dynamic data from the CIM Object Manager, the CIM Object Manager uses these interfaces to pass the request to the provider. Providers are classes that perform the following functions in response to a request from the CIM Object Manager:

- Map information from a managed device to CIM Java classes
  - Get information from a device
  - Pass the information to the CIM Object Manager in the form of CIM Java classes
- Map the information from CIM Java classes to managed device format
  - Get the required information from the CIM Java class
  - Pass the information to the device in native device format

The following table describes the interfaces in the Provider package.

TABLE 3–6 `com.sun.wbem.provider` Interfaces

Interface	Description
`CIMProvider`	Base interface implemented by all providers.

TABLE 3–6 `com.sun.wbem.provider` Interfaces *(Continued)*

Interface	Description
`InstanceProvider`	Interface implemented by instance providers. Instance providers serve dynamic instances of classes. This is a deprecated interface. Use the `InstanceProvider` interface in `com.sun.wbem.provider20` instead.
`MethodProvider`	Interface implemented by method providers, which provide implementation for all methods of CIM classes.
`PropertyProvider`	Interface implemented by property providers, which are used to retrieve and update dynamic properties. Dynamic data is not stored in the CIM Object Manager Repository.

The following table describes the interfaces in the `com.sun.wbem.provider20` package.

TABLE 3–7 `com.sun.wbem.provider20` Interfaces

Interface	Description
`AssociatorProvider`	Interface implemented by providers of dynamic associations.
`Authorizable`	Maker interface implemented by providers that handle authorization checking instead of relying on the CIM Object Manager to check user authorizations.
`InstanceProvider`	Interface implemented by instance providers. Instance providers serve dynamic instances of classes.

CHAPTER 4

Writing Client Applications

This chapter explains how to use the Client Application Programming Interfaces (APIs) to write client applications.

- Overview
- Opening and Closing a Client Connection
- Working with Instances
- Enumerating Objects
- Querying
- Associations
- Calling Methods
- Retrieving Class Definitions
- Handling Exceptions
- Advanced Programming Topics
- Sample Programs

For detailed information on the CIM and Client APIs, see the Javadoc reference pages.

Overview

A Web-Based Enterprise Management (WBEM) application is a standard Java program that uses Sun WBEM SDK APIs to manipulate CIM objects. A client application typically uses the CIM API to construct an object (for example, a namespace, class, or instance) and then initialize that object. The application then uses the Client APIs to pass the object to the CIM Object Manager and request a WBEM operation, such as creating a CIM namespace, class, or instance.

Sequence of a Client Application

Sun WBEM SDK applications typically follow this sequence:

1. Connect to the CIM Object Manager - (`CIMClient`).

   A client application contacts a CIM Object Manager to establish a connection each time it needs to perform a WBEM operation, such as creating a CIM Class or updating a CIM instance.

2. Use one or more APIs to perform some programming tasks.

   Once a program connects to the CIM Object Manager, it uses the APIs to request operations.

3. Close the client connection to the CIM Object Manager - `(close)`.

   Applications should close the current client session when finished. Use the `CIMClient` interface to close the current client session and free any resources used by the client session.

Example — Typical Sun WBEM SDK Application

Example 4–1 is a simple application that connects to the CIM Object Manager, using all default values. The program gets a class and then enumerates and prints the instances in that class.

EXAMPLE 4–1 Typical Sun WBEM SDK Application

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * Returns all instances of the specified class.
 * This method takes two arguments: hostname (args[0])
 * and name of class to list (args[1]).
 */
public class WBEMsample {
    public static void main(String args[]) throws CIMException {
        CIMClient cc = null;
            try {
                /* args[0] contains the namespace. We create
                a CIM namespace (cns) pointing to the default
                  root\cimv2 namespace on the specified host. *    /
                          CIMNameSpace cns = new CIMNameSpace(args[0]);
```

EXAMPLE 4–1 Typical Sun WBEM SDK Application *(Continued)*

```
                        /* Connect to the CIM Object manager and pass it
                          the namespace object containing the namespace. */
                        cc = new CIMClient(cns, "root", "root_password");
                        /* Create a CIMObjectPath from the class name. */
                        CIMObjectPath cop = new CIMObjectPath(args[1]);
                        /* Get the class, including qualifiers,
                  class origin, and properties. */
                  cc.getClass(cop, true, true, true, null);
                // Return all instances names belonging to the class.
                Enumeration e = cc.enumerateInstanceNames(cop);
                while(e.hasMoreElements()) {
                    CIMObjectPath op = (CIMObjectPath)e.nextElement();
                    System.out.println(op);
                  } // end while
                  } catch (Exception e) {
                System.out.println("Exception: "+e);
            }
            if(cc != null) {
               cc.close();
            }
    } // end main
} // end WBEMsample
```

Typical Programming Tasks

Once a client application connects to the CIM Object Manager, it uses the API to request operations. The program's feature set determines which operations it needs to request. The typical tasks that most programs perform are:

- Working with instances – creating, deleting, and updating
- Enumerating objects
- Calling methods
- Retrieving class definitions
- Handling errors

In addition, applications may occasionally perform the following tasks:

- Creating namespaces
- Deleting namespaces
- Creating a class
- Deleting a class
- Working with qualifiers

Opening and Closing a Client Connection

The first task an application performs is to open a client session to a CIM Object Manager. WBEM Client applications request object management services from a CIM Object Manager. The client and CIM Object Manager can run on the same hosts or on different hosts. Multiple clients can establish connections to the same CIM Object Manager.

This section describes some basic concepts about namespaces and explains how to use:

- The `CIMClient` class to connect to the CIM Object Manager
- The `close` method to close the client connection

Using Namespaces

Before writing an application, you need to understand the CIM concept of a namespace. A namespace is a directory-like structure that can contain other namespaces, classes, instances, and qualifier types. The names of objects within a namespace must be unique. All operations are performed within a namespace. The installation of Solaris WBEM Services creates two namespaces:

- `root\cimv2` – Contains the default CIM classes that represent objects on the system on which Solaris WBEM Services is installed. This is the default namespace.
- `root\security` – Contains the security classes

When an application connects to the CIM Object Manager, it must either connect to the default namespace (`root\cimv2`) or specify another namespace, for example, `root\security` or a namespace you created.

Once connected to the CIM Object Manager in a particular namespace, all subsequent operations occur within that namespace. When you connect to a namespace, you can access the classes and instances in that namespace (if they exist) and in any namespaces contained in that namespace. For example, if you create a namespace called `child` in the `root\cimv2` namespace, you could connect to `root\cimv2` and access the classes and instances in the `root\cimv2` and `root\cimv2\child` namespaces.

An application can connect to a namespace within a namespace. This is similar to changing to a subdirectory within a directory. Once the application connects to the new namespace, all subsequent operations occur within that namespace. If you open a new connection to `root\cimv2\child`, you can access any classes and instances in that namespace but cannot access the classes and instances in the parent namespace, `root\cimv2`.

Connecting to the CIM Object Manager

A client application contacts a CIM Object Manager to establish a connection each time it needs to perform a WBEM operation, such as creating a CIM class or updating a CIM instance. The application uses the `CIMClient` class to create an instance of the client on the CIM Object Manager. The `CIMClient` class takes three optional arguments:

- *namespace*

  A `CIMNameSpace` object that contains the names of the host name and namespace to use for this client connection. The default is `root\cimv2` on the local host.

- *user name*

  The name of a valid Solaris user account. The CIM Object Manager checks the access privileges for this user to determine what type of access to CIM objects is allowed. The default user account is `guest`. By default, the `guest` account allows users read access to all CIM objects in all namespaces.

- *password*

  The password for this user account. The password must be a valid password for the user's Solaris account. The default password is `guest`.

Once connected to the CIM Object Manager, all subsequent `CIMClient` operations occur within the specified namespace.

Examples — Connecting to the CIM Object Manager

The following examples show two ways of using the `CIMClient` class to connect to the CIM Object Manager.

In Example 4–2, the application takes all the default values. That is, it connects to the CIM Object Manager running on the local host (the same host the client application is running on), in the default namespace (`root\cimv2`), using the default user account and password, `guest`.

EXAMPLE 4–2 Connecting to the Default Namespace

```
/* Connect to root\cimv2 namespace on the local
host as user guest with password guest. */

cc = new CIMClient();
```

In Example 4–3, the application connects to the CIM Object Manager running on the local host, in the default namespace (`root\cimv2`). The application creates a `UserPrincipal` object for the root account, which has read and write access to all CIM objects in the default namespaces.

EXAMPLE 4–3 Connecting to the Root Account

```
{
    ...

    host as root. Create a namespace object initialized with two null strings
    that specify the default host (the local host) and the default
    namespace (root\cimv2).*/

    CIMNameSpace cns = new CIMNameSpace("", "");

    UserPrincipal up = new UserPrincipal("root");
       PasswordCredential pc = new PasswordCredential("root_password");
    /* Connect to the namespace as root with the
    root password. */

    CIMClient cc = new CIMClient(cns, up, pc");
    ...
}
```

In Example 4–4, the application connects to namespace `A` on host `happy`. The application first creates an instance of a namespace to contain the string name of the namespace (`A`). Next the application uses the `CIMClient` class to connect to the CIM Object Manager, passing it the namespace object, user name, and host name.

EXAMPLE 4–4 Connecting to a Non-Default Namespace

```
{
    ...
    /* Create a namespace object initialized with A
    (name of namespace) on host happy.*/
    CIMNameSpace cns = new CIMNameSpace("happy", "A");
    UserPrincipal up = new UserPrincipal("Mary");
    PasswordCredential pc = new PasswordCredential("marys_password");

    // Connect to the namespace as user Mary.
    cc = new CIMClient(cns, "Mary", "marys_password");
    ...

}
```

EXAMPLE 4–5 Authenticating as an RBAC Role Identity

Authenticating a user's role identity requires using the `SolarisUserPrincipal` and `SolarisPasswordCredential` classes. The following examples authenticates as `Mary` and assumes the role `Admin`.

```
{
...
CIMNameSpace cns = new CIMNameSpace("happy", "A");
SolarisUserPrincipal sup = new SolarisUserPrincipal("Mary", "Admin");
SolarisPasswordCredential spc = new
        SolarisPasswordCredential("marys_password", "admins_password");
CIMClient cc = new CIMClient(cns, sup, spc);
```

EXAMPLE 4–5 Authenticating as an RBAC Role Identity *(Continued)*

Closing a Client Connection

Applications should close the current client session when finished. Use the `close` method to close the current client session and free any resources used by the client session. The following sample code closes the client connection. The instance variable *cc* represents this client connection.

```
cc.close();
```

Working with Instances

This section describes how to create a CIM instance, delete a CIM instance, and update an instance (get and set the property values of one or more instances).

Creating an Instance

Use the `newInstance` method to create an instance of an existing class. If the existing class has a key property, an application must set it to a value that is guaranteed to be unique. As an option, an instance can define additional qualifiers that are not defined for the class. These qualifiers can be defined for the instance or for a particular property of the instance and do not need to appear in the class declaration.

Applications can use the `getQualifiers` method to get the set of qualifiers defined for a class.

Example — Creating an Instance

The code segment in Example 4–6 uses the `newInstance` method to create a Java class representing a CIM instance (for example, a Solaris package) from the `Solaris_Package` class.

EXAMPLE 4–6 Creating an Instance (`newInstance()`)

```
...
{
/*Connect to the CIM Object Manager in the root\cimv2
namespace on the local host. Specify the username and password of an
account that has write permission to the objects in the
root\cimv2namespace. */
```

EXAMPLE 4–6 Creating an Instance (`newInstance()`) *(Continued)*

```
CIMClient cc = new CIMClient(cns, "root", "root_password");

// Get the Solaris_Package class
cimclass = cc.getClass(new CIMObjectPath("Solaris_Package"), true, true, true, null);

/* Create a new instance of the Solaris_Package
 class populated with the default values for properties. If the provider
 for the class does not specify default values, the values of the
 properties will be null and must be explicitly set. */

 ci = cimclass.newInstance();
}
...
```

Deleting an Instance

Use the `deleteInstance` method to delete an instance.

Example — Deleting an Instance

The example in Example 4–7 connects the client application to the CIM Object Manager and uses the following interfaces to delete all instances of a class:

- `CIMObjectPath` to construct an object containing the CIM object path of the object to be deleted
- `enumInstance` to get the instances and all instances of its subclasses
- `deleteInstance` to delete each instance

EXAMPLE 4–7 Deleting Instances (`deleteInstance`)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * This example program takes four required command-line arguments and
 * deletes all instances of the specified class and its subclasses. The
 * running this program must specify the username and password of an
 * account that has write permission to the specified namespace.
```

EXAMPLE 4–7 Deleting Instances (`deleteInstance`) *(Continued)*

```
 * /
public class DeleteInstances {
    public static void main(String args[]) throws CIMException {

        // Initialize an instance of the CIM Client class
        CIMClient cc = null;

        // Requires 4 command-line arguments. If not all entered, prints command string.

        if(args.length != 4) {
          System.out.println("Usage: DeleteClass host className username password");
          System.exit(1);
         }
         try {

                 /**
                  * Creates a name space object (cns), which stores the host name
                  * (args[0]) from the command line.
                  */
                 CIMNameSpace cns = new CIMNameSpace(args[0]);

                 /**
                  * Connects to the CIM Object Manager, and passes it the
                  * namespace object (cns) and the username (args[2]) and
                  * password (args[3]) from the command line.
                  */

                 cc = new CIMClient(cns, args[2], args[3]);

                 /**
                  * Construct an object containing the CIM object path
                  * of the class to delete (args[1]) from the command line.
                  */

                 CIMObjectPath cop = new CIMObjectPath(args[1]);

                 /**
                  * Get an enumeration of the instance object paths of the
                  * class and all subclasses of the class. An instance object
                  * path is a reference used by the CIM Object Manager to locate
          * the instance.
                 */

                 Enumeration e = cc.enumerateInstanceNames(cop);

                 /**
                  * Iterate through the instance object paths in the enumeration.
                  * Construct an object to store the object path of each
                  * enumerated instance, print the instance, and then
                  * delete it.
                  */

                 while(e.hasMoreElements()) {
```

EXAMPLE 4–7 Deleting Instances (`deleteInstance`) *(Continued)*

```
                    CIMObjectPath op = (CIMObjectPath)e.nextElement();
                      System.out.println(op);
                      cc.deleteInstance(op);
                  }
            } catch (Exception e) {
                System.out.println("Exception: "+e);
            }
            if(cc != null) {
               cc.close();
            }
   }
}
```

Getting and Setting Instances

An application frequently uses the `getInstance` method to retrieve CIM instances from the CIM Object Manager. When an instance of a class is created, it inherits the properties of the class it is derived from and all parent classes in its class hierarchy. The `getInstance` method takes the Boolean argument *localOnly*. If *localOnly* is true, `getInstance` returns only the non-inherited properties in the specified instance. The non-inherited properties are those defined in the instance itself. If *localOnly* is false, all properties in the class are returned – those defined in the instance and all properties inherited from all parent classes in its class hierarchy.

To create a new instance, use the `CIMInstance` method in the `CIMClass` class to create the instance on the local system. Then use the `CIMClient.setInstance` method to update an existing instance in a namespace or use the `CIMClient.createInstance` method to add a new instance to a namespace.

Example — Getting Instances

The code segment in Example 4–8 lists all processes on a given system. This example uses the `enumerateInstanceNames` method to get the names of instances of the `CIM_Process` class. Running this code on a Microsoft Windows 32 system returns Windows 32 processes. Running this same code on a Solaris system returns Solaris processes.

EXAMPLE 4–8 Getting Instances of a Class (`getInstance`)

```
...
{
//Create namespace cns
CIMnameSpace cns = new CIMNameSpace();

//Connect to the cns namespace on the CIM Object Manager
cc = new CIMClient(cns, "root", "root_password");
```

EXAMPLE 4–8 Getting Instances of a Class (`getInstance`) *(Continued)*

```
/* Pass the CIM Object Path of the CIM_Process class
to the CIM Object Manager. We want to get instances of
this class. */

CIMObjectPath cop = new CIMObjectPath("CIM_Process");

/* The CIM Object Manager returns an enumeration of
object paths, the names of instances of
the CIM_Process class. */
Enumeration e = cc.enumerateInstanceNames(cop);

/* Iterate through the enumeration of instance object paths.
Use the CIM Client getInstance class to get
the instances referred to by each object name. */

while(e.hasMoreElements()) {
    CIMObjectPath op = (CIMObjectPath)e.nextElement();
    // Get the instance. Returns only the properties
        // that are local to the instance (localOnly is true).
        CIMInstance ci = cc.getInstance(op, true);
        }
...
```

Example — Getting a Property

Example 4–9 prints the value of the lockspeed property for all Solaris processes. This code segment uses the following methods:

- `enumInstances` – to get the names of all instances of Solaris processor
- `getProperty` – to get the value of the lockspeed for each instance
- `println` – to print the lockspeed value

EXAMPLE 4–9 Printing Processor Information (`getProperty`)

```
...
{
/* Create an object (CIMObjectPath) to store the name of the
Solaris_Processor class. */

CIMObjectPath cop = new CIMObjectPath("Solaris_Processor");

/* The CIM Object Manager returns an enumeration containing the names
of instances of the Solaris_Processor class and
all its subclasses (cc.DEEP). */

Enumeration e = cc.enumInstances(cop, cc.DEEP);

/* Iterate through the enumeration of instance object paths.
Use the getProperty method to get the lockspeed
value for each Solaris processor. */
```

EXAMPLE 4–9 Printing Processor Information (`getProperty`) *(Continued)*

```
while(e.hasMoreElements()) {
            CIMValue cv = cc.getProperty(e.nextElement(), "lockspeed");
        System.out.println(cv);
}
...
}
```

Example — Setting a Property

The code segment in "Example — Setting a Property" on page 80 sets a hypothetical lockspeed value for all Solaris processors. This code segment uses the following methods:

- `enumInstances` – to get the names of all instances of Solaris processor
- `setProperty` – to set the value of the lockspeed for each instance

EXAMPLE 4–10 Setting Processor Information (`setProperty`)

```
...
{
    /* Create an object (CIMObjectPath) to store the name of the
    Solaris_Processor class. */

    CIMObjectPath cop = new CIMObjectPath("Solaris_Processor");

    /* The CIM Object Manager returns an enumeration containing the names
    of instances of the Solaris_Processor class and
    all its subclasses. */

    Enumeration e = cc.enumerateInstanceNames(cop);

    /* Iterate through the enumeration of instance object paths.
    Use the setProperty method to set the lockspeed
    value to 500 for each Solaris processor. */

    for (; e.hasMoreElements(); cc.setProperty(e.nextElement(), "lockspeed",
        new CIMValue(new Integer(500))));

...
}
```

Example — Setting Instances

The code segment in Example 4–11 gets a CIM instance, updates one of its property values, and passes the updated instances to the CIM Object Manager.

A CIM property is a value used to describe a characteristic of a CIM class. Properties can be thought of as a pair of functions, one to *set* the property value and one to *get* the property value.

EXAMPLE 4–11 Setting Instances (`setInstance`)

```
...
{
    // Create an object path, an object that contains the
    // CIM name for "myclass"
    CIMObjectPath cop = new CIMObjectPath("myclass");
    /* Get instances for each instance object path in an enumeration,
    update the property value of b to 10 in each instance,
    and pass the updated instance to the CIM Object Manager. */

    while(e.hasMoreElements()) {
            CIMInstance ci = cc.getInstance(CIMObjectPath)(e.nextElement(),
                true, true, true, null);
        ci.setProperty("b", new CIMValue(new Integer(10)));
                cc.setInstance(new CIMObjectPath(),ci);
        }
}
...
```

Enumerating Namespaces, Classes, and Instances

An enumeration is a collection of objects that can be retrieved one at a time. The Sun WBEM SDK provides APIs for enumerating namespaces, classes, and instances.

The following examples show how to use the enumeration methods to enumerate namespaces, a classes, and instances.

Deep and Shallow Enumeration

The enumeration methods take a Boolean argument that can have the value *deep* or *shallow*. The behavior of *deep* and *shallow* depends upon the particular method being used, as shown in Table 4–1.

TABLE 4–1 Deep and Shallow Enumeration

Method	*deep*	*shallow*
`enumNameSpace`	Returns the entire hierarchy of namespaces under the enumerated namespace.	Returns the first-level children of the enumerated namespace.
`enumClass`	Returns all subclasses of the enumerated class, but does not return the class itself.	Returns the direct subclasses of that class.
`enumInstances`	Returns the class instances and all instances of its subclasses.	Returns the instances of that class

Getting Class and Instance Data

The following enumeration methods return the class and instance data:

- `enumInstances`(*CIMObjectPath path, boolean deep, boolean localOnly*) – Returns the instances for the class specified in *Path*. If *deep* is true, this method returns the instances of the specified class and all classes derived from the class. If *shallow* is true, this method returns the instances of the specified class.

  When an instance of a class is created, it inherits the properties of the class it is derived from and all parent classes in the class hierarchy. If *localOnly* is true, `enumInstances` returns only non-inherited properties. If *localOnly* is false, all properties in the class are returned.
- `enumClass`(*CIMObjectPath path, boolean deep, boolean localOnly*) – Returns the classes (the entire class not just the name of the class) for the class specified in *Path*. If *deep* is true, this method returns all classes derived from the enumerated class. If *shallow* is true, this method returns only the first-level children of the enumerated class.

  When a class is created, it inherits the methods and properties of the class it is derived from and all parent classes in the class hierarchy. If *localOnly* is true, this method returns only non-inherited properties and methods. If *localOnly* is false, all properties in the class are returned.

Getting Class and Instance Names

CIM WorkShop is an example of an application that uses enumeration methods to return the *names* of classes and instances. Once you get a list of object names, you can get the instances of that object, its properties, or other information about the object.

The following enumeration methods return the names of the enumerated class or instance:

- `enumerateInstanceNames`(*CIMObjectPath path*) — Returns the names of the instances for the specified class.
- `enumerateClassNames`(*CIMObjectPath path, boolean deep*) — Returns the names of the classes for the class specified in *Path*. If *deep* is true, this method returns the names of all classes derived from the enumerated class. If *shallow* is true, this method returns only the names of the first-level children of the enumerated class.

Example — Enumerating Namespaces

The sample program in Example 4–12 uses the `enumNameSpace` method in the `CIMClient` class to print the names of the namespace and all the namespaces contained within the namespace.

EXAMPLE 4–12 Enumerating Namespaces (`enumNameSpace`)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import java.util.Enumeration;

/ **
  * This program takes a namespace argument and calls the
  * enumNameSpace CIMClient interface to get a list of the
  * namespaces within the namespace specified by the CIMObjectPath,
  * (cop) and all the namespaces contained in the namespace
  * (CIMClient.DEEP).  The program then prints the name of the specified
  * namespace (CIMClient.SHALLOW).
 /**

public class EnumNameSpace {

    // EnumNameSpace takes a string of arguments
    public static void main (String args[ ]) {
            CIMClient cc = null;

              try {
                  // Create a namespace object for the namespace passed as an argument
                  CIMNameSpace cns = new CIMNameSpace(args[0], "");

                  // Connect to the CIM Object Manager in the namespace passed as an argumer
                  CIMClient cc = new CIMClient(cns);

                  // Create an object path to store the namespace name on the current host
                  CIMObjectPath cop = new CIMObjectPath("",args[1]);
```

EXAMPLE 4–12 Enumerating Namespaces (`enumNameSpace`) *(Continued)*

```
                // Enumerate the namespace and all namespaces it contains
            // (deep is set to CIMClient.DEEP)
                Enumeration e = cc.enumNameSpace(cop, CIMClient.DEEP);

                // Iterate through the list of namespaces and print each name.
                for (; e.hasMoreElements();
                          System.out.println(e.nextElement()));
                          System.out.println("++++++");
                // Iterate through the list of namespaces (CIMClient.SHALLOW) and
            // print each name.
                e = cc.enumNamesSpace(cop, CIMClient.SHALLOW);
                for (; e.hasMoreElements();
                        System.out.println(e.nextElement()));
              } catch (Exception e) {
                       System.out.println("Exception: "+e);
                  }
          // If the client connection is open, close it.
              if(cc != null) {
             cc.close();
          }
      }
}
```

Example — Enumerating Class Names

A Java GUI application might use the code segment in Example 4–13 to display a list of classes and subclasses to a user.

EXAMPLE 4–13 Enumerating Class Names (`enumClass`)

```
...
{
    /* Creates a CIMObjectPath object and initializes it
    with the name of the CIM class to be enumerated (myclass). */
    CIMObjectPath cop = new CIMObjectPath(myclass);

    /* This enumeration contains the names of the classes and subclasses
    in the enumerated class. */
    Enumeration e = cc.enumClass(cop, cc.DEEP);
}
...
```

An application might use the code segment in Example 4–14 to display the contents of a class and its subclasses.

EXAMPLE 4–14 Enumerating Class Data (`enumClass`)

```
...
{
    /* Creates a CIMObjectPath object and initializes it
```

EXAMPLE 4–14 Enumerating Class Data (`enumClass`) *(Continued)*

```
    with the name of the CIM class to be enumerated (myclass). */

    CIMObjectPath cop = new CIMObjectPath(myclass);

    /* This enumeration contains the classes and subclasses
    in the enumerated class (cc.DEEP). This enumeration
    returns only the non-inherited methods and properties
    for each class and subclass (localOnly is true).*/

    Enumeration e = cc.enumClass(cop, cc.DEEP, true);
}
...
```

The sample program in Example 4–15 does a deep and shallow enumeration of classes and instances. This example also shows the use of the *localOnly* flag to return class and instance data, instead of returning the names of the classes and instances.

EXAMPLE 4–15 Enumerating Classes and Instances

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * This example enumerates classes and instances. It does
 * a deep and shallow enumeration of a class that is passed
 * from the command line. It uses the localOnly flag to return
 * class and instance details.
 */
 public class ClientEnum {
     public static void main(String args[]) throws CIMException {
             CIMClient cc = null;
               CIMObjectPath cop = null;
               if(args.length != 2) {
                     System.out.println("Usage: ClientEnum host className");
                     System.exit(1);
                 }
               try {
                    // Create a CIMNameSpace object that contains the
                    // hostname (args[0] from the command line).
                    CIMNameSpace cns = new CIMNameSpace(args[0]);

                    // Creates a client connection to the CIM Object Manager
                    // on the specified host (args[0]).
```

EXAMPLE 4–15 Enumerating Classes and Instances *(Continued)*

```
cc = new CIMClient(cns);

// Get the class name from the command line
cop = new CIMObjectPath(args[1]);

// Do a deep enumeration of the class, which
// returns the class names.
Enumeration e = cc.enumClass(cop, cc.DEEP);

// Print the names of all subclasses of the enumerated class.
for (; e.hasMoreElements(); System.out.println(e.nextElement()));
    System.out.println("+++++");

// Do a shallow enumeration of the class, which
// returns the class names.
e = cc.enumClass(cop, cc.SHALLOW);

// Prints the names of the first-level subclasses.
for (; e.hasMoreElements(); System.out.println(e.nextElement()));
    System.out.println("+++++");

// Do a shallow enumeration of the class, which
// returns the class data, not just the class
// name (localOnly is true).
e = cc.enumClass(cop, cc.SHALLOW, true);

// Prints the details of the first-level subclasses.
for (; e.hasMoreElements(); System.out.println(e.nextElement()));
    System.out.println("+++++");

// Do a deep enumeration of the instances of the class, which
// returns the names of the instances.
e = cc.enumInstances(cop, cc.DEEP);

// Prints the names of all instances of the class and its subclasses.
for (; e.hasMoreElements(); System.out.println(e.nextElement()));
    System.out.println("+++++");

// Do a deep enumeration of the instances of the class, which
// returns the actual instance data, not just the instance
// name. (localOnly is true).
e = cc.enumInstances(cop, cc.DEEP);

// Prints the details of the instances of the class and its subclasses.
for (; e.hasMoreElements(); System.out.println(e.nextElement()));
    System.out.println("+++++");

// Do a shallow enumeration of the instances of the class,
// which returns the names of the instances.
e = cc.enumInstances(cop, cc.SHALLOW);

// Prints the names of the instances of the class.
for (; e.hasMoreElements(); System.out.println(e.nextElement()));
```

EXAMPLE 4–15 Enumerating Classes and Instances *(Continued)*

```
                    System.out.println("+++++");
                  } catch (Exception e) {
                        System.out.println("Exception: "+e);
                 }
                // close session.
                if(cc != null) {
                    cc.close();
                }
    }
}
```

Querying

The enumeration APIs return all instances in a class or class hierarchy. You can choose to return the instance names or the details of the instance. Querying allows you to narrow your search by specifying a query string. You can search for instances that match a specified query in a particular class or in all classes in a particular namespace. For example, you can search for all instances of the `Solaris_DiskDrive` class that have a particular value for the `Storage_Capacity` property.

The `execQuery` Method

The `execQuery` method retrieves an enumeration of CIM instances that match a query string. The query string must be formed using the WBEM Query Language (WQL).

Syntax

The syntax for the `execQuery` method is:

```
Enumeration execQuery(CIMObjectPath relNS, java.lang.String query, int ql)
```

The `execQuery` method takes the following parameters and returns an enumeration of CIM instances:

Parameter	Data Type	Description
relNS	CIMObjectPath	The namespace relative to the namespace to which you are connected. For example, if you are connected to the `root` namespace and want to query classes in the `root\cimv2` namespace, you would pass `new CIMObjectPath("", "cimv2");`.
query	String	The text of the query in WBEM Query Language
ql	Integer constant	Identifies the query language. WQL level 1 is the only currently supported query language.

Example

The following `execQuery` call returns an enumeration of all instances of the `CIM_device` class in the current namespace.

```
cc.execQuery(new CIMObjectPath(), SELECT * FROM CIM_device, cc.WQL)
```

Using the WBEM Query Language

The WBEM Query Language is a subset of standard American National Standards Institute Structured Query Language (ANSI SQL) with semantic changes to support WBEM on Solaris. Unlike SQL, in this release WQL is a retreval-only language. You cannot use WQL to modify, insert, or delete information.

SQL was written to query databases, in which data is stored in tables with a row-column structure. WQL has been adapted to query data that is stored using the CIM data model. In the CIM model, information about objects is stored in CIM classes and CIM instances. CIM instances can contain properties, which have a name, data type, and value. WQL maps the CIM object model to SQL tables.

TABLE 4–2 Mapping of SQL to WQL Data

SQL	Is Represented in WQL as...
Table	CIM class
Row	CIM instance
Column	CIM property

Supported WQL Key Words

The Sun WBEM SDK supports Level 1 WBEM SQL, which enables simple select operations without joins. The following table describes the WQL key words supported in the Sun WBEM SDK.

TABLE 4–3 Supported WQL Key Words

Key Word	Description
AND	Combines two Boolean expressions and returns TRUE when both expressions are TRUE.
FROM	Specifies the classes that contain the properties listed in a SELECT statement.
NOT	Comparison operator used with NULL.
OR	Combines two conditions. When more than one logical operator is used in a statement, OR operators are evaluated after AND operators.
SELECT	Specifies the properties that will be used in a query.
WHERE	Narrows the scope of a query.

WBEM Query Language Operators

The following table lists the standard WQL operators that can be used in the WHERE clause of a SELECT statement.

TABLE 4–4 WQL Operators

Operator	Description
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to

Making a Data Query

Data queries are statements that request instances of classes. To issue a data query, applications use the `execQuery` method to pass a WBEM Query Language string to the CIM Object Manager.

The SELECT Statement

The SELECT statement is the SQL statement for retrieving information, with a few restrictions and extensions specific to WQL. Although the SQL SELECT statement is typically used in the database environment to retrieve particular columns from tables, the WQL SELECT statement is used to retrieve instances of a single class. WQL does not support queries across multiple classes.

The SELECT statement specifies the properties to query in an object specified in the FROM clause.

The basic syntax for the SELECT statement is:

`SELECT` *instance* `FROM` *class*

The following tables shows examples of using arguments in the SELECT clause to refine a search.

TABLE 4–5 SELECT Statement

Example Query	Description
SELECT * FROM *class*	Selects all instances of the specified class and any of its subclasses.
SELECT PropertyA FROM class	Selects only instances of the specified class and any of its subclasses that contain `PropertyA`.
SELECT PropertyA, PropertyB FROM *class*	Selects only instances of the specified class and any of its subclasses that contain `PropertyA` or `PropertyB`.

The WHERE Clause

You can use the WHERE clause to narrow the scope of a query. The WHERE clause can contain a property or key word, an operator, and a constant. All WHERE clauses must specify one of the predefined WQL operators.

The basic syntax for appending the WHERE clause to the SELECT statement is:

`SELECT` *instance* `FROM` *class* `WHERE` *expression*

The expression is composed of a property or key word, an operator, and a constant. You can append the WHERE clause to the SELECT statement using one of the following forms:

`SELECT` *instance* `FROM` *class* `[WHERE` *property operator constant*`]`

`SELECT` *instance* `FROM` *class* `[WHERE` *constant operator property*`]`

Valid WHERE clauses follow these rules:

- The value of the constant must be of the correct data type for the property.
- The operator must be one of the valid WQL operators listed in Table 4–4.
- Either a property name or a constant must appear on either side of the operator in the WHERE clause.
- Arbitrary arithmetic expressions cannot be used. For example, the following query returns only instances of the `Solaris_Printer` class that represent a printer with ready status:

  ```
  SELECT * FROM Solaris_Printer WHERE Status = “ready”
  ```

  The following is an invalid query:

  ```
  SELECT * FROM PhysicalDisk WHERE Partitions < (8 + 2 - 2)
  ```

Multiple groups of properties, operators, and constants can be combined in a WHERE clause using logical operators and parenthetical expressions. Each group must be joined with the AND, OR, or NOT operators as shown in the following table.

TABLE 4–6 Queries Using Logical Operators

Example Query	Description
SELECT * FROM Solaris_FileSystem WHERE Name= "home" OR Name= "files"	Retrieves all instances of the `Solaris_FileSystem` class with the Name property set to either `home` or `files`.
SELECT * FROM Solaris_FileSystem WHERE (Name = “home” OR Name = “files”) AND AvailableSpace > 2000000 AND FileSystem = “Solaris”	Retrieves disks named `home` and `files` only if they have a certain amount of available space remaining and have Solaris file systems.

Associations

This section explains the CIM concept of associations and the `CIMClient` methods you can use to get information about associations.

About Associations

An association describes a relationship between two or more managed resources, for example a computer and the system disk it contains. This relationship is described in an association class, a special type of class that contains an association qualifier.

An association class also contains two or more references to the CIM instances representing its managed resources. A reference is a special property type that is declared with the REF keyword, indicating that it is a pointer to other instances. A reference defines the role each managed resource plays in an association.

The following figure shows two classes, `Teacher` and `Student`. Both classes are linked by the association, `TeacherStudent`. The `TeacherStudent` association has two references: `Teaches`, a property that refers to instances of the `Teacher` class and `TaughtBy`, a property that refers to instances of the `Student` class.

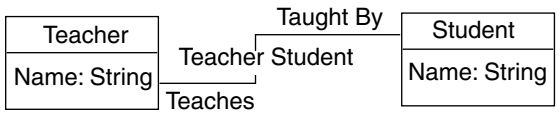

FIGURE 4–1 An Association Between Teacher and Student

You must delete an association before deleting one of its references. You can add or change the association between two or more objects without affecting the objects.

The Association Methods

The following methods in the `CIMClient` class return information about associations (relationships) between classes and instances:

TABLE 4–7 The `CIMClient` Association Methods

Method	Description
`associators`	Gets the CIM classes or instances that are associated with the specified CIM class or instance.
`associatorNames`	Gets the names of the CIM classes or instances that are associated with the specified CIM class or instance.
`references`	Gets the associations that refer to the specified CIM class or instance.
`referenceNames`	Gets the names of the associations that refer to the specified CIM class or instance.

Specifying the Source Class or Instance

The association methods each take one required argument, `CIMObjectPath`, which is the name of a source CIM class or CIM instance whose associations or associated classes or instances you want to return. If the CIM Object Manager finds no associations or associated classes or instances, it returns nothing.

If the `CIMObjectpath` is a class, the association methods return the associated classes and the subclasses of each associated class. If the `CIMObjectpath` is an instance, the methods return the associated instances and the class from which each instance is derived.

Using the Model Path to Specify an Instance

To specify the name of an instance or class, you must specify its model path. The model path for a class includes the namespace and class name. The model path for an instance uniquely identifies a particular managed resource. The model path for an instance includes the namespace, class name, and keys. A key is a property or set of properties used to uniquely identify managed resource. Key properties are marked with the KEY qualifier.

The model path `\\myserver\\Root\cimv2\Solaris_ComputerSystem.Name=mycomputer:CreationClassName=Solaris_ComputerSystem` has three parts:

- `\\myserver\Root\cimv2` – The default CIM namespace on host `myserver`.
- `Solaris_ComputerSystem` – The name of the class from which the instances is derived.
- `Name=mycomputer, CreationClassName=Solaris_ComputerSystem` – Two key properties in the form key property = value.

Using the APIs to Specify an Instance

In practice, you will usually use the `enumInstances` method to return all instances of a given class. Then, use a loop structure to iterate through the instances In the loop, you can pass each instance to an association method. The code segment in the following example does the following:

1. Enumerates the instances in the current class (`op`) and the subclasses of the current class.
2. Uses a While loop to cast each instance to a `CIMObjectPath` (`op`),
3. Passes each instance as the first argument to the `associators` method.

This code example passes null or false values for all other parameters.

EXAMPLE 4–16 Passing Instances to the associators Method

```
{
    ...
```

EXAMPLE 4–16 Passing Instances to the associators Method *(Continued)*

```
Enumeration e = cc.enumInstances(op, true);
while (e.hasMoreElements()) {
    op = (CIMObjectPath)e.nextElement();
    Enumeration e1 = cc.associators(op, null, null,
            null, null, false, false, null);
...
}
```

Using Optional Arguments to Filter Returned Classes and Instances

The association methods also take the following optional arguments, which filter the classes and instances that are returned. Each optional parameter value passes its results to the next parameter for filtering until all arguments have been processed.

You can pass values for any one or a combination of the optional arguments. You must enter a value for each parameter. The `assocClass`, `resultClass`, `role`, and `resultRole` arguments filter the classes and instances that are returned. Only the classes and instances that match the values specified for these parameters are returned. The `includeQualifiers`, `includeClassOrigin`, and `propertyList` arguments filter the information that is included in the classes and instances that are returned.

The following table lists the optional arguments to the association methods:

TABLE 4–8 Optional Arguments to the Association Methods

Argument	Type	Description	Value
`assocClass`	String	Returns target objects that participate in this type of association with the source CIM class or instance. If Null, does not filter returned objects by association.	Valid CIM association class name or Null.
`resultClass`	String	Returns target objects that are instances of the `resultClass` or one of its subclasses, or objects that match the `resultClass` or one of its subclasses.	Valid name of a CIM class or Null.

TABLE 4–8 Optional Arguments to the Association Methods *(Continued)*

Argument	Type	Description	Value
role	String	Specifies the role played by the source CIM class or instance in the association. Returns the target objects of associations in which the source object plays this role.	Valid property name or Null.
resultRole	String	Returns target objects that play the specified role in the association.	Valid property name or Null.
`includeQualifiers`	Boolean	If true, returns all qualifiers for each target object (qualifiers on the object and any returned properties). If false, returns no qualifiers.	True or False.
includeClassOrigin	Boolean	If true, includes the CLASSORIGIN attribute in all appropriate elements in each returned object. If false, excludes CLASSORIGIN attributes.	True or False.
propertyList	String array	Returns objects that include only elements for properties on this list. If an empty array, no properties are included in each returned object. If NULL, all properties are included in each returned object. Invalid property names are ignored. If you specify a property list, you must specify a non-Null value for `resultClass`.	An array of valid property names. an empty array, or Null.

Examples — `associators` and `associatorNames` Methods

The examples in this section show how to use the `associators` and `associatorNames` methods to get information about the classes associated with the `Teacher` and `Student` classes shown in the following figure. Notice that the `associatorNames` method does not take the arguments *includeQualifiers*, *includeClassOrigin*, and *propertyList* because these arguments are irrelevant to a method that returns only the names of instances or classes, not their entire contents.

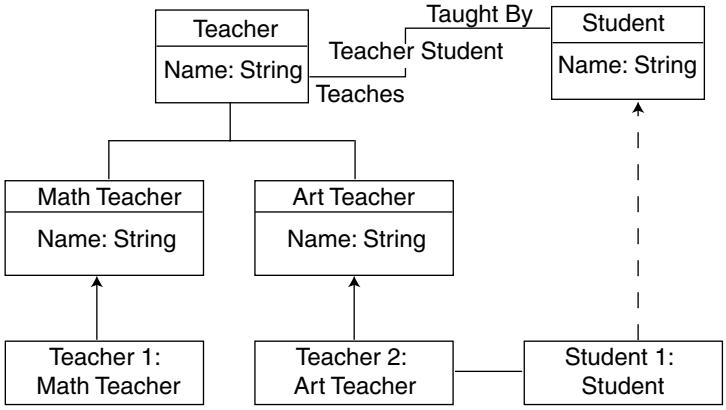

FIGURE 4–2 Teacher-Student Association Example

TABLE 4–9 `associators` and `associatorNames` Methods

Example	Output	Description
`associators(Teacher, null, null, null, null, false, false, null)`	`Student` class	Returns associated classes and their subclasses.`Student` is linked to `Teacher` by the `TeacherStudent` association.
`associators(Student, null, null, null, null, false, false, null)`	`Teacher`, `MathTeacher`, and `ArtTeacher` classes	Returns associated classes and their subclasses. `Teacher` is linked to `Student` by the `TeacherStudent` association. `MathTeacher` and `ArtTeacher` inherit the `TeacherStudent` association from `Teacher`.
`associatorNames (Teacher, null, null, null, null)`	Name of the `Student` class	Returns the names of the associated classes and their subclasses`Student` is linked to `Teacher` by the `TeacherStudent` association.
`associatorNames (Student, null, null, null, null)`	`Teacher`, `MatchTeacher`, and `ArtTeacher` class names.	Returns the names of the associated classes and their subclasses. `Teacher` is linked to `Student` by the `TeacherStudent` association. `MatchTeacher` and `ArtTeacher` inherit the `TeacherStudent` association from `Teacher`.

Examples — `references` and `referenceNames` Methods

The examples in this section show how to use the `references` and `referenceNames` methods to get information about the associations between the `Teacher` and `Student` classes in Figure 4–2. Notice that the `referenceNames` method does not take the arguments *includeQualifiers*, *includeClassOrigin*, and *propertyList* because these arguments are irrelevant to a method that returns only the names of instances or classes, not their entire contents.

TABLE 4–10 `references` and `referenceNames` Methods

Example	Output	Comments
`references(Student, null, null. false, false, null)`	`TeacherStudent`	Returns the associations in which `Student` participates.
`references(Teacher, null, null. false, false, null)`	`TeacherStudent`	Returns the associations in which `Teacher` participates.
`referenceNames(Teacher, null, null)`	The name of the `TeacherStudent` class.	Returns the names of the associations in which `Teacher` participates.

Calling Methods

Use the `invokeMethod` interface to call a method in a class supported by a provider. To retrieve the signature of a method, an application must first get the definition of the class to which the method belongs. The `invokeMethod` interface takes four arguments described in the following table:

TABLE 4–11 Parameters to the `invokeMethod`Method

Parameter	Data Type	Description
name	CIMObjectPath	The name of the instance on which the method must be invoked.
methodName	String	The name of the method to call.
inParams	Vector	Input parameters to pass to the method.

TABLE 4–11 Parameters to the `invokeMethod`Method *(Continued)*

Parameter	Data Type	Description
outParams	Vector	Output parameters to get from the method.

The `invokeMethod` method returns a `CIMValue`. The return value is null when the method you invoke does not define a return value.

Example — Calling a Method

The code segment in Example 4–17 gets the instances of the `CIM_Service` class (services that manage device or software features) and uses the `invokeMethod` method to stop each service.

EXAMPLE 4–17 Calling a Method (`invokeMethod`)

```
{
    ...
    /* Pass the CIM Object Path of the CIM_Service class
    to the CIM Object Manager. We want to invoke a method defined in
    this class. */

    CIMObjectPath op = new CIMObjectPath("CIM_Service");

    /* The CIM Object Manager returns an enumeration of instance
    object paths, the names of instances of the CIM_Service
    class. */

    Enumeration e = cc.enumInstances(op, cc.DEEP);

    /* Iterate through the enumeration of instance object paths.
    Use the CIM Client getInstance class to get
    the instances referred to by each object path. */

    while(e.hasMoreElements()) {
            // Get the instance
            CIMInstance ci = cc.getInstance(e.nextElement(), true);
            //Invoke the Stop Service method to stop the CIM services.
            cc.invokeMethod(ci, "StopService", null, null);
        }
}
```

Retrieving Class Definitions

Use the `getClass` method to get a CIM class. When a class is created, it inherits the methods and properties of the class it is derived from and all parent classes in the class hierarchy. The `getClass` method takes the Boolean argument *localOnly*. If *localOnly*is true, this method returns only non-inherited properties and methods. If *localOnly* is false, all properties in the class are returned.

Example — Retrieving a Class Definition

The code shown in Example 4–18, uses the following methods to retrieve a class definition:

- `CIMNameSpace` – to create a new namespace
- `CIMClient` – to create a new client connection to the CIM Object Manager
- `CIMObjectPath` – to create an object path, an object to contain the name of the class to retrieve
- `getClass` – to retrieve the class from the CIM Object Manager

EXAMPLE 4–18 Retrieving a Class Definition (`getClass`)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;
/**
 * Gets the class specified in the command line. Works in the default
 * namespace root\cimv2.
 */
public class GetClass {
    public static void main(String args[]) throws CIMException {
        CIMClient cc = null;
        try {
            CIMNameSpace cns = new CIMNameSpace(args[0]);
            cc = new CIMClient(cns);
            CIMObjectPath cop = new CIMObjectPath(args[1]);
                        // Returns only the methods and properties that
                        // are local to the specified class (localOnly is true).
            cc.getClass(cop, cc.DEEP);
        } catch (Exception e) {
            System.out.println("Exception: "+e);
```

EXAMPLE 4–18 Retrieving a Class Definition (`getClass`) *(Continued)*

```
        }
        if(cc != null) {
            cc.close();
        }
    }
}
```

Handling Exceptions

Each interface has a throws clause that defines a CIM Exception. An exception is an error condition. The CIM Object Manager uses Java exception handling and creates a hierarchy of WBEM-specific exceptions. The `CIMException` class is the base class for CIM exceptions. All other CIM exception classes extend from the `CIMException` class.

Each class of CIM exceptions defines a particular type of error condition that API code handles. See Table 3–2 for a description of the CIM exception APIs.

Using the Try/Catch Clauses

The Client API uses standard Java try/catch clauses to handle exceptions. Generally, an application catches exceptions and either takes some corrective action or passes some information about the error to the user.

The CIM rules are not explicitly identified in the CIM specification. In many cases, they are implied by example. In many cases, the error code refers to a general problem, for example, a data type mismatch, but the programmer must figure out what the correct data type is for the data.

Syntactic and Semantic Error Checking

The MOF Compiler (`mofc`) compiles `.mof` text files into Java classes (bytecode). The MOF Compiler does syntactical checking of the `MOF` files. The CIM Object Manager does semantic and syntactical checking because it can be accessed by many different applications.

The MOF file in Example 4–19 defines two classes, A and B. If you compiled this example file, the CIM Object Manager would return a semantic error because only a key can override another key.

EXAMPLE 4–19 Semantic Error Checking

```
Class A         \\Define Class A
 {     [Key]
int a;
}
Class B:A       \\Class B extends A
{  [overrides ("c", key (false)) ]
int b;
}
```

Advanced Programming Topics

This section describes advanced programming operations and operations that you would use less frequently.

Creating a Namespace

The installation compiles the standard CIM MOF files into the default namespaces, `root\cimv2` and `root\security`. If you create a new namespace, you must compile the appropriate CIM MOF files into the new namespace before creating objects in it. For example, if you plan to create classes that use the standard CIM elements, compile the CIM Core Schema into the namespace. If you plan to create classes that extend the CIM Application Schema, compile the CIM Application into the namespace.

Example — Creating a Namespace

The code segment in Example 4–20 uses a two-step process to create a namespace within an existing namespace.

- First, it uses the `CIMNameSpace` method to construct a namespace object. This namespace object contains the parameters to be passed to the CIM Object Manager when the namespace is actually created.
- Second, the example uses the `CIMClient` class to connect to the CIM Object Manager and pass it the namespace object. The CIM Object Manager creates the namespace, using the parameters contained in the namespace object.

EXAMPLE 4–20 Creating a Namespace (`CIMNameSpace`)

```
{
    ...
    /* Creates a namespace object on the client, which stores parameters
```

EXAMPLE 4–20 Creating a Namespace (`CIMNameSpace`) *(Continued)*

```
    passed to it from the command line. args[0] contains the host
    name (for example, myhost); args[1] contains the
    parent namespace (for example, the toplevel directory.) */

    CIMNameSpace cns = new CIMNameSpace (args[0], args[1]);

    /* Connects to the CIM Object Manager and passes it three parameters:
    the namespace object (cns), which contains the host name (args[0]) and
    parent namespace name (args[1]), a user name string (args[3]), and a
    password string (args[4]). */

    CIMClient cc = new CIMClient (cns, "root", "secret");

    /* Passes to the CIM Object Manager another namespace object that
    contains a null string (host name) and args[2], the name of a
    child namespace (for example, secondlevel). */

    CIMNameSpace cop = new CIMNameSpace("", args[2]);

    /* Creates a new namespace called secondlevel under the
    toplevel namespace on myhost./*

    cc.createNameSpace(cop);
    ...
}
```

Deleting a Namespace

Use the `deleteNameSpace` method to delete a namespace.

Example — Deleting a Namespace

The sample program in Example 4–21, deletes the specified namespace on the specified host. The program takes five required string arguments (host name, parent namespace, child namespace, username, and password). The user running this program must specify the username and password for an account that has write permission to the namespace to be deleted.

EXAMPLE 4–21 Deleting a Namespace (`deleteNameSpace`)

```
{
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
```

EXAMPLE 4–21 Deleting a Namespace (`deleteNameSpace`) *(Continued)*

```
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;
 /**
 * This example program deletes the specified namespace on the
 * specified host. The user running this program must specify
 * the username and password for a user account that has write
 * permission for the specified namespace.
 */
 public class DeleteNameSpace {
    public static void main(String args[]) throws CIMException {
        // Initialize an instance of the CIM Client class
                CIMClient cc = null;
        // Requires 5 command-line arguments. If not all entered,
        // prints command string.
                if(args.length != 5) {
                    System.out.println("Usage: DeleteNameSpace host parentNS
                              childNS username password");
                      System.exit(1);
                }
                try {
                 /**
                         * Creates a namespace object (cns), which stores the host
                         * name and parent namespace.
                         */
                       CIMNameSpace cns = new CIMNameSpace(args[0], args[1]);
                     /**
                         * Connects to the CIM Object Manager, and passes it the
                         * namespace object (cns) and the username and password
                      * command line arguments.
                      */
                      cc = new CIMClient(cns, args[3], args[4]);
                     /**
                         * Creates another namespace object (cop), which stores the
                         * a null string for the host name and a string for the
                         * child namespace (from the command line arguments).
                         */
                        CIMNameSpace cop = new CIMNameSpace("",args[2]);
                         /**
                         * Deletes the child name space under the parent namespace.
                    */
                       cc.deleteNameSpace(cop);
                } catch (Exception e) {
                    System.out.println("Exception: "+e);
                }
                 // Close the session
                if(cc != null) {
                   cc.close();
                }
        }
}
```

EXAMPLE 4–21 Deleting a Namespace (`deleteNameSpace`) *(Continued)*

Creating a Base Class

Applications can create classes using either the MOF language or the client APIs. If you are familiar with MOF syntax, use a text editor to create a MOF file and then use the MOF Compiler to compile it into Java classes. This section describes how to use the client APIs to create a base class.

Use the `CIMClass` class to create a Java class representing a CIM class. To declare the most basic class, you need only specify the class name. Most classes include properties that describe the data of the class. To declare a property, include the property's data type, name, and an optional default value. The property data type must be an instance of `CIMDataType` (one of the predefined CIM data types).

A property can have a *key* qualifier, which identifies it as a key property. A key property uniquely defines the instances of the class. Only keyed classes can have instances. Therefore, if you do not define a key property in a class, the class can only be used as an abstract class.

If you define a key property in a class in a new namespace, you must first compile the core MOF files into the namespace. The core MOF files contain the declarations of the standard CIM qualifiers, such as the *key* qualifier.

Class definitions can be more complicated, including such MOF features as aliases, qualifiers, and qualifier flavors.

Example — Creating a CIM Class

The example in Example 4–22 creates a new CIM class in the default namespace (`root\cimv2`) on the local host. This class has two properties, one of which is the key property for the class. The example then uses the `newInstance` method to create an instance of the new class.

EXAMPLE 4–22 Creating a CIM Class (`CIMClass`)

```
{
...
    /* Connect to the root\cimv2 namespace
    on the local host and create a new class called myclass */

    // Connect to the default namespace on local host.
    CIMClient cc = new CIMClient();

    // Construct a new CIMClass object
    CIMClass cimclass = new CIMClass();

    // Set CIM class name to myclass.
    cimclass.setName("myclass");
```

EXAMPLE 4–22 Creating a CIM Class (`CIMClass`) *(Continued)*

```
// Construct a new CIM property object
CIMProperty cp = new CIMProperty();

// Set property name
cp.setName("keyprop");

// Set property type to one of the predefined CIM data types.
cp.setType(CIMDatatype.getPredefinedType(CIMDataType.STRING));

// Construct a new CIM Qualifier object
CIMQualifier cq = new CIMQualifier();

// Set the qualifier name
cq.setName("key");

// Add the new key qualifier to the property
cp.addQualfier(cq);

/* Create an integer property initialized to 10 */

// Construct a new CIM property object
CIMProperty mp = new CIMProperty();

// Set property name to myprop
mp.setName("myprop");

// Set property type to one of the predefined CIM data types.
mp.setType(CIMDatatype.getPredefinedType(CIMDataType.SINT16));

// Initialize mp to a CIMValue that is a new Integer object
// with the value 10. The CIM Object Manager converts this
// CIMValue to the CIM Data Type (SINT16) specified for the
// property in the mp.setType statement in the line above.
// If the CIMValue (Integer 10) does not fall within the range
// of values allowed for the CIM Data Type of the property
// (SINT16), the CIM Object Manager throws an exception.
mp.setValue(new CIMValue(new Integer(10)));

/* Add the new properties to myclass and call
the CIM Object Manager to create the class. */

// Add the key property to class object
cimclass.addProperty(cp);

// Add the integer property to class object
cimclass.addProperty(mp);

/* Connect to the CIM Object Manager and pass the new class */
cc.createClass(new CIMObjectPath(),cimclass);

// Create a new CIM instance of myclass
ci = cc.newInstance();
```

EXAMPLE 4–22 Creating a CIM Class (`CIMClass`) *(Continued)*

```
    // If the client connection is open, close it.
        if(cc != null) {
        cc.close();
    }
}
```

Deleting a Class

Use the `CIMClient deleteClass` method to delete a class. Deleting a class removes the class, its subclasses, and all instances of the class; it does not delete any associations that refer to the deleted class.

Example — Deleting a Class

The example in Example 4–23 uses the `deleteClass` method to delete a class in the default namespace `root\cimv2`. This program takes four required string arguments (host name, class name, username, and password). The user running this program must specify the username and password for an account that has write permission to the `root\cimv2`namespace.

EXAMPLE 4–23 Deleting a Class (`deleteClass`)

```
        import java.rmi.*;
        import com.sun.wbem.client.CIMClient;
        import com.sun.wbem.cim.CIMInstance;
        import com.sun.wbem.cim.CIMValue;
        import com.sun.wbem.cim.CIMProperty;
        import com.sun.wbem.cim.CIMNameSpace;
        import com.sun.wbem.cim.CIMObjectPath;
        import com.sun.wbem.cim.CIMClass;
        import com.sun.wbem.cim.CIMException;
        import java.util.Enumeration;
     /**
        * Deletes the class specified in the command line. Works in the default
        * namespace root\cimv2.
        */
        public class DeleteClass {
          public static void main(String args[]) throws CIMException {
                  CIMClient cc = null;
                  if(args.length != 4) {
                          System.out.println("Usage:
                        DeleteClass host className username password");
                          System.exit(1);
                    }
                    try {
                         /**
```

EXAMPLE 4–23 Deleting a Class (`deleteClass`) *(Continued)*

```
                        * Creates a namespace object (cns), which stores the host
                        * name.
                           */
                            CIMNameSpace cns = new CIMNameSpace(args[0]);

                        /**
                            * Connects to the CIM Object Manager, and passes it the
                            * namespace object (cns) and the username and password
                       * command line arguments.
                       */
                            cc = new CIMClient(cns, args[2], args[3]);

                          /**
                             * Create an object (CIMObjectPath) that
                             * contains the name of the class specified in args[1].
                             */
                            CIMObjectPath cop = new CIMObjectPath(args[1]);

                            /**
                             * Delete the class referenced by the CIM object path.
                             */
                                cc.deleteClass(cop);
                       } catch (Exception e) {
                              System.out.println("Exception: "+e);
                       }
                       if(cc != null) {
                            cc.close();
                  }
          }
}
```

Working with Qualifier Types and Qualifiers

A CIM qualifier is an element that characterizes a CIM class, instance, property, method, or parameter. Qualifiers have the following attributes:

- Type
- Value
- Name

In Managed Object Format syntax, each CIM qualifier must have a CIM qualifier type declared in the same MOF file. Qualifiers do not have a scope attribute. Scope indicates which CIM elements can use the qualifier. Scope can only be defined in the qualifier type declaration; it cannot be changed in a qualifier.

The following sample code shows the MOF syntax for a CIM qualifier type declaration. This statement defines a qualifier type named `key`, with a Boolean data type (default value false), which can describe only a property and a reference to an object. The `DisableOverride` flavor means that key qualifiers cannot change their value.

```
Qualifier Key : boolean = false, Scope(property, reference),
                                 Flavor(DisableOverride);
```

The following sample code shows the MOF syntax for a CIM qualifier. In this sample MOF file, `key` and description are qualifiers for the property `test`. The property data type is an integer with the value `a`.

```
{
[key, Description("test")]
int a
}
```

Example — Getting CIM Qualifiers

The code segment in Example 4–24 uses the `CIMQualifier` class to identify the CIM qualifiers in a vector of CIM elements. The example returns the property name, value, and type for each CIM Qualifier.

A qualifier flavor is a flag that governs the use of a qualifier. Flavors describe rules that specify whether a qualifier can be propagated to derived classes and instances and whether or not a derived class or instance can override the qualifier's original value.

EXAMPLE 4–24 Getting CIM Qualifiers (`CIMQualifier`)

```
    {
    ...
    } else if (tableType == QUALIFIER_TABLE) {
              CIMQualifier prop = (CIMQualifier)cimElements.elementAt(row);
              if (prop != null) {
                  if (col == nameColumn) {
                    return prop.getName();
                } else if (col == typeColumn) {
                      CIMValue cv = prop.getValue();
                    if (cv != null) {
                        return cv.getType().toString();
                    } else {
                        return "NULL";
            }
    }
...
```

Example — Setting CIM Qualifiers

Example 4–25 is a code segment that sets a list of CIM qualifiers for a new class to the qualifiers in its superclass.

EXAMPLE 4–25 Set Qualifiers (`setQualifiers`)

```
{
    ...
       try {
           cimSuperClass = cimClient.getClass(new CIMObjectPath(scName));
                Vector v = new Vector();
                for (Enumeration e = cimSuperClass.getQualifiers().elements();
                           e.hasMoreElements();) {
                       CIMQualifier qual = (CIMQualifier)((CIMQualifier)e.nextElement()).cl
                       v.addElement(qual);
                }
                cimClass.setQualifiers(v);
          } catch (CIMException exc) {
                    return;
              }
    }
}
...
```

Sample Programs

The examples directory contains sample programs that use the client API to perform a function. You can use these examples to start writing your own applications more quickly. The sample programs are described in Chapter 7.

To run a sample program, type the command:

`java` *program_name*

For example, `java createNameSpace`.

CHAPTER 5

Writing a Provider Program

This chapter describes how to write a provider, including the following topics:

- About Providers
- Implementing a Provider Interface
- Installing a Provider
- Registering a Provider
- Modifying a Provider
- Handling WBEM Query Language Queries

For detailed information on the provider APIs, see the Javadoc reference pages.

About Providers

Providers are classes that communicate with managed resources to access data. Providers forward this information to the CIM Object Manager for integration and interpretation. When the CIM Object Manager receives a request from a management application for data that is not available from the CIM Object Manager Repository, it forwards the request to a provider.

Object providers must be installed on the same machine as the CIM Object Manager. The CIM Object Manager uses object provider application programming interfaces (APIs) to communicate with locally installed providers.

When an application requests dynamic data from the CIM Object Manager, the CIM Object Manager uses the provider interfaces to pass the request to the provider.

Providers perform the following functions in response to a request from the CIM Object Manager:

- Map the native information format to CIM Java classes
  - Get information from a device

    - Pass the information to the CIM Object Manager in the form of CIM Java classes
- Map the information from CIM Java classes to native device format
    - Get the required information from the CIM Java class
    - Pass the information to the device in native device format

Types of Providers

Providers are categorized according to the types of requests they service. The Sun WBEM SDK supports four types of providers:

- Instance – Supply dynamic instances of a given class, for example, Solaris packages. Instance providers support one or more of the following operations:
    - Instance retrieval
    - Enumeration
    - Modification
    - Deletion
- Property – Supply dynamic property values, for example, disk space.
- Method – Supply methods of one or more classes. A method is a function that describes the behavior of a class. Methods must be implemented by a provider.
- Association – Supply instances of dynamic association classes.
- Event — Handle indications of CIM events. Event providers are described in Chapter 6.

A single provider can support instances, properties, methods, and associations, which can be convenient.

Most providers are pull providers, which means they maintain their own data, generating it dynamically when necessary. Pull providers have minimal interaction with the CIM Object Manager and the CIM Repository. The data managed by a pull provider typically changes frequently, requiring the provider to either generate the data dynamically or retrieve it from a local cache whenever an application issues a request. A provider can also contact the CIM Object Manager.

A single provider can act simultaneously as a class, instance, and method provider by proper registration and implementation of all relevant methods.

Implementing a Provider Interface

Providers implement a provider interface that supports the type of service specific to their role. In order to implement the interface, a provider class must first declare the interface in an `implements` clause, and then it must provide an implementation (a body) for all of the abstract methods of the interface.

The CIM Object Manager communicates with providers using the `initialize` method. The `initialize` method takes an argument of type `CIMOMhandle`, which is a reference to the CIM Object Manager. The `CIMOMhandle` class contains methods that providers can use to transfer data to and from the CIM Object Manager.

The following table describes the provider interfaces in the `com.sun.wbem.provider` package. Use the `InstanceProvider` interface in the `com.sun.wbem.provider20` package. You can include a method provider, instance provider, and property provider in a single Java class file, or store each provider in a separate file.

TABLE 5–1 Provider Interfaces

Interface	Description
`CIMProvider`	Base interface implemented by all providers.
`InstanceProvider`	Base interface implemented by instance providers. Instance providers serve dynamic instances of classes.
`MethodProvider`	Base interface implemented by method providers, which implement all methods of CIM classes.
`PropertyProvider`	Base interface implemented by property providers, which are used to retrieve and update dynamic properties. Dynamic data is not stored in the CIM Object Manager Repository.
`AssociatorProvider`	Base interface implemented by providers of instances of dynamic associations.
`Authorizable`	Marker interface indicates to the CIM Object Manager that the provider handles its own authorization checking.

The Instance Provider Interface (`InstanceProvider`)

The following table describes the methods in the instance provider interface in the Provider package (`com.sun.wbem.provider20`).

These methods each take the *op* argument, the `CIMObjectPath` of the specified CIM class or CIM instance. The object path includes the namespace, class name, and keys (if the object is an instance). The namespace is a directory that can contain other namespaces, classes, instances, and qualifier types. A key is a property that uniquely identifies an instance of a class. Key properties have a *KEY* qualifier.

For example, the following object path has two parts:

```
\\myserver\root\cimv2\Solaris_ComputerSystem:Name=mycomputer:   CreationClassName=Solaris_Computer
```

- `\\myserver\root\cimv2`

  The default CIM namespace on host `myserver`.

- `Solaris_ComputerSystem:Name=mycomputer: CreationClassName=Solaris_ComputerSystem`

  A specific Solaris Computer System object in the default namespace on host `myserver`. This Solaris computer system is uniquely identified by two key property values in the format (property=value):

  - `Name=mycomputer`
  - `CreationClassName=Solaris_ComputerSystem`

TABLE 5–2 `InstanceProvider` Interface Methods

Method	Description
CIMObjectPath `createInstance(CIMObjectPath op, CIMInstance ci)`	Creates the instance *ci*specified by *op*, if it does not exist. If the CIM instance already exists, the provider should throw `CIMInstanceException` with ID `CIM_ERR_ALREADY_EXISTS`.Returns the `CIMObjectPath` of the created instance.
void `deleteInstance(CIMObjectPath op)`	Deletes the instance specified in the object path (*op*).
Vector `enumInstances(CIMObjectPath path, boolean deep, CIMClass cc)`	Returns the *names* of the instances for the class specified in *path*. If *deep*is true, returns the names of all instances of the specified class and all classes derived from the class. Otherwise, returns only the names of instances belonging to the specified class. Providers that do not want to create instances from scratch can create a template for the new instance by calling the `newInstance()` method for the class to which the instance belongs (*cc*).

TABLE 5–2 `InstanceProvider` Interface Methods *(Continued)*

Method	Description
Vector `enumInstances(CIMObjectPath path, boolean deep, CIMClass cc, boolean localOnly)`	Returns the instances (the entire instance not just the name of the instance) for the class specified in *path*. Providers that do not want to create instances from scratch can create a template for the new instance by calling the `newInstance()` method for the class to which the instance belongs (*cc*). If *localOnly* is true, returns the local (non-inherited) properties in the enumerated instances. Otherwise, returns all inherited and local properties.
Vector `execQuery (CIMObjectPath op, String query, int ql, CIMClass cc`	Executes a query to retrieve CIM objects. This method returns a vector of CIM instances of the specified CIM class (*cc*)that match the specified query string.
CIMInstance `getInstance(CIMObjectPath op, CIMClass cc, boolean localOnly)`	Returns the instance specified in the object path (*op*). Providers that do not want to create instances from scratch can create a template for the new instance by calling the `newInstance()` method for the class to which the instance belongs (*cc*). If *localOnly* is true, only the local (non-inherited) properties are returned. Otherwise, returns all inherited and local properties.
void `setInstance(CIMInstance ci)`	Updates the specified CIM instance if it exists. If the instance does not exist, throws a `CIMInstanceException` with ID `CIM_ERR_NOT_FOUND`.

Example — Implementing an Instance Provider

The following example shows the Java source code for an instance provider, `SimpleInstanceProvider`, that implements the `enumInstances` and `getInstance` interfaces for the `Ex_SimpleInstanceProvider` class. For brevity, this example implements the `deleteInstance`, `createInstance`, `setInstance`, and `execQuery` interfaces by throwing a `CIMException`. In practice, an instance provider must implement all `InstanceProvider` interfaces.

EXAMPLE 5–1 `SimpleInstanceProvider` Instance Provider

```
/*
 * "@(#)SimpleInstanceProvider.java"
 */
import com.sun.wbem.cim.*;
import com.sun.wbem.client.*;
import com.sun.wbem.provider.CIMProvider;
import com.sun.wbem.provider20.InstanceProvider;
import com.sun.wbem.provider.MethodProvider;
```

EXAMPLE 5–1 `SimpleInstanceProvider` Instance Provider *(Continued)*

```
import java.util.*;
import java.io.*;

public class SimpleInstanceProvider implements InstanceProvider{
     static int loop = 0;
     public void initialize(CIMOMHandle cimom) throws CIMException {
     }
     public void cleanup() throws CIMException {
     }
     public Vector enumInstances(CIMObjectPath op, boolean deep, CIMClass cc,
            boolean localOnly) throws CIMException {
                return null;
     }
     /*
      * enumInstances:
      * The entire instances and not just the names are returned.
      * Deep or shallow enumeration is possible, however
      * currently the CIMOM only asks for shallow enumeration.
      */
     public Vector enumInstances(CIMObjectPath op, boolean deep, CIMClass cc)
             throws CIMException {
         if (op.getObjectName().equalsIgnoreCase("Ex_SimpleInstanceProvider"))
             {
             Vector instances = new Vector();
             CIMObjectPath cop = new CIMObjectPath(op.getObjectName(),
                     op.getNameSpace());
                 if (loop == 0){
                     cop.addKey("First", new CIMValue("red"));
                     cop.addKey("Last", new CIMValue("apple"));
                     // To delete this class, comment this following
                        // line and compile it.
                     instances.addElement(cop);
                     loop += 1;
                 } else {
                     cop.addKey("First", new CIMValue("red"));
                     cop.addKey("Last", new CIMValue("apple"));
                     // To delete this class, comment this following
                     // line and compile it.
                     instances.addElement(cop);
                     cop = new CIMObjectPath(op.getObjectName(),
                             op.getNameSpace());
                     cop.addKey("First", new CIMValue("green"));
                     cop.addKey("Last", new CIMValue("apple"));
                     // To delete this class, comment this following
                     // line and compile it.
                     instances.addElement(cop);
                 }
             return instances;
             }
         return new Vector();
     }

     public CIMInstance getInstance(CIMObjectPath op,
```

EXAMPLE 5–1 `SimpleInstanceProvider` Instance Provider *(Continued)*

```
            CIMClass cc, boolean localOnly) throws CIMException {
            if (op.getObjectName().equalsIgnoreCase("Ex_SimpleInstanceProvider"))
            {
                CIMInstance ci = cc.newInstance();
                ci.setProperty("First", new CIMValue("yellow"));
                ci.setProperty("Last", new CIMValue("apple"));
                return ci;
            }
           return new CIMInstance();
     }

    public Vector execQuery(CIMObjectPath op, String query, int ql, CIMClass cc)
            throws CIMException {
                throw(new CIMException(CIMException.CIM_ERR_NOT_SUPPORTED));
   }

    public void setInstance(CIMObjectPath op, CIMInstance ci)
            throws CIMException {
                throw(new CIMException(CIMException.CIM_ERR_NOT_SUPPORTED));
   }

    public CIMObjectPath createInstance(CIMObjectPath op, CIMInstance ci)
             throws CIMException {
                throw(new CIMException(CIMException.CIM_ERR_NOT_SUPPORTED));
   }

    public void deleteInstance(CIMObjectPath cp) throws CIMException {
        throw(new CIMException(CIMException.CIM_ERR_NOT_SUPPORTED));
    }
}
```

The Property Provider Interface (`PropertyProvider`)

The following table describes the methods in the property provider interface.

TABLE 5–3 `PropertyProvider` Interface Methods

Method	Description
CIMValue `getPropertyValue(CIMObjectPath` op, `String originClass, String propertyName)`	Returns a `CIMValue` containing the value of the property specified by *propertyName* for the instance specified in *op*. The *originClass* contains the name of the class in the class hierarchy that originally defined this property.

TABLE 5–3 `PropertyProvider` Interface Methods *(Continued)*

Method	Description
void `setPropertyValue(CIMObjectPath op, String originClass, String propertyName, CIMValue cv)`	Sets the value of the property specified by *propertyName* for the instance specified in *op* to the `CIMValue` *cv*. The *originClass* contains the name of the class in the class hierarchy that originally defined this property.

Example — Implementing a Property Provider

The code segment in Example 5–2 creates a property provider (`fruit_prop_provider`) class that is registered in Example 5–2. The `fruit_prop_provider` implements the `PropertyProvider` interface.

This sample property provider illustrates the `getPropertyValue` method, which returns a property value for the specified class, parent class, and property name. A CIM property is defined by its name and origin class. Two or more properties can have the same name, but the origin class uniquely identifies the property.

EXAMPLE 5–2 Implementing a Property Provider

```
...

public class SimplePropertyProvider implements PropertyProvider{
    public void initialize(CIMOMHandle cimom)
    throws CIMException {
    }

    public void cleanup()
    throws CIMException {
    }

    public CIMValue getPropertyValue(CIMObjectpath op, string originclass,
               string PropertyName){
            if (PropertyName.equals("A")
                return new CIMValue("ValueA")
            else
                  return new CIMValue("ValueB");
    }
    ...
}
```

The Method Provider Interface (`MethodProvider`)

The following table describes the method in the Method Provider interface.

TABLE 5–4 `MethodProvider` Interface Methods

Method	Description
CIMValue `invokeMethod(CIMObjectPath op, String methodName, Vector inParams, Vector outParams)`	The CIM Object Manager calls this method when `methodName` in the instance referred to by `op` is invoked.. *inParams* is a vector of `CIMValues` that are input parameters to the invoked method. *outParams* is a vector of `CIMValues` that are output parameters from the invoked method.

Example — Implementing a Method Provider

The code segment in Example 5–3 creates a Solaris provider class that routes requests to execute methods from the CIM Object Manager to one or more specialized providers. These specialized providers service requests for dynamic data for a particular type of Solaris object. For example, the `Solaris_Package` provider services requests to execute methods in the `Solaris_Package` class.

The method provider in this example implements a single method `invokeMethod` that calls the appropriate provider to perform one of following operations:

- Reboot a Solaris system
- Reboot or shut down a Solaris system
- Delete a Solaris serial port

EXAMPLE 5–3 Implementing a Method Provider

```
...
public class Solaris implements MethodProvider {
    public void initialize(CIMONHandle, ch) throws CIMException {
    }
    public void cleanup() throws CIMException {
    }
    public CIMValue invokeMethod(CIMObjectPath op, String methodName,
            Vector inParams, Vector outParams) throws CIMException {
        if (op.getObjectName().equalsIgnoreCase("solaris_computersystem")) {
            Solaris_ComputerSystem sp = new Solaris_ComputerSystem();
            if (methodName.equalsIgnoreCase("reboot")) {
                return new CIMValue (sp.Reboot());
            }
        }
        if (op.getObjectName().equalsIgnoreCase("solaris_operatingsystem")) {
            Solaris_OperatingSystem sos = new Solaris_OperatingSystem();
            if (methodName.equalsIgnoreCase("reboot")) {
                return new CIMValue (sos.Reboot());
            }
            if (methodName.equalsIgnoreCase("shutdown")) {
                return new CIMValue (sos.Shutdown());
            }
        }
```

EXAMPLE 5–3 Implementing a Method Provider *(Continued)*

```
        if (op.getObjectName().equalsIgnoreCase("solaris_serialport")) {
            Solaris_SerialPort ser = new Solaris_SerialPort();
            if (methodName.equalsIgnoreCase("disableportservice")) {
                return new CIMValue (ser.DeletePort(op));
            }
        }
        return null;
    }
}
...
```

The Associator Provider Interface (`AssociatorProvider`)

The following table describes the methods in the `AssociatorProvider` interface. For detailed information about the arguments these methods take, see "The Association Methods" on page 92.

TABLE 5–5 `AssociatorProvider` Interface Methods

Method	Description
Vector `associators(CIMObjectPath assocName, CIMObjectPath objectName, String role, String resultRole, boolean includeQualifiers, boolean includeClassOrigin, String[] propertyList)`	Returns a vector of CIM instances that are associated to the instance specified by `objectName`.
Vector `associatorNames(CIMObjectPath assocName, CIMObjectPath objectName, String role, String resultRole)`	Returns a vector of the names of CIM instances that are associated to the CIM instance specified by `objectName`.
Vector `references(CIMObjectPath assocName, CIMObjectPath objectName, String role, boolean includeQualifiers, boolean includeClassOrigin, String[] propertyList)`	Returns a vector of associations in which the CIM instance specified by `objectName` participates.
Vector `referenceNames(CIMObjectPath assocName, CIMObjectPath objectName, String role)`	Returns a vector of the names of associations in which the CIM instance specified by `objectName` participates.

Example — Implementing an Association Provider

A complete association provider must implement all `AssociatorProvider` methods. For brevity, the code segment in the following example implements only the `associators` method. The CIM Object Manager passes values for `assocName`, `objectName`, `role`, `resultRole`, `includeQualifiers`, `includeClassOrigin`, and `propertyList` to the association provider.

Example 5–4 prints the name of a CIM association class and the CIM class or instance whose associated objects are to be returned. This provider handles instances of `example_teacher` and `example_student` classes.

EXAMPLE 5–4 Implementing an Association Provider

```
public Vector associators(CIMObjectPath assocName,
        CIMObjectPath objectName, String role,
                String resultRole, boolean includeQualifiers,
                boolean includeClassOrigin, String propertyList[]) throws CIMException {
    System.out.println("Associators "+assocName+" "+objectName);
       if (objectName.getObjectName()equalsIgnoreCase("example_teacher")) {
             Vector v = new Vector();
             if ((role != null)  &&
                  (!role.equalsIgnoreCase("teaches"))) {
                   // Teachers only play the teaches role.
                   return v;
             }
          // Get the associators of a teacher
          CIMProperty nameProp = (CIMProperty)objectName.getKeys().elementAt(0);
          String name = (String)nameProp.getValue().getValue();
          // Get the student class
          CIMObjectPath tempOp = new CIMObjectPath("example_student");
          tempOp.setNameSpace(assocName.getNameSpace());
          CIMClass cc = cimom.getClass(tempOp, false);
    // Test the instance name passed by objectName
    // and return the associated instances of the student class.
       if(name.equals("teacher1")) {
             // Get students for teacher1
             CIMInstance ci = cc.newInstance();
             ci.setProperty("name", new CIMValue("student1"));
             v.addElement(ci.filterProperties(propertyList,
                       includeQualifiers, includeClassOrigin));
             ci = cc.newInstance();
             ci.setProperty("name", new CIMValue("student2"));
             v.addElement(ci.filterProperties(propertyList,
                       includeQualifiers, includeClassOrigin));
             return v;
          }
}
```

Writing a Native Provider

Providers get and set information on managed devices. A native provider is a machine-specific program written to run on a managed device. For example, a provider that accesses data on a Solaris system will most likely include C functions to query the Solaris system. Two common reasons for writing a native provider are:

- Efficiency – You may want to implement a small portion of time-critical code in a lower-level programming language, such as assembly, and then have your Java application call these functions.

- Need to access platform-specific features – The standard Java class library may not support the platform-dependent features needed by your application.
- Legacy code - Often, you have legacy code written in some programming language other than Java and want to continue to use the code with a Java provider.

The Java Native Interface is the native programming interface for Java that is part of the JDK. By writing programs using the JNI, you ensure that your code is completely portable across all platforms. The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly.

For more information on writing and integrating Java programs with native methods, visit the Java web site at
http://www.javasoft.com/docs/books/tutorial/native1.1/index.html.

Installing a Provider

After you write a Provider, you must specify the location of the provider class files and any shared library files and then stop and restart the CIM Object Manager.

▼ How to Install a Provider

1. **Specify the location of shared library files in one of the following ways:**
   - Set the `LD_LIBRARY_PATH` environment variable to the location of the shared library files. For example, using the C shell:

     Note – If you set the `LD_LIBRARY_PATH` environment variable in a shell, you must stop and restart the CIM Object Manager in the same shell for this new value to be recognized.

     ```
     % setenv LD_LIBRARY_PATH /wbem/provider/
     ```
     For example, using the Bourne shell:
     ```
     % LD_LIBRARY_PATH = /wbem/provider/
     ```
   - Copy the shared library files to one of the directories specified by the `LD_LIBRARY_PATH` environment variable. The installation sets this environment variable to `/usr/sadm/lib/wbem` and `/usr/snadm/lib`. For example:
     ```
     % cp libnative.so /usr/sadm/lib/wbem% cp native.c /usr/sadm/lib/wbem
     ```
2. **Move the provider class files to `/usr/sadm/lib/wbem`.**

Move the provider class files to the same path as the package in which they are defined. For example, if the provider is packaged as `com.sun.providers.myprovider.*`, move the provider class files to `/usr/sadm/lib/wbem/com/sun/providers/myprovider/`.

3. **Set the Solaris Provider `CLASSPATH` variable to the directory that contains the provider class files as described in "To Set the Provider CLASSPATH" on page 123.**

4. **Stop the CIM Object Manager by typing the following command:**

Note – If you set the `LD_LIBRARY_PATH` environment variable in a shell, you must stop and restart the CIM Object Manager in the same shell for this new value to be recognized.

```
# /etc/init.d/init.wbem -stop
```

5. **Restart the CIM Object Manager by typing the following command:**

```
# /etc/init.d/init.wbem -start
```

Setting the Solaris Provider `CLASSPATH`

To set the Solaris provider's `CLASSPATH`, use the client APIs to create an instance of the `Solaris_ProviderPath` class and set its `pathurl` property to the location of your provider class files. The `Solaris_ProviderPath` class is stored in the `\root\system` namespace.

You can also set the provider `CLASSPATH` to the location of your provider class files. You can set the class path to the `jar` file or to any directory that contains the classes. Use the standard URL format that Java uses for setting the `CLASSPATH`.

Provider `CLASSPATH`	Syntax
Absolute path to directory	`file:///a/b/c/`
Relative path to directory from which the CIM Object Manager was started (`/`). .	`file://a/b/c`

▼ To Set the Provider CLASSPATH

1. **Create an instance of the `Solaris_ProviderPath` class. For example:**

```
/* Create a namespace object initialized with root\system
(name of namespace) on the local host. */
CIMNameSpace cns = new CIMNameSpace("", "root\system");
```

```
// Connect to the root\system namespace as root.
cc = new CIMClient(cns, "root", "root_password");

// Get the Solaris_ProviderPath class
cimclass = cc.getClass(new CIMObjectPath("Solaris_ProviderPath");

// Create a new instance of Solaris_ProviderPath.
class ci = cimclass.newInstance();
```

2. **Set the `pathurl` property to the location of your provider class files. For example:**

```
...
/* Set the provider CLASSPATH to //com/mycomp/myproviders/.*/
ci.setProperty("pathurl", new CIMValue(new String("//com/mycomp/myproviders/"));
...
```

3. **Update the instance. For example:**

```
// Pass the updated instance to the CIM Object Manager
cc.setInstance(new CIMObjectPath(), ci);
```

Registering a Provider

Providers register with the CIM Object Manager to publish information about the data and operations they support and their physical implementation. The CIM Object Manager uses this information to load and initialize the provider and to determine the right provider for a particular client request. All types of providers follow the same procedure for registration.

▼ How To Register a Provider

1. **Create a MOF file defining a CIM class.**

2. **Assign the *provider* qualifier to the class. Assign a provider name to the *provider* qualifier.**

   The provider name identifies the Java class to serve as the provider for this class. You must specify the complete class name. For example:

```
[Provider("com.kailee.wbem.providers.provider_name")]
Class_name {
...
};
```

Note – We recommend following Java class and package naming conventions for providers so that provider names are unique. The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently `com`, `edu`, `gov`, `mil`, `net`, `org`, or one of the English two-letter codes identifying countries as specified in ISO Standards 3166, 1981.

Subsequent components of the package name vary according to an organizations own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names, for example, `com.mycompany.wbem.myprovider`.

3. **Compile the MOF file. For example:**

```
% mofcomp class_name
```

For more information on using the Managed Object Format (MOF) Compiler to compile a MOF file, see the *Solaris WBEM Services Administrator's Guide*.

Changing a MOF File

If you change a class definition in a MOF file that was previously compiled, you must delete the class from the CIM Object Manager Repository before recompiling the MOF file. Otherwise, you will get an error that the class already exists and the new information will not propagate to the CIM Object Manager. You can use the CIM WorkShop to delete a class as described in "Deleting Classes and Their Attributes" on page 38.

Example — Registering a Provider

The following example shows the MOF file that declares to the CIM Object Manager the `Ex_SimpleInstanceProvider` class that is served by the `SimpleInstanceProvider` (shown in Example 5–1). Provider and class names in a valid MOF file follow these rules:

- The class name must be a valid CIM Schema name, which means that it must have a prefix of characters followed by an underscore. For example: `green_apples` and `red_apples` are valid CIM schema names. The class name `apples` is not a valid CIM Schema name.
- The class name must match the class name specified in the provider for this MOF file. The MOF file in Example 5–5 declares the `Ex_SimpleInstanceProvider` class. For example:

  `class Ex_SimpleInstanceProvider`
  The `enumInstances` method in the provider in Example 5–1 specifies the same class name. For example:

```
public Vector enumInstances(CIMObjectPath op, boolean deep, CIMClass cc)
    throws CIMException {
        if (op.getObjectName().equalsIgnoreCase("Ex_SimpleInstanceProvider"))
```

- The provider name specified in the MOF file must match the name of the provider class file. The MOF file in Example 5–5 specifies the `SimpleInstanceProvider` as the provider for the `Ex_SimpleInstanceProvider` class. For example:

```
[Provider("SimpleInstanceProvider")]
class Ex_SimpleInstanceProvider
```

EXAMPLE 5–5 `SimpleInstanceProvider` MOF File

```
// ===================================================================
// Title:       SimpleInstanceProvider
// Filename:    SimpleInstanceProvider.mof
// Description:
// ===================================================================

// ===================================================================
// Pragmas
// ===================================================================
#pragma Locale ("en-US")

// ===================================================================
//   SimpleInstanceProvider
// ===================================================================
[Provider("SimpleInstanceProvider")]
class Ex_SimpleInstanceProvider
{
   // Properties
      [Key, Description("First Name of the User")]
   string First;
      [Description("Last Name of the User")]
   string Last;
};
```

Modifying a Provider

You can make changes to a Provider class while the CIM Object Manager and provider are running. But you must stop and restart the CIM Object Manager to make the changes take effect.

▼ How To Modify a Provider

1. **Edit the provider source file.**
2. **Compile the provider source file. For example:**

```
% javac MyProvider.java
```

3. **Become root on your system by typing the following command at the system prompt:**

```
% su
```

4. **Type the root password when you are prompted.**

5. **Change directories to the location of the `init.wbem` command by typing the following command:**

```
# cd /etc/init.d/
```

6. **Stop the CIM Object Manager by typing the following command:**

```
# init.wbem -stop
```

7. **Restart the CIM Object Manager by typing the following command:**

```
# init.wbem -start
```

Handling WBEM Query Language Queries

WBEM clients use the `execQuery` method in the `CIMClient` class to search for instances that match a set of search criteria. The CIM Object Manager handles client queries for CIM data stored in the CIM Object Manager Repository and it passes to providers queries for CIM data that is served by a particular provider.

All instance providers must imlement the `execQuery` interface in the `com.sun.wbem.provider20` package to handle client queries for the dynamic data they provide. Providers can use the classes and methods in the `com.sun.wbem.query` package to filter WBEM Query Language (WQL) query strings. Providers with access to an entity that handles indexing can pass the query string to that entity for parsing.

Using the Query APIs to Parse Query Strings

The classes and methods in the `com.sun.wbem.query` package represent a WBEM Query Language parser and the WQL string to be parsed. The package includes classes that represent clauses within the query string and methods for manipulating the strings within those clauses.

Currently, the only type of WQL expression that can be parsed is the SELECT expression. A SELECT expression contains the following parts:

- SELECT statement
- FROM clause
- WHERE clause

The WBEM Query Language Expression

The following figure shows the WBEM classes that represent the clauses in a WQL expression.

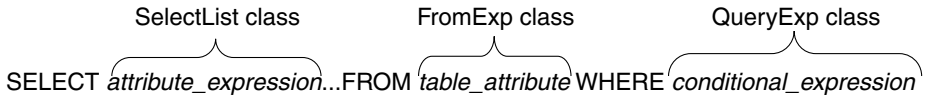

FIGURE 5–1 WBEM Classes that Represent the WBEM Query Language Expression

WBEM Query Lanuage	WBEM Query Class
`SELECT` *attribute_expression*	`SelectList`
`FROM` *table_attribute*	`FromExp`
`WHERE` *conditional_expression*	`QueryExp`

WQL has been adapted to query data that is stored using the CIM data model. In the CIM model, information about objects is stored in CIM classes and CIM instances. CIM instances can contain properties, which have a name, data type, and value. WQL maps the CIM object model to SQL tables, as shown in the following table:

SQL	WQL
Table	CIM class
Row	CIM instance
Column	CIM property

In CIM, a WQL expression could be expressed in the following form:

`SELECT` *CIM property* **...`FROM`** *CIM class* **`WHERE`** *propertyA = 40*

A more realistic example of a WQL expression follows:

```
SELECT * FROM Solaris_FileSystem WHERE (Name="home" OR Name="files") AND AvailableSpace > 2000000
```

The SELECT Statement

The `SelectExp` class represents the SELECT statement.

The SELECT statement is the SQL statement for retrieving information, with a few restrictions and extensions specific to WQL. Although the SQL SELECT statement is typically used in the database environment to retrieve particular columns from tables, the WQL SELECT statement is used to retrieve instances of a single class. WQL does not support queries across multiple classes.

The SELECT expression identifies the search list. The SELECT statement can take one of the following forms:

SELECT Statement	Selects
`SELECT *`	All instances of the specified class and any of its subclasses.
`SELECT attr_exp, attr_exp...attr_exp`	Only instances of the specified class and any of its subclasses that contain the specifies identifiers.

The FROM Clause

The FROM clause is represented by the abstract class, `fromExp`. Currently `NonJoinExp` is the only direct subclass of `fromExp`. The `NonJoinExp` represents FROM clauses with only one table (CIM class) to which the select operation should be applied.

The FROM clause identifies the class in which to search for instances that match the query string. In SQL terms, the FROM clause identifies a qualified attribute expression, which is the name of a class to search. A qualified attribute expression identifies the table and class. We currently support only non-join expressions, which means that a valid WQL FROM clause includes only a single class.

The WHERE Clause

The `QueryExp` class is an abstract class whose subclasses represent conditional expressions which return a boolean value when a particular `CIMInstance` is applied to them.

The WHERE clause narrows the scope of a query. The WHERE clause contains a conditional expression, which can contain a property or key word, an operator, and a constant. All WHERE clauses must specify one of the predefined WQL operators.

The basic syntax for a WHERE clause appended to a SELECT statement follows:

`SELECT` *CIM instance* **`FROM`** *CIM class***`WHERE`** *conditional_expression*

The conditional expression in a WHERE clause takes the following form:

property operator constant

The following subclasses of the `QueryExp` class manipulate particular types of conditional expressions in the WHERE clause:

- `AndQueryExp`
- `BinaryRelQueryExp`
- `NotQueryExp`
- `OrQueryExp`

The conditional expression in the WHERE clause is represented by the `QueryExp` class. A conditional expression is represented by a tree structure. For example, the conditional expression (a=2 and b=3 or c=4) is represented by the tree structure shown in the following figure:

The `QueryExp` class returns only the top level of the query expression tree. In the above example, an OR QueryExp. The provider can then use methods within that class to get branches down the query expression tree.

Using the Canonize Methods

The following methods are useful for providers that pass the WQL query string to another entity that parses the string:

- `canonizeDOC` - Canonizes the expression into a Disjunction of Conjunctions form. (OR of ANDed comparison expressions). This enables handling of the expression as a List of Lists rather than a tree form, enabling ease of evaluation. For example: (x > 5 and y > 6) or (y > 6 and z=7)
- `canonizeCOD` - Canonizes the expression into a Conjunction of Disjunctions form. (AND of ORed comparison expressions). This enables handling of the expression as a List of Lists rather than a tree form, enabling ease of evaluation. For example: (x > 5 or y > 6) and (y > 6 or z=7)

Writing a Provider that Parses WQL Query Strings

The general procedure for writing a provider that parses WQL queries using the Query APIs follows.

▼ How to Write A Provider that Parses WQL Query Strings

1. Initialize the WQL parser, for example:

```
/* Read query string passed to execQuery from the CIM Object
   Manager into an input data stream. */
```

```
ByteArrayInputStream in = new ByteArrayInputStream(query.getBytes());

/* Initialize the parser with the input data stream. */
WQLParser parser = new WQLParser(in);
```

2. **Create a vector to store the result of the query. For example:**

```
Vector result = new Vector();
```

3. **Get the select expression from the query. For example:**

```
/* querySpecification returns the WQL expression from the parser.
  (SelectExp)parser casts the WQL expression to a select expression. */
SelectExp q = (SelectExp)parser.querySpecification();
```

4. **Get the select list from the select expression. For example:**

```
/* Use the SelectList method in the SelectExp class
   to return the select list. The select list is the list
   of attributes, or CIM properties. */
SelectList attrs = q.getSelectList();
```

5. **Get the From clause. For example:**

```
/* Use the getFromClause method in the SelectExp class
   to return the From clause. Cast the From clause to a
   Non Join Expression, a table that represents a single
   CIM class. */
NonJoinExp from = (NonJoinExp)q.getFromClause();
```

6. **Use the `enumInstances` method to return a deep enumeration of the class. For example:**

```
/* Returns all instances, including inherited and local properties,
   belonging to the specified class (cc). */
Vector v = new Vector();
v = enumInstances(op, true, cc, true);
...
```

7. **Iterate through the instances in the enumeration, matching the query expression and select list to each instance. For example:**

```
/* Test whether the query expression in the WHERE
```

```
        clause matches the CIM instance. Apply the select
       list to the CIM instance and add any instance that
       matches the select list (list of CIM properties)
       to the result. */
    for (int i = 0; i < v.size(); i++) {
        if ((where == null) || // If there is a WHERE clause
            (where.apply((CIMInstance)v.elementAt(i)) == true)) {
              result.addElement(attrs.apply((CIMInstance)v.elementAt(i)));
     ...
```

8. Return the query result. For example:

```
return result;
```

Example — Implementing the `execQuery` Method

The sample program in Example 5–6 uses the Query APIs to parse the WQL string passed to it by the `execQuery`method. This program parses the Select Expression in the query string, does a deep enumeration of the class, and iterates through the instances in the enumeration, matching the query expression and select list to each instance. Finally, the program returns a vector containing the enumeration of the instances that match the query string.

EXAMPLE 5–6 Provider that Implements the execQuery Method

```
/*
     * The execQuery method will support only limited queries
     * based upon partial key matching. An empty Vector is
     * returned if no entries are selected by the query.
     *
     * @param    op            The CIM object path of the CIM instance to be returned
     * @param    query    The CIM query expression
     * @param    ql            The CIM query language indicator
     * @param    cc            The CIM class reference
     *
     * @return    A vector of CIM object instances
     *
     * @version    1.19    01/26/00
     * @author       Sun Microsystems, Inc.
     */
    public Vector execQuery(CIMObjectPath op,
                String query,
                int ql,
                CIMClass cc)
        throws CIMException {

    ByteArrayInputStream in = new ByteArrayInputStream(query.getBytes());
    WQLParser parser = new WQLParser(in);
    Vector result = new Vector();
    try {
        SelectExp q = (SelectExp)parser.querySpecification();
        SelectList attrs = q.getSelectList();
```

EXAMPLE 5–6 Provider that Implements the execQuery Method *(Continued)*

```
        NonJoinExp from = (NonJoinExp)q.getFromClause();
        QueryExp where = q.getWhereClause();

        Vector v = new Vector();
        v = enumInstances(op, false, cc, true);

        // filtering the instances
        for (int i = 0; i < v.size(); i++) {
            if ((where == null) || (where.apply((CIMInstance)v.elementAt(i)) == true)) {
                result.addElement(attrs.apply((CIMInstance)v.elementAt(i)));
             }
        }
    } catch (Exception e) {
        throw new CIMException(CIMException.CIM_ERR_FAILED, e.toString());
    }
    return result;
    } // execQuery
}
```

CHAPTER 6

Handling CIM Events

This chapter describes the CIM event model and explains how providers generate CIM events and how applications subscribe to be notified of the occurrence of CIM events.

- The CIM Event Model
- Creating a Subscription
- Generating an Event Indication

The CIM Event Model

An event is an occurrence, something that happens. In programming terms, an event is an occurrence in a computer system to which an application might need to respond, for example, opening a dialog box in response to a mouse click on a GUI. A CIM event is a change in an occurrence of interest in a managed environment.

The CIM event model, a framework for handling CIM events, conforms to the Common Information Model (CIM) Indications Specification, published by the Desktop Management Task Force (DMTF). The CIM event model distinguishes between an event and the notification of that event (an indication). In CIM, events are not published; indications are published.

CIM events can be classified as intrinsic or extrinsic. An intrinsic event is a built-in CIM event that occurs in response to changes to data in which a namespace, class, or class instance is created, modified, or deleted. An extrinsic event is a user-defined event that is not already described by an intrinsic events.

Currently, only intrinsic events for creating, modifying, and deleting CIM instances are handled. The following classes are used to report intrinsic events:

- `CIM_InstCreation` — Indicates that a new instance was created.

- `CIM_InstDeletion` — Indicates that an existing instance was deleted.
- `CIM_InstModification` — Indicates that an instance was modified.

How Indications of Events are Generated

By default, the CIM Object Manager polls for indications of intrinsic events at regular intervals. Administrators can change the event polling interval and the default polling behavior of the CIM Object Manager by editing the properties in the `cimom.properties` file. For instructions on editing the `cimom.properties` file, see the Solaris WBEM Services Administrator's Guide.

The CIM Object Manager polls by enumerating instances, repeating the enumeration at the specified polling interval, and then finding any new, changed, or deleted instances between the two sets of enumerated instances. If a provider generates indications for events, the CIM Object Manager will not poll the provider. If possible, providers should generate indications for the intrinsic events that occur in the dynamic instances of the classes they support. This prevents the performance cost that results from the CIM Object Manager polling for the event.

The CIM Object Manager Repository generates an indication when a static instance is created, modified, or deleted.

How Subscriptions are Created

A client application can subscribe to be notified of CIM events. A subscription is a declaration of interest in one or more streams of indications. Currently, providers cannot subscribe for event indications.

An application that subscribes for indications of CIM events describes:

- Which events in which it is interested.
- What action the CIM Object Manager should take when the events occur.

The occurrence of an event is represented as an instance of one of the subclasses of the `CIM_Indication` class. An indication is generated only when a client subscribes for the event. If no provider delivers the event for which a client subscribes, the subscription will fail.

Creating a Subscription

Creating a subscription involves adding the `CIMListener` interface and creating instances of the following classes:

- `CIM_IndicationFilter` — Defines the criteria for generating an indication and what data should be returned in the indication.
- `CIM_IndicationHandler` — Describes how an indication is to be processed and delivered (handled). This may define a destination and protocol for delivering indications.
- `CIM_IndicationSubscription` — An association that binds a particular event filter with a particular event handler.

An application can create one or more event filters with one or more event handlers. Indications of events are not delivered until an application creates a subscription for the events.

Adding a CIM Listener

A client application must add the `CIMListener` interface to register for indications of CIM events. The CIM Object Manager generates indications for CIM events that are specified by the event filter when a client subscription is created.

The `CIMListener` interface must implement the `indicationOccured` method which takes the argument, `CIMEvent`, the CIM event returned by the `CIMListener`.

EXAMPLE 6–1 Adding a CIM Listener

```
// Connect to the CIM Object Manager
cc = new CIMClient();

// Register the CIM Listener
cc.addCIMListener(
new CIMListener() {
    public void indicationOccured(CIMEvent e) {
    }
});
```

Creating an Event Filter

Event filters describe the types of events to be delivered and the conditions under which they are delivered. An application creates an event filter by creating an instance of the `CIM_IndicationFilter` class and defining values for its properties. Event filters belong to a namespace. Each event filter works only on events that belong to the namespace to which the filter also belongs.

The CIM_IndicationFilter class has string properties that an application can set to uniquely identify the filter, specify a query string, and the query language to parse the query string, as shown in the following table. Currently, only the WBEM Query Language is supported.

TABLE 6–1 Properties in the CIM_IndicationFilter Class

Property	Description	Required/Optional
SystemCreationClassName	The name of the system on which the creation class for the filter resides or to which it applies.	Optional. The default for this key property is the CIM_System.CreationClassName.
SystemName	The name of the system on which the filter resides or to which it applies.	Optional. The default for this key property is the name of the system on which the CIM Object Manager is running.
CreationClassName	The name of the class or subclass used to create the filter.	Optional. The CIM Object Manager assigns CIM_IndicationFilter as the default for this key property.
Name	The unique name of the filter.	Required. The client application must assign a unique name.
SourceNamespace	The path to a local namespace where the CIM indications originate.	Optional. The default is \root\cimv2.
Query	A query expression that defines the conditions under which indications will be generated. Currently, only Level 1 WBEM Query Language expressions are supported. To learn how to construct WQL query expressions, see "Querying" on page 87.	Required.
QueryLanguage	The language in which the query is expressed.	Required. The default is WQL (WBEM Query Language).

▼ How to Create an Event Filter

1. Create an instance of the CIM_IndicationFilter class. For example.

```
CIMClass cimfilter = cc.getClass
        (new CIMObjectPath("CIM_IndicationFilter"),
```

```
          true, true, true, null);CIMInstance ci = cimfilter.newInstance();
```

2. **Specify the name of the event filter. For example.**

```
Name = "filter_all_new_solarisdiskdrives"
```

3. **Create a WQL string to identify event indications to be returned. For example.**

```
String filterString = "SELECT *
        FROM CIM_InstCreation WHERE sourceInstance
        is ISA Solaris_DiskDrive";
```

4. **Set property values in the `cimfilter` instance to identify the name of the filter, the filter string to select CIM events, and the query language to parse the query string.**

   Currently, only the WBEM Query Language can be used to part query strings. For example.

```
ci.setProperty("Name", new
        CIMValue("filter_all_new_solarisdiskdrives"));
ci.setProperty("Query", new CIMValue(filterString));
ci.setProperty("QueryLanguage", new CIMValue("WQL");)
```

5. **Create an instance from the `cimfilter` instance, called `filter`, and store it in the CIM Object Manager Repository.**

```
CIMObjectPath filter = cc.createInstance(new CIMObjectPath(), ci);
```

Creating an Event Handler

An event handler is an instance of a `CIM_IndicationHandler` class. The CIM Event MOF defines a `CIM_IndicationHandlerXMLHTTP` class for describing the destination for indications to be delivered to client applications using the HTTP protocol. Event delivery to HTTP clients is not supported because HTTP delivery for events is not defined yet.

The Solaris Event MOF extends the `CIM_IndicationHandler` class by creating the `Solaris_RMIDelivery` class to handle delivery of indications of CIM events to client applications using the RMI protocol. RMI clients must instantiate the `Solaris_RMIDelivery` class to set up an RMI delivery location.

An application sets the properties in the `CIM_IndicationHandler` class to uniquely name the handler and identify the UID of its owner.

TABLE 6–2 Properties in the `CIM_IndicationHandler` Class

Property	Description	Required/Optional
SystemCreationClassName	The name of the system on which the creation class for the handler resides or to which it applies.	Optional. The default for this key property is the name of the creation class for the `CIM_System` class.
SystemName	The name of the system on which the handler resides or to which it applies.	Optional. The default value for this key property is the name of the system on which the CIM Object Manager is running.
CreationClassName	The class or subclass used to create the handler.	Optional. The CIM Object Manager assigns `CIM_IndicationFilter` as the default for this key property.
Name	The unique name of the handler.	Required. The client application must assign a unique name.
Owner	The name of the entity that created or maintains this handler. Provider can check this value to determine whether or not to authorize a handler to receive an indication.	Optional. The default value is the Solaris user name of the user creating the instance.

The following example shows the code for creating a CIM event handler.

EXAMPLE 6–2 Creating a CIM Event Handler

```
// Create an instance of the Solaris_RMIDelivery class.
CIMClass rmidelivery = cc.getClass(new CIMObjectPath
        ("Solaris_RMIDelivery"), false, true, true, null);

CIMInstance ci = rmidelivery.newInstance();

//Create a new instance (delivery) from
//the rmidelivery instance.
CIMObjectPath delivery = cc.createInstance(new CIMObjectPath(), ci);
```

Binding an Event Filter to an Event Handler

An application binds an event filter to an event handler by creating an instance of the `CIM_IndicationSubscription` class. When a `CIM_IndicationSubscription` is created, indications for the events specified by the event filter are delivered.

The following example creates a subscription (`filterdelivery`) and defines the `filter` property to the `filter` object created in "How to Create an Event Filter" on page 138, and defines the `handler` property to the `delivery` object created in Example 6–2.

EXAMPLE 6–3 Binding an Event Filter to an Event Handler

```
CIMClass filterdelivery = cc.getClass(new
        CIMObjectPath("CIM_IndicationSubscription"),
        true, true, true, null);
ci = filterdelivery.newInstance():

//Create a property called filter that refers to the filter instance.
ci.setProperty("filter", new CIMValue(filter));

//Create a property called handler that refers to the delivery instance.
ci.setProperty("handler", new CIMValue(delivery));
```

Generating an Event Indication

Generating an indication for a CIM event involves:

- Using the methods in the `EventProvider` interface to detect when to start and stop delivering indications of CIM event.
- Creating an instance of one or more subclasses of the `CIM_Indication` class to store information about the CIM event that occurred.
- Using the `deliverEvent` method in the `ProviderCIMOMHandle` interface to deliver indications to the CIM Object Manager.

Methods in the `EventProvider` Interface

An event provider must implement the `EventProvider` interface. This interface contains methods that the CIM Object Manager uses to notify the provider when a client has subscribed for indications of CIM events, and when a client has cancelled the subscription for CIM events. These methods also allow the provider to indicate whether or not the CIM Object Manager should poll for some event indications and whether or not the provider should authorize the return of an indication to a handler.

The following table lists the methods in the `EventProvider` interface that must be executed by an event provider.

TABLE 6–3 Methods in the `EventProvider` Interface

Method	Description
`activateFilter`	When a client creates a subscription, the CIM Object Manager calls this method to ask the provider to check for CIM events.
`authorizeFilter`	When a client creates a subscription, the CIM Object Manager calls this method to test if the specified filter expression is allowed.
`deActivateFilter`	When a client removes a subscription, the CIM Object Manager calls this method to ask the provider to deactivate the specified event filter.
`mustPoll`	When a client creates a subscription, the CIM Object Manager calls this method to test if the specified filter expression is allowed by the provider, and if it must be polled.

The CIM Object Manager passes values for the following arguments to all methods:

- filter — `SelectExp` that specifies the CIM events for which indications must be generated.
- eventType — `String` that specifies the type of CIM event, which can also be extracted from the FROM clause of the select expression.
- classPath — `CIMObjectPath` that specifies the name of the class for which the event is required.

In addition, the `activateFilter` method takes the boolean `firstActivation`, indicating that this is the first filter for this event type. The `deActivateFilter` method takes the boolean `lastActivation`, indicating that this is the last filter for this event type.

Creating and Delivering Indications

When a client application subscribes for indications of CIM events by creating an instance of the `CIM_IndicationSubscription` class, the CIM Object Manager forwards the request to the appropriate provider. If the provider implements the `EventProvider` interface, the CIM Object Manager notifies the provider when to start sending indications for the specified events by calling the provider's `activateFilter` method, and it notifies the provider when to stop sending indications for the specified events by calling the provider's `deActivateFilter` method.

The provider responds to the CIM Object Manager's requests by creating and delivering an indication each time the provider creates, modifies, or deletes an instance. A provider typically defines a flag variable that is set when the CIM Object Manager calls the `activateFilter` method and cleared when the CIM Object Manager calls the `deActivateFilter` method. Then in each method that creates, modifies, or deletes an instance, the provider checks the status of the activate filter flag. If the flag is set, the provider creates an indication containing the created CIM instance object and uses the `deliverEvent` method to return the indication to the CIM Object Manager. If the flag is not set, the provider does not create and deliver an indication of the event.

Authorizations

A provider that handles sensitive data can check authorizations for requests for indications. The provider must implement the `Authorizable` interface to indicate that it handles authorization checking. The provider also implements the `authorizeFilter` method. The CIM Object Manager calls this method to test if the owner (UID) of an event handler is authorized to receive the indications that result from evaluating a filter expression. The UID for the owner of the event destination (event handler) can be different than the owner of the client application requesting the filter activation.

CIM Indication Classes

Providers generate indications of CIM events by creating instances of subclasses of the `CIM_Indication` class.

The following table lists the intrinsic CIM events that a provider should generate.

TABLE 6–4 CIM Events Indication Classes

Event Class	Description
`CIM_InstCreation`	Notifies when a new instance is created.
`CIM_InstDeletion`	Notifies when an existing instance is deleted.
`CIM_InstModification`	Notifies when an instance is modified. The indication must include a copy of the previous instance whose change generated the indication.

▼ How to Generate an Event Indication

1. **Implement the `EventProvider` interface. For example:**

```
public class sampleEventProvider implements
        InstanceProvider EventProvider{

    // Reference for provider to contact the CIM Object Manager
    private ProviderCIMOMHandle cimom;
   }
```

2. **Execute each of the methods listed in Table 6–3 for each instance indication that the provider handles.**

3. **Create an indication listed in Table 6–4 for each create, modify, and delete instance event type. For example, in the `createInstance` method:**

```
public CIMObjectPath createInstance(CIMObjectPath op,
        CIMInstance ci)
    throws CIMException {
        CIMObjectpath newop = ip.createInstance(op, ci);
        CIMInstance indication = new CIMInstance();
        indication.setClassName("CIM_InstCreation");
        CIMProperty cp = new CIMProperty();
        cp.setName("SourceInstance");
        cp.setValue(new CIMValue(ci));
        Vector v = new Vector();
        v.addElement(cp);
        indication.setProperties(v);
        ...
    }
```

4. **Deliver the event indication to the CIM Object Manager. For example:**

```
cimom.deliverEvent(op.getNameSpace(), indication);
        return newop;
```

CHAPTER 7

Using Sun WBEM SDK Examples

This chapter describes the sample programs provided in the Sun WBEM SDK and includes the following topics:

- About Example Programs
- Using the Applet
- Using the Client Examples
- Using the Provider Examples

About Example Programs

The Sun WBEM SDK provides example Java programs, which are installed in `/usr/demo/wbem`. You can use the source code as a basis for developing your own programs. Three types of example programs are provided:

- Applet – Programs that you can run in a Java-enabled Web Browser or run with the Java Development Kit (JDK) Appletviewer.
- Client programs – Programs that use the client and CIM application programming interfaces (APIs) to request WBEM operations from the CIM Object Manager
- Provider programs – Programs that communicate with managed objects to access data

Using the Applet

The `GetPackageInfoAp` is a Java Applet that you can use to list the Solaris software packages that are installed on a system running Solaris WBEM Services. You can select a package and display the detailed information about that package. You can use this applet to connect to a CIM Object Manager running on your local system or a remote system.

To run the applet you need one of the following:

- Java Development Kit (JDK) 1.2 Appletviewer
- Java-enabled Web browser that uses JRE 1.2.2 or has the Java Plug-in 1.2.2 software enabled

For detailed information on running the applet, see `/usr/demo/wbem/README`.

Using Client Examples

The client examples use the client APIs to create, delete, and list classes, instances, and namespaces. The five types of client programs are:

- Enumeration – Enumerates classes and instances. It does deep and shallow enumeration on a class that is passed from the command line.
- Logging – Writes and reads log records
- Miscellaneous – Deletes classes and instances.
- Namespace – Creates and deletes namespaces.
- System Information – Displays Solaris process information for the system and network you select.

Client Example Files

The following table describes the example client program files and lists the commands and arguments to run each example.

TABLE 7–1 Client Example Files

Example File Name	Description	Command to Run
`CreateNameSpace`	Connects to the CIM Object Manager as the specified user and creates a namespace on the specified host. You must type the root user name and password.	`java CreateNameSpace` *host parentNS childNS username password*
`DeleteNameSpace`	Deletes the specified namespace on the specified host. You must type the root user name and password.	`java DeleteNameSpace` *host parentNS childNS username password*
`ClientEnum`	Enumerates classes and instances in the specified class in the default namespace `root\cimv2` on the specified host.	`java ClientEnum` *host className*
`CreateLog`	Creates a log record on the specified host. You must type the root user name and password.	`java CreateLog` *host username password*
`ReadLog`	Reads a log record on the specified host. You must type the root user name and password.	`java ReadLog` *host username password*
`DeleteClass`	Deletes the specified class in the default namespace `root\cimv2` on the specified host. You must type the root user name and password.	`java DeleteClass` *host className username password*
`DeleteInstances`	Deletes instances of the specified class in the default namespace `root\cimv2` on the specified host. You must type the root user name and password.	`java DeleteInstances` *host className username password*

TABLE 7–1 Client Example Files *(Continued)*

Example File Name	Description	Command to Run
`CreateQualifierType`	Creates the specified qualifier type in the specified namespace on the specified host. You must type the root user name and password.	`java CreateQualifierType` *host parentNS username password qualifierTypeName*
`SystemInfo`	Displays Solaris processor and system information for the specified host in a window.	`java SystemInfo` *host*

Running the Client Examples

To run a client example program, type the command:

```
% java program_name
```

Most of the example programs take required arguments that have default values. For example, the `CreateNameSpace` example program takes five arguments:

- Host name
- Parent namespace
- Child namespace
- User name
- Password

Use the following syntax to specify default values for command line arguments.

Argument	Default Value	Syntax
Host name	`local host`	`.`
Parent namespace	`root\cimv2`	`" "`
Child namespace	`Null`	`" "`
User name	`GUEST`	`" "`
Password	`GUEST`	`" "`

The following example runs the `CreateNameSpace` example, which connects to the default `root\cimv2` namespace on the local host as user `root` with password `secret`.

```
% java CreateNameSpace . "" root secret
```

Using the Provider Examples

The example provider is a Java program that returns system properties and prints the string, “Hello World.” The provider calls native C methods to execute the code and return the values to the provider.

Provider Example Files

The following table describes the files that make up the example Provider program.

TABLE 7–2 Provider Example Files

File	Purpose
`NativeProvider`	Top level provider program that fulfills requests from the CIM Object Manager and routes them to the `Native_Example` provider. The `NativeProvider` program implements the `instanceProvider` and `methodProvider` APIs, and declares methods that enumerate instances and get an instance of the `Native_Example` class. It also declares a method that invokes a method to print the string "Hello World."
`Native_Example.mof`	Creates a class that registers the `NativeProvider` provider with the CIM Object Manager. The `Native_Example.mof` file identifies `NativeProvider` as the provider to service requests for dynamic data in the `Native_Example` class. This MOF file also declares the properties and methods to be implemented by the `NativeProvider`.
`Native_Example.java`	The `NativeProvider` program calls this provider to implement methods that enumerate instances and get an instance of the `Native_Example` class. The `Native_Example` provider uses the APIs to enumerate objects and create instances of objects. The `Native_Example` class declares native methods, which call C functions in the `native.c` file to get system-specific values, such as host name, serial number, release, machine, architecture, and manufacturer.

TABLE 7–2 Provider Example Files *(Continued)*

File	Purpose
`native.c`	C program that implements calls from the `Native_Example` Java provider in native C code.
`Native_Example.h`	Machine-generated header file for `Native_Example` class. Defines the correspondence between the Java native method names and the native C functions that execute those methods.
`libnative.so`	Binary native C code compiled from the `native.c` file.

Writing a Native Provider

For detailed information on writing and integrating Java programs with native methods, visit the Java Web page at `http://www.javasoft.com/docs/books/tutorial/native1.1/TOC.html`.

Setting Up the Provider Example

The example provider program, `NativeProvider`, enumerates instances and gets properties for instances of the `Native_Example` class. You can use the CIM WorkShop to view this class and its instances.

▼ How to Set Up the Provider Example

1. **Specify the location of shared library files in one of the following ways:**

   - Set the `LD_LIBRARY_PATH` environment variable to the location of the shared library files. For example, using the C shell:

     Note – If you set the `LD_LIBRARY_PATH` environment variable in a shell, you must stop and restart the CIM Object Manager in the same shell for this new value to be recognized.

     ```
     % setenv LD_LIBRARY_PATH /wbem/provider/
     ```

     For example, using the Bourne shell:

     ```
     % LD_LIBRARY_PATH = /wbem/provider/
     ```

   - Copy the shared library files to the directory specified by the `LD_LIBRARY_PATH` environment variable. Installation sets this environment variable to `/usr/sadm/lib/wbem`. For example:

```
% cp libnative.so /usr/sadm/lib/wbem
% cp native.c /usr/sadm/lib/wbem
% cp Native_Example.h /usr/sadm/lib/wbem
```

2. **Move the provider class files to `/usr/sadm/lib/wbem`.**

   Move the provider class files to the same path as the package in which they are defined. For example, if the provider is packaged as `com.sun.providers.myprovider.*`, move the provider class files to `/usr/sadm/lib/wbem/com/sun/wbem/myprovider/.class`.

3. **Set the Solaris Provider `CLASSPATH` variable to the directory that contains the provider class files as described in "To Set the Provider CLASSPATH" on page 123.**

4. **Stop the CIM Object Manager by typing the following command:**

   ```
   # /etc/init.d/init.wbem -stop
   ```

5. **Restart the CIM Object Manager by typing the following command:**

   ```
   # /etc/init.d/init.wbem -start
   ```

6. **Compile the `Native_Example.mof` file. For example:**

   ```
   % mofcomp Native_Example.mof
   ```

   Compiling this MOF file loads the `Native_Example` class in the CIM Object Manager and identifies `NativeProvider` as its provider.

7. **Run CIM WorkShop and view the `Native_Example` class. For example:**

   ```
   % /usr/sadm/bin/cimworkshop &
   ```

8. **In the Toolbar, click the Find Class icon.**

9. **In the Input dialog box, type `Native_Example` and click OK.**

CHAPTER 8

Error Messages

This chapter discusses the error messages generated by components of Solaris WBEM Services and the Sun WBEM SDK. The following topics are covered.

- How Error Messages are Generated
- Parts of Error Messages
- Finding Information About Error Messages
- Generated Error Messages

How Error Messages are Generated

The CIM Object Manager generates initial error messages that are used by both the MOF Compiler and the CIM WorkShop. The MOF Compiler appends a line to the error message indicating where in a `.mof` file the error occurred.

Parts of Error Messages

Error messages are made up of the following parts:

- Unique identifier – Character string that differentiates the error message from other error messages
- Exception message – Explanation of the error message
- Parameters – Placeholders for the specific classes, methods, and qualifiers that are cited in the exception message

Error Message Example

For example, the MOF Compiler may return the following error message:

```
REF_REQUIRED = Association class CIM_Docked needs
 at least two refs. Error in line 12.
```

- `REF_REQUIRED` is the unique identifier.
- `Association class CIM_Docked needs at least two refs` is the exception message.
- `CIM_Docked` is a parameter. A parameter can be replaced with the name of any appropriate class, property, method, or qualifier.

For Developers: Error Message Templates

WBEM provides exception templates for all possible error messages in the `ErrorMessages_en.properties` file of the APIs. In an exception template that requires parameters, the first parameter is represented as `{0}` and the second parameter is represented as `{1}`.

The following exception template is used in the previous example:

```
REF_REQUIRED = Association class {0} needs at least two refs.
```

Finding Information About Error Messages

You can search for the unique identifier of an error message in the *Javadoc* reference pages to receive an explanation about the content of the error message.

The following section provides a detailed explanation of each error message. The error messages are organized by unique identifiers. For each error message, the following types of information are provided, when applicable:

- Unique identifier, displayed as a heading.
- Description of the parameters used in the error message.
- Example of the error message as it is displayed to a user. This example is provided if the error message uses parameters to show how the error message will be displayed when elements, such as a class name, are substituted for the parameters.
- Cause, or reason why the error message was generated, and background or referential information that is helpful for understanding the error message.

- Solution, including steps you can take to resolve the error are provided when available.

Generated Error Messages

The following section lists and describes the error messages generated by the MOF Compiler, CIM Object Manager, and CIM WorkShop.

`ABSTRACT_INSTANCE`

Description:

The `ABSTRACT_INSTANCE` error message uses one parameter, `{0}`, which is replaced by the name of the abstract class.

Example:

`ABSTRACT_INSTANCE` = Abstract class `ExampleClass` cannot have instances.

Cause:

Instances were programmed for the specified class. However, the specified class is an abstract class, and abstract classes cannot have instances.

Solution:

Remove the programmed instances.

`CHECKSUM_ERROR`

The `CHECKSUM_ERROR` error message uses no parameters.

Example:

`CHECKSUM_ERROR` = Checksum not valid.

Cause:

The message could not be sent because it was damaged or corrupted. The damage could have occurred accidentally in transit or by a malicious third party.

Note – This error message is displayed when the CIM Object Manager receives an invalid checksum. A checksum is the number of bits in a packet of data passed over the network. This number is used by the sender and the receiver of the information to ensure that the transmission is secure and that the data has not been corrupted or intentionally modified during transit.

An algorithm is run on the data before transmission, and the checksum is generated and included with the data to indicate the size of the data packet. When the message is received, the receiver can recompute the checksum and compare it to the sender's checksum. If the checksums match, the transmission was secure and the data was not corrupted or modified.

Solution:

Resend the message using Solaris WBEM Services security features. For information about Solaris WBEM Services security, see "Administering Security" in *Solaris WBEM Services Administrator's Guide*.

`CIM_ERR_ACCESS_DENIED`

Description:

The `CIM_ERR_ACCESS_DENIED` error message does not use parameters.

Example:

`CIM_ERR_ACCESS_DENIED` = Insufficient privileges.

Cause:

This error message is displayed when a user does not have the appropriate privileges and permissions to complete an action.

Solution:

See your CIM Object Manager administrator to request privileges to complete the operation.

`CIM_ERR_ALREADY_EXISTS`

Instance 1: CIM_ERR_ALREADY_EXISTS

Description:

This instance of the `CIM_ERR_ALREADY_EXISTS` error message uses one parameter, `{0}`, which is replaced by the name of the duplicate class.

Example:

`CIM_ERR_ALREADY_EXISTS` = Duplicate class `CIMRack`

Cause:

The class you attempted to create uses the same name as an existing class.

Solution:

In CIM WorkShop, search for existing classes to see the class names that are in use, then create the class using a unique class name.

Instance 2: CIM_ERR_ALREADY_EXISTS

Description:

This instance of the `CIM_ERR_ALREADY_EXISTS` error message uses one parameter, `{0}`, which is replaced by the name of the duplicate instance.

Example:

`CIM_ERR_ALREADY_EXISTS` = Duplicate instance `SolarisRack`

Cause:

The instance for a class you attempted to create uses the same name as an existing instance.

Solution:

In CIM WorkShop, search for existing instances to see the names that are in use, then create the instance using a unique name.

Instance 3: CIM_ERR_ALREADY_EXISTS

Description:

This instance of the `CIM_ERR_ALREADY_EXISTS` error message uses one parameter, `{0}`, which is replaced by the name of the duplicate namespace.

Example:

`CIM_ERR_ALREADY_EXISTS` = Duplicate namespace `root/cimv2`

Cause:

The namespace you attempted to create uses the same name as an existing namespace.

Solution:

In CIM WorkShop, search for existing namespaces to see the names that are in use, then create the namespace using a unique name.

Instance 4: CIM_ERR_ALREADY_EXISTS

Description:

This instance of the CIM_ERR_ALREADY_EXISTS error message uses one parameter, {0}, which is replaced by the name of the duplicate qualifier type.

Example:

CIM_ERR_ALREADY_EXISTS = Duplicate qualifier type Key

Cause:

The qualifier type you attempted to create uses the same name as an existing qualifier type of the property it modifies.

Solution:

In CIM WorkShop, search for qualifier types that exist for the property to see the names that are in use, then create the qualifier type using a unique name.

CIM_ERR_FAILED

Description: Description

The CIM_ERR_FAILED error message uses one parameter, {0}, which is replaced by a character string, a message that explains the error condition and its possible cause.

Example: Example

CIM_ERR_FAILED=Invalid entry.

Cause: Cause

The CIM_ERR_FAILED error message is a generic message that can be displayed for a large number of different error conditions.

Solution

Because CIM_ERR_FAILED is a generic error message, many types of conditions can cause the message. The solution varies depending on the error condition.

CIM_ERR_INVALID_PARAMETER

Description:

The CIM_ERR_INVALID_PARAMETER error message uses one parameter, {0}, which is replaced by the name of the class that is missing a schema prefix.

Example:

CIM_ERR_INVALID_PARAMETER = Class System has no schema prefix.

Cause:

A class was created without providing a schema prefix in front of the class name. The Common Information Model requires that all classes are provided with a schema prefix. For example, classes developed as part of the CIM Schema require a `CIM` prefix: `CIM_Container`. Classes developed as part of the Solaris Schema require a `Solaris` prefix: `Solaris_System`.

Solution:

Provide the appropriate schema prefix for the class definition. Find all instances of the class missing the prefix and replace them with the class name and prefix.

`CIM_ERR_INVALID_SUPERCLASS`

Description:

The parameter `CIM_ERR_INVALID_SUPERCLASS` uses two parameters:

- `{0}` is replaced by the name of the specified subclass.
- `{1}` is replaced by the name of the class for which a specified subclass does not exist.

Example:

`CIM_ERR_INVALID_SUPERCLASS` = Superclass `CIM_Chassis` for class `CIM_Container` does not exist.

Cause:

A class is specified to belong to a particular superclass, but the superclass does not exist. The specified superclass may be misspelled, or a non-existent superclass name may have been specified accidentally in place of the intended superclass name. Or, the superclass and the subclass may have been interpolated: the specified superclass actually may be a subclass of the specified subclass. In the previous example, `CIM_Chassis` is specified as the superclass of `CIM_Container`, but `CIM_Chassis` is a subclass of `CIM_Container`.

Solution:

Check the spelling and the name of the superclass to ensure it is correct. Ensure that the superclass exists in the namespace.

`CIM_ERR_NOT_FOUND`

Instance 1: CIM_ERR_NOT_FOUND

Description:

This instance of the `CIM_ERR_NOT_FOUND` error message uses one parameter, `{0}`, which is replaced by the name of the non-existent class.

Example:

`CIM_ERR_NOT_FOUND` = Class `Solaris_Device` does not exist.

Cause:

A class is specified, but it does not exist. The specified class may be misspelled, or a non-existent class name may have been specified accidentally in place of the intended class name.

Solution:

Check the spelling and the name of the class to ensure that it is correct. Ensure that the class exists in the namespace.

Instance 2: CIM_ERR_NOT_FOUND

Description:

This instance of the error message `CIM_ERR_NOT_FOUND` uses two parameters:

- `{0}` is replaced by the name of the specified instance
- `{1}` is replaced by the name of the specified class

Example:

`CIM_ERR_NOT_FOUND` = Instance `Solaris_EnterpriseData` does not exist for class `Solaris_ComputerSystem`.

Cause:

The instance does not exist.

Solution:

Create the instance.

Instance 3: CIM_ERR_NOT_FOUND

Description:

This instance of the `CIM_ERR_NOT_FOUND` error message uses one parameter, `{0}`, the name of the specified namespace.

Example:

`CIM_ERR_NOT_FOUND` = Namespace `verdant` does not exist.

Cause:

The specified namespace is not found. This error may occur if the name of the namespace was entered incorrectly due to a typing error or spelling mistake.

Solution:

Retype the name of the namespace. Ensure that typing and spelling are correct.

CLASS_REFERENCE

Description:

The CLASS_REFERENCE error message uses two parameters.

- {0} parameter is replaced by the name of the class that was defined to participate in a reference.
- {1} parameter is replaced by the name of the reference.

Example:

CLASS_REFERENCE = Class SolarisExample1 must be declared as an association to have reference SolarisExample2

Cause:

A property was defined for a class to indicate that the class has a reference. However, the class is not part of an association relationship. A class can only be defined to have a reference as a property if it participates in an association relationship with another class.

Solution:

Create the association relationship, then set up the reference to the association as a property of this class.

INVALID_CREDENTIAL

Description:

The INVALID_CREDENTIAL error message does not use parameters.

Example:

INVALID_CREDENTIAL = Invalid credentials.

Cause:

This error message is displayed when an invalid password has been entered.

Solution:

If you receive this message from CIM WorkShop, delete the invalid password from the Password field of the CIM WorkShop authentication dialog box and type the password again. If this error message was received from the MOF Compiler, at the system prompt, log in again and type the correct password. Ensure that you spell the password correctly.

INVALID_QUALIFIER_NAME

Description:

The INVALID_QUALIFIER_NAME error message uses one parameter, {0}, which is replaced by the Managed Object Format notation that depicts an empty qualifier name.

Example:

INVALID_QUALIFIER_NAME = Invalid qualifier name " "

Cause:

A qualifier was created for a property, but a qualifier name was not specified.

Solution:

Include the qualifier name in the context of the qualifier definition.

KEY_OVERRIDE

Description:

The KEY_OVERRIDE error message uses two parameters:

- {0} parameter is replaced by the name of the non-abstract class that is put in an override relationship with a class that has one or more Key qualifiers.
- {1} parameter is replaced by the name of the concrete class that has the Key qualifier.

Example:

KEY_OVERRIDE = Non-key Qualifier SolarisCard cannot override key Qualifier SolarisLock.

Cause:

A non-abstract class, referred to as a concrete class, is put into an override relationship with a concrete class that has one or more Key qualifiers. In CIM, all concrete classes require at least one Key qualifier, and a non-Key class cannot override a class that has a Key.

Solution:

Create a Key qualifier for the non-Key class.

KEY_REQUIRED

Description:

The KEY_REQUIRED error message uses one parameter, {0} which is replaced by the name of the class that requires a key.

Example:

KEY_REQUIRED = Concrete (non-abstract) class ClassName needs at least one key.

Cause:

A Key qualifier was not provided for a concrete class. In CIM, all non-abstract classes, referred to as concrete classes, require at least one Key qualifier.

Solution:

Create a Key qualifier for the class.

`METHOD_OVERRIDDEN`

Description:

The `METHOD_OVERRIDDEN` command uses three parameters:

- `{0}` replaced by the name of the method that is trying to override the method represented by parameter `{1}`.
- `{1}` is replaced by the name of the method that has already been overridden by the method represented by parameter `{2}`.
- `{2}` is replaced by the name of the method that has overridden parameter `{1}`.

Example:

`METHOD_OVERRIDDEN` = Method `Resume ()` cannot override `Stop()` which is already overridden by `Start()`

Cause:

A method is specified to override another method that has already been overridden by a third method. Once a method has been overridden, it cannot be overridden again.

Solution:

Specify a different method to override.

`NEW_KEY`

Description:

The `NEW KEY` error message uses two parameters.

- `{0}` is replaced by the name of the key.
- `{1}` is replaced by the name of the class that is trying to define a new key.

Example:

`NEW_KEY` = Class `CIM_PhysicalPackage` cannot define new key `[Key]`

Cause:

A class is trying to define a new key when keys already have been defined in a superclass. Once keys have been defined in a superclass, new keys cannot be introduced into the subclasses.

Solution:

No action can be taken.

`NO_CIMOM`

Description:

The `NO_CIMOM` error message uses one parameter, `{0}` which is replaced by the name of the host that is expected to be running the CIM Object Manager.

Example:

`NO_CIMOM = CIMOM molly` not detected.

Cause:

The CIM Object Manager is not running on the specified host.

Solution:

Ensure that the CIM Object Manager is running on the host to which you are trying to connect. If the CM Object Manager is not running on that host, connect to a host running the CIM Object Manager.

`NO_INSTANCE_PROVIDER`

Description:

The `NO_INSTANCE_PROVIDER` error message uses two parameters:

- `{0}` is replaced by the name of the class for which the instance provider cannot be found.
- `{1}` is replaced by the name of the instance provider class that was specified.

Example:

`NO_INSTANCE_PROVIDER` = Instance provider `RPC_prop` for class `RPC_Agent` not found.

Cause:

The Java class of the specified instance provider is not found. This error message indicates that the class path of the CIM Object Manager does not contain one or more of the following:

- Name of the provider class
- Parameters of the provider class
- CIM class for which the provider is defined

Solution:

Set the CIM Object Manager environment variable.

`NO_METHOD_PROVIDER`

Description:

The NO_METHOD_PROVIDER error message uses two parameters:

- {0} is replaced by the name of the class for which the method provider cannot be found.
- {1} is replaced by the name of the method provider class that was specified.

Example:

NO_METHOD_PROVIDER = Method provider Start_prop for class RPC_Agent not found.

Cause:

The Java class of the specified method provider is not found. This error message indicates that the class path of the CIM Object Manager does not contain one or more of the following:

- Name of the provider class
- Parameters of the provider class
- CIM class for which the provider is defined

Solution:

Set the CIM Object Manager class path.

NO_OVERRIDDEN_METHOD

Description:

The error message NO_OVERRIDDEN_METHOD uses two parameters:

- {0} is replaced by the name of the method that has overridden the method represented by {1}.
- {1} is replaced by the name of the method that has been overridden.

Example:

NO_OVERRIDDEN_METHOD = Method Write overridden by Read does not exist in class hierarchy.

Cause:

The method of a subclass is trying to override the method of the superclass, but the method of the superclass already has been overridden by a method that belongs to another subclass. The overridden method that you are trying to override does not exist in the class hierarchy because it has never been defined.

When you override a method, you override its implementation and its signature.

Solution:

Ensure that the method exists in the superclass.

`NO_OVERRIDDEN_PROPERTY`

Description:

The `NO_OVERRIDDEN_PROPERTY` error message uses two parameters.

- `{0}` is replaced by the name of the property that has overridden `{1}`.
- `{1}` is replaced by the name of the overriding property.

Example:

`NO_OVERRIDDEN_PROPERTY` = Property `A` overridden by `B` does not exist in class hierarchy.

Cause:

The property of a subclass is trying to override the property of the superclass, but it doesn't succeed because the property of the superclass already has been overridden. The property that you are trying to override does not exist in the class hierarchy.

Solution:

Ensure that the property exists in the superclass.

`NO_PROPERTY_PROVIDER`

Description:

The `NO_PROPERTY_PROVIDER` error message uses two parameters:

- `{0}` is replaced by the name of the class for which the property provider cannot be found.
- `{1}` is replaced by the name of the property provider class that was specified.

Example:

`NO_PROPERTY_PROVIDER` = Property provider `Write_prop` for class `RPC_Agent` not found.

Cause:

The Java class of the specified property provider is not found. This error message indicates that the class path of the CIM Object Manager does not contain one or more of the following:

- Name of the provider class
- Parameters of the provider class
- CIM class for which the provider is defined

Solution:

Set the CIM Object Manager class path.

NO_QUALIFIER_VALUE

Description:

The NO_QUALIFIER_VALUE error message uses two parameters:

- {0} is replaced by the name of the qualifier that modifies the element {1}
- {1} is the element to which the qualifier refers. Depending on the qualifier, {1} can be a class, property, method, or reference.

Example:

NO_QUALIFIER_VALUE = Qualifier [SOURCE] for Solaris_ComputerSystem has no value.

Cause:

A qualifier was specified for a property or method, but values were not included for the qualifier. For example, the qualifier VALUES requires a string array to be specified. If the VALUES qualifier is specified without the required string array, the NO_QUALIFIER_VALUE error message is displayed.

Solution:

Specify the required parameters for the qualifier. For information about what attributes are required for which qualifiers, see the CIM Specification by the Distributed Management Task Force at the following URL: http://dmtf.org/spec/cims.html.

NO_SUCH_METHOD

Description:

The NO_SUCH_METHOD error message uses two parameters:

- {0} is replaced by the name of the specified method
- {1} is replaced by the name of the specified class

Example:

NO_SUCH_METHOD = Method Configure() does not exist in class Solaris_ComputerSystem

Cause:

Most likely, the method was not defined for the specified class. If the method is defined for the specified class, another method name may have been mispelled or typed differently in the definition.

Solution:

Define the method as an operation for the specified class. Otherwise, ensure that the method name and class name were typed correctly.

NO_SUCH_PRINCIPAL

Description:

The NO_SUCH_PRINCIPAL error message uses one parameter, {0}, which is replaced by the name of the principal, a user account.

Example:

NO_SUCH_PRINCIPAL = Principal molly not found.

Cause:

The specified user account cannot be found. The user name may have been mistyped upon login, or a user account has not been set up for the user.

Solution:

Ensure that the user name is spelled and typed correctly upon login. Ensure that a user account has been set up for the user.

NO_SUCH_QUALIFIER1

Description:

The NO_SUCH_QUALIFIER1 error message uses one parameter, {0}, which is replaced by the name of the undefined qualifier.

Example:

NO_SUCH_QUALIFIER1 = Qualifier [LOCAL] not found.

Cause:

A new qualifier was specified, but was not defined as part of the extension schema. The qualifier is required to be defined as part of the CIM Schema or an extension schema to be recognized as a valid qualifier for a property or method of a particular class.

Solution:

Define the qualifier as part of the extension schema or use a standard CIM qualifier. For information about standard CIM qualifiers and the usage of qualifiers in the CIM schema, see the CIM Specification by the Distributed Management Task Force at the following URL: http://www.dmtf.org/spec/cims.html.

NO_SUCH_QUALIFIER2

Description:

The NO_SUCH_QUALIFIER2 error message uses two parameters:

- {0} is replaced by the name of the class, property, or method that the qualifier modifies.
- {1} is replaced by the name of the qualifier that cannot be found.

Example:

NO_SUCH_QUALIFIER2 = Qualifier [LOCAL] not found for CIM_LogicalElement

Cause:

A new qualifier was specified to modify a property or method of a particular class. The qualifier was not defined as part of the extension schema. The qualifier is required to be defined as part of the CIM schema or an extension schema to be recognized as a valid qualifier for a property or method of a particular class.

Solution:

Define the qualifier as part of the extension schema or use a standard CIM qualifier. For information about standard CIM qualifiers and the usage of qualifiers in the CIM schema, see the CIM Specification by the Distributed Management Task Force at the URL, http://www.dmtf.org/spec/cims.html.

NO_SUCH_SESSION

Description:

The error message NO_SUCH_SESSION uses one parameter, {0}, which is replaced by the session identifier.

Example:

NO_SUCH_SESSION = No such session 4002.

Cause:

This message is displayed when a session has been infringed upon by an intruder. The CIM Object Manager removes the session when it detects that someone is trying to maliciously change data. For information about Solaris WBEM Services security features, see "Administering Security" in *Solaris WBEM Services Administrator's Guide*.

Solution:

Ensure that your CIM environment is secure.

NOT_HELLO

Description:

The NOT_HELLO error message uses no parameters.

Example:

NOT_HELLO = Not a Hello message.

Cause:

This error message is displayed if the data in the hello message—the first message sent to the CIM Object Manager—is corrupted, indicating a security breach.

Solution:

No action is available in response to this error message. For information about Solaris WBEM Services security features, see "Administering Security" in *Solaris WBEM Services Administrator's Guide*.

`NOT_INSTANCE_PROVIDER`

Description:

The `NOT_INSTANCE_PROVIDER` error message uses two parameters:

- `{0}` is replaced by the name of the instance for which the `InstanceProvider` interface is being defined.
- `{1}` is replaced by the name of the Java provider class that does not implement the `InstanceProvider` interface. The `InstanceProvider` interface must be implemented to enumerate all instances of the specified class.

Example:

`NOT_INSTANCE_PROVIDER` = `device_prop_provider` for class `Solaris_Provider` does not implement `InstanceProvider`.

Cause:

The path to the Java provider class specified by the CLASSPATH environment variable does not implement the `InstanceProvider` interface.

Solution:

Ensure that the Java provider class present in the class path implements the InstanceProvider interface. Use the following command when you declare the provider: `public Solaris implements InstanceProvider`. For information about how to implement Solaris WBEM Services providers, see Chapter 5.

`NOT_METHOD_PROVIDER`

Description:

The `NOT_METHOD_PROVIDER` error message uses two parameters:

- `{0}` is replaced by the name of the method for which the `MethodProvider` interface is being defined. The `MethodProvider` causes a specified method to be implemented in a program and enacted.
- `{1}` is replaced by the name of the Java provider class that does not implement the `MethodProvider` interface.

Example:

`NOT_METHOD_PROVIDER` = Provider `device_method_provider` for class `Solaris_Provider` does not implement `MethodProvider`.

Cause:

The Java provider class present in the class path does not implement the `MethodProvider` interface.

Solution:

Ensure that the Java provider class present in the class path implements the `MethodProvider` interface. Use the following command when you declare the provider: `public Solaris implements MethodProvider`. For information about how to implement Solaris WBEM Services providers, see Chapter 5.

`NOT_PROPERTY_PROVIDER`

Description:

The `NOT_PROPERTY_PROVIDER` error message uses two parameters:

- `{0}` is replaced by the name of the method for which the `PropertyProvider` interface is being defined. The `PropertyProvider` interface is required to retrieve the values of the specified property.
- `{1}`is replaced by the name of the Java provider class that does not implement the `PropertyProvider` interface.

Example:

`NOT_PROPERTY_PROVIDER` = Provider `device_property_provider` for class `Solaris_Provider` does not implement `PropertyProvider`.

Cause:

The Java provider class present in the class path does not implement the PropertyProvider interface.

Solution:

Ensure that the Java provider class present in the class path implements the PropertyProvider interface. Use the following command when you declare the provider: `public Solaris implements PropertyProvider`. For information about how to implement Solaris WBEM Services providers, see Chapter 5.

`NOT_RESPONSE`

Description:

The `NOT_RESPONSE` error message uses no parameters.

Example:

`NOT_RESPONSE` = Not a response message.

Cause:

This error message is displayed when the data in a first response message from the CIM Object Manager is corrupted, indicating a security breach.

Solution:

No action is available in response to this error message. For information about Solaris WBEM Services security features, see "Administering Security" in *Solaris WBEM Services Administrator's Guide*.

`PROPERTY_OVERRIDDEN`

Description:

The `PROPERTY_OVERRIDDEN` error message uses three parameters:

- `{0}` is replaced by the name of the property that is trying to override the property represented by parameter `{1}`.
- `{1}` is replaced by the name of the property that already has been overridden.
- `{2}` is replaced by the name of the property that has overridden the property represented by parameter `{1}`.

Example:

`PROPERTY_OVERRIDDEN` = Property `Volume` cannot override `MaxCapacity` which is already overridden by `RawCapacity`

Cause:

A property is specified to override another method that has already been overridden by a third method. Once a property has been overridden, it cannot be overridden again.

Solution:

Specify a different property to override.

`PS_CONFIG`

The `PS_CONFIG` error message uses one parameter, `{0}`, which is replaced by a description of the details that cause the error to occur. The description varies depending on the type of database used for the repository and the type of situation that causes the error message.

`PS_CONFIG` = The persistent store configuration is incorrect or has not been completed. You may need to run the `wbemconfig` script.

Cause:

Solaris WBEM Services requires the `wbemconfig` script to be run after installation. The `wbemconfig` script configures the persistent store and compiles the MOF files that provide the CIM and Solaris Schema classes. If the `wbemconfig` script was

not run after the Solaris WBEM Services installation, this error message occurs. If the repository was configured after installation and this error message occurs, the database configuration may have become corrupted.

Solution:

Run the `wbemconfig` script. For information about the `wbemconfig` script, see "Installing Solaris WBEM Services" in *Solaris WBEM Services Administrator's Guide*.

`PS_UNAVAILABLE`

Description:

The `PS_UNAVAILABLE` error message uses one parameter `{0}`, which is replaced by a message that describes why the persistent store became unavailable.

Example:

`PS_UNAVAILABLE` = The persistent store is unavailable. The exception thrown by the repository is 'segmentation fault.'

Cause:

This error message is displayed when the CIM Repository is unavailable. This situation could occur if the host on which the CIM Repository resides is brought down temporarily for maintenance, or if the host on which the CIM Repository resides becomes damaged and the repository is taken down and then restored on another host.

Solution:

If you receive this message while working in the CIM WorkShop, click the icon that causes the CIM WorkShop authentication dialog box to display. Then, in the Host field, type the name of another host that is running the CIM Repository and the CIM Object Manager. Type the namespace in the Namespace field, your user name and password, and log in. If you receive this message when running the MOF Compiler, type the following command to point to another host running the CIM Repository and CIM Object Manager: `mofcomp -c` *hostname* where `mofcomp` is the command to start the MOF Compiler, `-c` is the parameter that enables you to specify a host computer running the CIM Object Manager, and *hostname* is the name of the specified computer.

`QUALIFIER_UNOVERRIDABLE`

Description:

The `QUALIFIER_UNOVERRIDABLE` error message uses two parameters:

- `{0}` parameter is replaced by the name of the qualifier that is set with the `DisableOverride` flavor.
- `{1}` parameter is replaced by the name of the qualifier that is set to be disabled by `{0}`.

Example:

QUALIFIER_UNOVERRIDABLE = Test cannot override qualifier Standard because it has DisableOverride flavor.

Cause:

The ability of the specified qualifier to override another qualifier is disabled because the flavor of the specified qualifier has been set to DisableOverride or Override=False.

Solution:

Reset the ability of the qualifier to EnableOverride or to Override=True.

REF_REQUIRED

Description:

The REF_REQUIRED error message uses one parameter, {0}, which is replaced by the name of the class specified to participate in an association relationship.

Example:

REF_REQUIRED = Association class CIM_Chassis needs at least two refs.

Cause:

A class was set up to participate in an association, but no references were cited. The rules of the Common Information Model specify that an association must contain two or more references.

Solution:

Set up the references to the class, then set up the association.

SCOPE_ERROR

Description:

The SCOPE_ERROR command uses three parameters:

- {0} is replaced by the name of the class the specified qualifier modifies.
- {1} is replaced by the name of the specified qualifier.
- {2} is replaced by the type of attribute that the qualifier modifies.

Example:

SCOPE_ERROR = Qualifier [UNITS] for CIM_Container does not have a Property scope.

Cause:

A qualifier was specified in a manner that conflicts with the requirements of the CIM Specification. For example, the `[READ]` qualifier is defined in the CIM Specification to modify a Property. The scope of the `[READ]` qualifier is the definition that directs the `[READ]` qualifier to modify a Property. If the `[READ]` qualifier is used in a manner other than the direction of its scope—for example, if the `[READ]` qualifier is specified to modify a Method—the `SCOPE_ERROR` message is returned.

Note – The CIM Specification defines the types of CIM elements that a CIM qualifier can modify. This definition of the way in which a qualifier can be used is referred to as its scope. Most qualifiers, by definition, have a scope that directs them to modify properties or methods or both. Many qualifiers have a scope that directs them to modify parameters, classes, associations, indications, or schemas.

Solution:

Confirm the scope of the specified qualifier. Refer to the section, "1.Qualifiers" of the CIM Specification by the Distributed Management Task Force at the following URL:http://www.dmtf.org/spec/cim_spec_v20 for the standard definitions of CIM qualifiers. Use a different qualifier for the results you want to achieve, or change your program to use the qualifier according to its CIM definition.

`SIGNATURE_ERROR`

Description:

The `SIGNATURE_ERROR` error message uses no parameters.

Example:

`SIGNATURE_ERROR` = Signature not verified

Cause:

This message is displayed when a message is corrupted either accidentally or maliciously. It differs from the checksum error in that the message has a valid checksum, but the signature cannot be verified by the public key of the client. This protection ensures that even though the session key has been compromised, only the initial client which created the session is authenticated.

Solution:

No action is provided for this message, which is displayed when a session has been infringed upon by an intruder. For information about Solaris WBEM Services security features, see "Administering Security" in *Solaris WBEM Services Administrator's Guide*.

`TYPE_ERROR`

Description:

The TYPE_ERROR error message uses five parameters:

- {0} is replaced by the name of the specified element, such as a property, method, or qualifier.
- {1} is replaced by the name of the class to which the specified element belongs.
- {2} is replaced by the type defined for the element.
- {3} is replaced by the type of value assigned.
- {4} is replaced by the actual value assigned.

Example:

TYPE_ERROR = Cannot convert sint16 4 to a string for VolumeLabel in class Solaris_DiskPartition

Cause:

The value of a property or method parameter and its defined type are mismatched.

Solution:

Match the value of the property or method with its defined type.

UNKNOWNHOST

Description:

The UNKNOWNHOST error message uses one parameter, {0}, which is replaced by the name of the host.

Example:

UNKNOWNHOST = Unknown host molly

Cause:

A call was made to a specified host. The specified host is unavailable or cannot be located. It is possible that the host name was spelled incorrectly. It is also possible that the host computer was moved to a different domain or that the host name has not been registered in the list of hosts that belong to the domain. The host may be temporarily unavailable due to system conditions.

Solution:

Check the spelling of the host name. Ensure that no typing errors were made. Use the ping command to ensure that the host computer is responding. Check the system conditions of the host. Ensure that the host belongs to the specified domain.

VER_ERROR

Description:

The VER_ERROR error message uses one parameter, {0}, which is replaced by the version number of the running CIM Object Manager.

Example:

`VER_ERROR` = Unsupported version `0`.

Cause:

The upgraded version of Solaris WBEM Services does not support the current CIM Object Manager.

Solution:

Install the supported version.

APPENDIX **A**

Common Information Model (CIM) Terms and Concepts

CIM Concepts

The following sections describe basic CIM terms and concepts that are essential to understanding how network entities and management functions are described and related within the context of CIM. For more detailed information about the Common Information Model and object-oriented modeling practices, including how to model your own schema, refer to the CIM Tutorial provided by the Distributed Management Task Force.

Object-Oriented Modeling

CIM uses the principles of Object-Oriented Modeling, a way to represent an object, entity, concept, or function that has a physical or logical existence. The goal of Object-Oriented Modeling is to set a representation of a physical entity into a framework, or model, to express the qualities and functions of the entity and its relationships with other entities. In the context of CIM, Object-Oriented Modeling is used to model hardware and software elements.

Uniform Modeling Language

Models are expressed in the form of visual representation and language. CIM conventions for rendering the model are based on the diagrammatic concepts of Uniform Modeling Language UML. UML uses shapes to represent physical entities and lines to represent relationships. For example, in UML, classes are represented as rectangles. Each rectangle contains the name of the class it represents. A line between two rectangle represents a relationship between the two. A line that forks to join two classes to a higher-level class represents an association.

CIM diagrams add color to the diagrams to further express relationships:

- Red lines→Associations
- Blue lines→Inheritance relationships
- Green lines→Aggregation

CIM Terms

The following terms are innate to the CIM Schema.

Schema

The terms model, schema, and framework are synonymous. Each is an abstract representation of an entity that has a physical or logical existence. In CIM, a schema is a named collection of classes used for class naming and administration. Within a schema, classes and their subclasses are represented hierarchically using the following syntax: `Schemaname_classname.propertyname`. Each class name in a schema must be unique. Sun WBEM includes a Solaris Schema. It contains all classes specific to the Solaris extension to CIM.

Class and Instance

In WBEM, a class is a collection of objects that represents the most basic unit of management. For example, in Sun WBEM, the three main functional classes include `CIMClass`, `CIMProperty`, and `CIMInstance`.

Abstractly, classes are used to create managed objects. Class characteristics are inherited by the child objects, or instances, that are created from a class. For example, using `CIMClass`, you can create an instance, `CIMClass (Solaris_Computer_System)`.

This instance of `CIMClass` answers the question, "What is the computer system?" The value of the instance is `Solaris_Computer_System`. All instances of the same class type are created from the same class template. In the example, the name of the computer system provides a template to create managed objects of the type `Computer_System`.

Classes can be static or dynamic. Instances of static classes are stored by the CIM Object Manager and can be retrieved from the CIM Repository when a request is made. Instances of dynamic classes—classes containing data that changes regularly, such as system usage—are created by provider applications as the data changes.

Custom Classes: Extensions to CIM

For extensions to CIM, custom classes can be developed to support managed objects that are specific to their managed environment. The CIM Object Manager API provides new classes to extend CIM for the Solaris operating environment.

Property

A property defines a characteristic of a class. For example, using the `CIMProperty` class, you can define a key as a property of a particular CIM class. Values of properties can be passed back from the CIM Object Manager as a string or as a vector for a range of properties. Each property has a unique name and only one domain—the class that owns the property. A property of a given class can be overridden by a property of its subclass.

In Sun WBEM, an example of a property is the `CIMProperty`, which denotes the properties of a `CIMClass`.

Method

Like properties, methods belong to the class that owns them. A method is an action the objects of a given class are programmed to complete. For example, the method `public String getName()` returns the name of an instance as a concatenation of its keys and their values. Collectively, these actions describe the behavior of the class. Methods can belong only to the class that owns them. Within the context of a class, each method must have a unique name. A method of a given class can be overridden by a method of its subclass.

New classes inherit the definition of the method from the superclass, but not the implemented method. The definition of the method, indicated by a qualifier, serves as a placeholder in which a new implemented method can be provided. The CIM Object Manager checks for methods by starting from the lowest-level class and moving up the tree to the root class searching for a qualifier type that indicates a method.

Domain

Properties and methods are declared within a class. The class that owns the property or method is referred to as the domain of the property or method.

Qualifier and Flavor

A CIM qualifier is a modifier used to characterize CIM classes, properties, methods, and parameters. Qualifiers have unique attributes, including Name, Type, and Value, that are inherited by new classes.

Indication

An indication, an object and a type of class, is created as a result of the occurrence of an event. Indications can be arranged in a type hierarchy. Indications may have properties, methods, and triggers. Triggers are system operations, such as a change made to an existing class, or events that result in the creation of new instances of an indication.

Association

An association is a class that represents a relationship between two or more classes. Associations enable the creation of multiple relationship instances for a given class. System components can be related in many different ways, and associations provide a way of representing the relationships of these components.

Because of the way associations are defined, it is possible to establish a relationship between classes without affecting any of the related classes. The addition of an association does not affect the interface of the related classes. Only associations can have references.

Reference and Range

A reference is a type of property that defines the roles of objects involved in an association. The reference specifies the role name of the class in the context of the association. The domain of a reference is an association. The range of a reference is a character string that indicates the reference type.

Override

The override relationship is used to indicate the substitution of a property or method inherited from a subclass for a property or method inherited from the superclass. In CIM, guidelines determine what qualifiers of properties and methods can be overridden. For example, if the qualifier type of a class is flagged as a key, then the key cannot be overridden, because CIM guidelines specify that a key property cannot be overridden.

Core Model Concepts

The following sections provide descriptive information about the Core Model of CIM.

System Aspects of the Core Model

The Core Model provides classes and associations you can use to develop applications in which systems and their functions are represented as managed objects. These classes and associations embody the characteristics unique to all elements that comprise a system: physical and logical elements. Physical characteristics refer to the qualities of occupying space and conforming to the elementary laws of physics. Logical characteristics represent abstractions used to manage and coordinate aspects of the physical environment, such as system state or the capabilities of a system.

In the Core Model, logical elements can include the following.

TABLE A–1 Core Model Elements

Element Name	Description
Systems	A grouping of other logical elements. Because systems are themselves logical elements, a system can be composed of other systems.
Network Components	Classes that provide a topological view of a network.
Services and Access Points	Provide a mechanism for organizing the structures that provide access to the capabilities of a system.
Devices	An abstraction or emulation of a hardware entity, that may or may not be realized in physical hardware.

The following sections describe the classes and associations provided by the Core Model to emulate the qualities of systems.

System Classes Provided by the Core Model

The following table lists the classes that represent system aspects of the Core schema. The instances of these classes will most often belong to the descendents of the objects contained within the class.

TABLE A–2 Core Model System Classes

Class Name	Description	Example
`Managed System Element`	Base class for the system element hierarchy. Any distinguishable component of a system is a candidate for inclusion in this class.	Software components, such as files; and devices, such as disk drives and controllers, and physical components, such as chips and cards.
`Logical Element`	Base class for all the components of the system that represent abstract system components	Profiles, processes, or system capabilities in the form of logical devices.
`System`	Logical Element that aggregates an enumerable set of ManagedSystemElements. The aggregation operates as a functional whole. Within any particular subclass of System, there is a well-defined list of Managed System Element classes, whose instances must be aggregated.	Local Area Network, Wide Area Network, subnet, intranet
`Service`	Logical Element that contains the information necessary to represent and manage the functionality provided by a Device and/or SoftwareFeature. A Service is a general-purpose object to configure and manage the implementation of functionality. It is not the functionality itself.	Printer, modem, fax machine

System Associations Provided by the Core Model

Associations are classes that define the relationships shared by other classes. Association classes are flagged with an ASSOCIATION qualifer that denotes the purpose of the class. An association class must have at least two references, the names of the classes that share a particular relationship. Instances of an association always belong to the association class.

Associations can have the following types of relationships:

- One to one
- One to many
- One to zero
- Aggregation, such as a containment relationship between a system and its parts

Associations express the relationship between a system and the managed elements that make up the system. Two broad types of associations are used to define the relationships between classes:

The CIM Schema defines two basic types of associations:

- Component associations, which indicate that one class is part of another
- Dependency associations, which indicate that a class cannot function or exist without another class

These association types are abstract, which means that association classes do not have instances alone. Instances must belong to one of their descendent classes.

Component Associations

Component associations express the relationship between the parts of a system and the system itself. Component associations describe what elements make up a system. Abstract classes that express component associations are used to create concrete associations of this type in descendent classes. The descendent concrete associations answer the question: "What composition relationships does the component, or class, have with other components?"

In its most specialized role, the component association expresses the relationship between a system and its logical and physical parts.

Dependency Associations

Dependency associations establish the relationships between objects that rely on one another. The Core Model provides for the following types of dependencies:

- Functional—the dependent object cannot function without the object on which it depends
- Existence—the dependent object cannot exist without the object on which it depends

The following types of dependencies are included in the Core Model.

TABLE A–3 Core Model Dependencies

Dependency Association	Description

TABLE A–3 Core Model Dependencies *(Continued)*

`HostedService`	An association between a Service and the System on which the functionality resides. The cardinality of this association is one-to-many. A System may host many Services. Services are weak with respect to their hosting System. Generally speaking, a Service is hosted on the System where the LogicalDevices or SoftwareFeatures that implement the Service are located. The model does not represent Services hosted across multiple systems. This is modeled as an ApplicationSystem that acts as an aggregation point for Services that are each located on a single host.
`HostedAccessPoint`	An association between a ServiceAccessPoint (SAP) and the System on which it is provided. The cardinality of this association is one-to-many and is weak with respect to the System. Each System may host many SAPs. A feature of the model is that the access point of a service can be located on the same or a different host from the system to which the service provides access. This allows the model to depict both distributed systems (an ApplicationSystem with component Service on multiple hosts) and distributed access (a Service with access points hosted on other systems).
`ServiceSAPDependency`	An association between a Service and a ServiceAccessPoint indicating that the referenced SAP is required for the Service to provide its functionality.
`SAPSAPDependency`	An association between a SAP and another SAP indicating that the latter is required in order for the former to utilize or connect with its Service.
`ServiceAccessBySAP`	An association that identifies the access points for a Service. For example, a printer may be accessed by Netware, Apple Macintosh, or Windows ServiceAccessPoints, potentially hosted on different Systems.

Example of an Extension into the Core Model

It is possible to develop many extensions into the Core Model. One possible extension includes the addition of a Managed Element class as an abstraction of the Managed System Element class. Descendents of this Managed Element class—classes that represent objects outside the managed system domain, such as Users or Administrators—may be added to the Core Model.

Common Model Schemas

The Common Model provides a set of base classes for the following technology-specific schemas.

Systems

The Systems Model describes the computer, application, and network systems that comprise the top-level system objects that make up the managed environment.

Devices

The Devices Model is a representation of the discrete logical units on the system that provide the basic capabilities of the system, such as storage, processing, communication, and input/output functions. There is a strong temptation to identify the system devices with the physical components of the system. This approach is incorrect because what is being managed is not the physical components themselves but rather the operating system's representation of the devices.

The representation provided by the operating system does not have a one-to-one correspondence with the physical components of the system. For example, a modem may correspond to a discrete physical component. It may just as well be provided by a multi-function card that supports a LAN adapter as well as a modem, or the modem may be provided by an ordinary process running on the system. It is very important in using or making extensions to the model to understand this distinction between Logical Devices and Physical Components and not to get them confused.

Applications

The CIM Application Management Model is an information model designed to describe a set of details that is commonly required to manage software products and applications. This model can be used for various application structures, ranging from stand-alone desktop applications to a sophisticated, multiplatform, distributed, Internet-based application. Likewise, the model can be used to describe a single software product as well as a group of interdependent applications that form a business system.

A fundamental characteristic of the application model is the idea of the application life cycle. An application may be in one of four states: Deployable, Installable, Executable, and Executing. The interpretation and characteristics of the various objects used to represent applications are largely tied to the mechanisms used to transform applications from one state to another.

Networks

The Networks Model represents the various aspects of the network environment. This includes the topology of the network, the connectivity of the network, and the various protocols and services necessary to drive and provide access to the network.

Physical

The Physical Model provides a representation of the actual physical environment. Most of the managed environment is represented by logical objects, that is, objects that represent informational aspects of the environment rather than actual physical objects. Most of systems management is concerned with manipulating information that represents and controls the state of the system. Any impact on the actual physical environment (such as the movement of a read head on a physical drive, or the starting of a fan) is likely to only happen as an indirect consequence of the manipulation of the logical environment. As such, the physical environment is typically not of direct concern.

Apart from anything else, physical parts of the system are not instrumented. Their current state (and possibly even their very existence) can only be indirectly inferred from other information about the system. In the CIM, the physical model is a representation of this aspect of the environment and it is expected that it will differ dramatically from system to system and over time as technology evolves. It is also expected that the physical environment will always be very difficult to track and instrument, spawning the opportunity for a separate specialty, that of deploying applications, tools, and environments specifically aimed at providing information about the physical aspect of the managed environment.

Glossary

This Glossary defines terms used in the Sun WBEM documentation. Many of these terms are familiar to developers, but have new or altered meaning in the WBEM environment.

alias A symbolic reference in either a class or instance declaration to an object located elsewhere in a MOF file. Alias names follow the same rules as instance and class names. Aliases are typically used as shortcuts to lengthy paths.

aggregation relationship A relationship in which one entity is made up of the aggregation of some number of other entities.

association class A class that describes a relationship between two classes or between instances of two classes. The properties of an association class include pointers, or references, to the two classes or instances. All WBEM classes can be included in one or more associations.

Backus-Naur Form (BNF) A metalanguage that specifies the syntax of programming languages.

cardinality The number of values that may apply to an attribute for a given entity.

class A collection or set of objects that have similar properties and fulfill similar purposes.

CIM Object Manager Repository A central storage area managed by the Common Information Model Object Manager (CIM Object Manager). This repository contains the definitions of classes and instances that represent managed objects and the relationships among them.

CIM Schema A collection of class definitions used to represent managed objects that occur in every management environment.

See also core model, common model, and extension schema.

	The CIM is divided into the metamodel and the standard schema. The metamodel describes what types of entities make up the schema. It also defines how these entities can be combined into objects that represent managed objects.
common model	The second layer of the CIM schema, which includes a series of domain-specific but platform-independent classes. The domains are systems, networks, applications, and other management-related data. The common model is derived from the core model. *See also* extension schema.
core model	The first layer of the CIM schema, which includes the top-level classes and their properties and associations. The core model is both domain- and platform-independent. *See also* common model and extension schema.
Distributed Management Task Force (DMTF)	An industry-wide consortium committed to making personal computers easier to use, understand, configure, and manage.
domain	The class to which a property or method belongs. For example, if status is a property of Logical Device, it is said to belong to the Logical Device domain.
dynamic class	A class whose definition is supplied by a provider at runtime as needed. Dynamic classes are used to represent provider-specific managed objects and are not stored permanently in the CIM Object Manager Repository. Instead, the provider responsible for a dynamic class stores information about its location. When an application requests a dynamic class, the CIM Object Manager locates the provider and forwards the request. Dynamic classes support only dynamic instances.
dynamic instances	An instance that is supplied by a provider when the need arises and is not stored in the CIM Object Manager Repository. Dynamic instances can be provided for either static or dynamic classes. Supporting instances of a class dynamically allows a provider to always supply up-to-the-minute property values.
enumeration	Java term for getting a list of objects. Java provides an `Enumeration` interface that has methods for enumerating a list of objects. An individual object on this list to be enumerated is called an element.
extension schema	The third layer of the CIM Schema, which includes platform-specific extensions of the CIM Schema such as Solaris and UNIX. *See also* common model and core model.
flavor	*See* qualifier flavor.

indication An operation executed as a result of some action such as the creation, modification, or deletion of an instance, access to an instance, or modification or access to a property. Indications can also result from the passage of a specified period of time. An indication typically results in an event.

inheritance The relationship that describes how classes and instances are derived from parent classes or superclasses. A class can spawn a new subclass, also called a child class. A subclass contains all the methods and properties of its parent class. Inheritance is one of the features that allows WBEM classes to function as templates for actual managed objects in the WBEM environment.

instance A representation of a managed object that belongs to a particular class, or a particular occurrence of an event. Instances contain actual data.

instance provider A type of provider that supports instances of system- and property-specific classes. Instance providers can support data retrieval, modification, deletion, and enumeration. Instance providers can also invoke methods.

See also property provider.

interface class The class used to access a set of objects. The interface class can be an abstract class representing the scope of an enumeration.

See also enumeration and scope.

Interface Definition Language (IDL) A generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language.

key A property that is used to provide a unique identifier for an instance of a class. Key properties are marked with the Key qualifier.

Key qualifier A qualifier that must be attached to every property in a class that serves as part of the key for that class.

managed object A hardware or software component that is represented as an instance of the CIM class. Information about managed objects is supplied by providers as well as the CIM Object Manager.

See also managed resource.

Managed Object Format (MOF) A compiled language for defining classes and instances. The MOF compiler (`mofc`) compiles `.mof` text files into Java classes and adds the data to the CIM Object Manager Repository. MOF eliminates the need to write code, thus providing a simple and fast technique for modifying the CIM Object Manager Repository.

managed resource	A hardware or software component that can be managed by a management application. Hard disks, CPUs, and operating systems are examples of managed resources. Managed resources are described in WBEM classes. *See also* managed object.
management application	An application or service that uses information originating from one or more managed objects in a managed environment. Management applications retrieve this information through calls to the CIM Object Manager API from the CIM Object Manager and from providers.
management information base	A database of managed objects.
metamodel	A CIM component that describes the entities and relationships representing managed objects. For example, classes, instances, and associations are included in the metamodel.
metaschema	A formal definition of the Common Information Model, which defines the terms used to express the model, its usage, and its semantics.
method	A function describing the behavior of a class. Including a method in a class does not guarantee an implementation of the method.
MOF file	A text file that contains definitions of classes and instances using the Managed Object Format (MOF) language.
Named Element	An entity that can be expressed as an object in the metaschema.
namespace	A directory-like structure that can contain classes, instances, and other namespaces.
object path	A formatted string used to access namespaces, classes, and instances. Each object on the system has a unique path which identifies it locally or over the network. Object paths are conceptually similar to Universal Resource Locators (URLs).
override	Indicates that the property, method, or reference in the derived class overrides the similar construct in the parent class in the inheritance tree or in the specified parent class.
polymorphism	The ability to alter methods and properties in a derived class without changing their names or altering interfaces. For example, a subclass can redefine the implementation of a method or property inherited from its superclass. The property or method is thereby redefined even if the superclass is used as the interface class.

	Thus, the LogicalDevice class can define the variable status as a string, and can return the values "on" or "off." The Modem subclass of LogicalDevice can redefine (override) status by returning "on," "off," and "connected." If all LogicalDevices are enumerated, any LogicalDevice that happens to be a modem can return the value "connected" for the status property.
property	A value used to characterize the instances of a class. Property names cannot begin with a digit and cannot contain white space. Property values must have a valid Managed Object Format (MOF) data type.
property provider	A program that communicates with managed objects to access data and event notifications from a variety of sources, such as the Solaris operating environment or a Simple Network Management Protocol (SNMP) SNMP device. Providers forward this information to the CIM Object Manager for integration and interpretation.
qualifier	A modifier containing information that describes a class, an instance, a property, a method, or a parameter. The three categories of qualifiers are: those defined by the Common Information Model (CIM), those defined by WBEM (standard qualifiers), and those defined by developers. Standard qualifiers are attached automatically by the CIM Object Manager.
qualifier flavor	An attribute of a CIM qualifier that governs the use of a qualifier. WBEM flavors describe rules that specify whether a qualifier can be propagated to derived classes and instances and whether or not a derived class or instance can override the qualifier's original value.
range	A class that is referenced by a reference property.
reference	A special string property type that is marked with the reference qualifier, indicating that it is a pointer to other instances.
required property	A property that must have a value.
schema	A collection of class definitions that describe managed objects in a particular environment.
scope	An attribute of a CIM qualifier that indicates which CIM elements can use the qualifier. Scope can only be defined in the Qualifier Type declaration; it cannot be changed in a qualifier.
selective inheritance	The ability of a descendant class to drop or override the properties of an ancestral class.
Simple Network Management Protocol (SNMP)	A protocol of the Internet reference model used for network management.
singleton class	A WBEM class that supports only a single instance.

Solaris Schema	A Sun extension to the CIM Schema that contains definitions of classes and instances to represent managed objects that exist in a typical Solaris operating environment.
standard schema	A common conceptual framework for organizing and relating the various classes representing the current operational state of a system, network, or application. The standard schema is defined by the Distributed Management Task Force (DMTF) in the Common Information Model (CIM).
static class	A WBEM class whose definition is persistent. The definition is stored in the CIM Object Manager Repository until it is explicitly deleted. The CIM Object Manager can provide definitions of static classes without the help of a provider. Static classes can support either static or dynamic instances.
static instance	An instance that is persistently stored in the CIM Object Manager Repository.
subclass	A class that is derived from a superclass. The subclass inherits all features of its superclass, but can add new features or redefine existing ones.
subschema	A part of a schema owned by a particular organization. The Win32 and Solaris Schemas are examples of subschemas.
superclass	The class from which a subclass inherits.
transitive dependency	In a relation having at least three attributes R (A, B, C), the situation in which A determines B, B determines C, but B does not determine A.
trigger	A recognition of a state change (such as create, delete, update, or access) of a class instance, and update or access of a property. The WBEM implementation does not have an explicit object representing a trigger. Triggers are implied either by the operations on basic objects of the system (create, delete, and modify on classes, instances and namespaces) or by events in the managed environment.
Unified Modeling Language (UML)	A notation language used to express a software system using boxes and lines to represent objects and relationships.
Unicode	A 16-bit character set capable of encoding all known characters and used as a worldwide character-encoding standard.
UTF-8	An 8-bit transformation format that may also serve as a transformation format for Unicode character data.
virtual function table (VTBL)	A table of function pointers, such as an implementation of a class. The pointers in the VTBL point to the members of the interfaces that an object supports.

Win32 Schema A Microsoft extension to the CIM Schema that contains definitions of classes and instances to represent managed objects that exist in a typical Win32 environment.

Index

D

E

H

I