Java Dynamic Management Kit 4.2 Tutorial

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

Part Number 806-6633–10
December 2000

Adobe PostScript™

Please
Recycle

Contents

Preface

The Java Dynamic Management™ Kit provides a set of Java™ classes and tools for developing management solutions. This product conforms to the Java Management extensions (JMX™) v1.0 Final Release, which define resource instrumentation, dynamic agents and remote management applications. The JMX architecture is applicable to network management, remote system maintenance, application provisioning, and the new management needs of the service-based network.

Once you are familiar with management concepts, the *Java Dynamic Management Kit 4.2 Tutorial* is intended to demonstrate each of the management levels and how they interact. The lessons of this tutorial will show you:

- The different ways of making your resources manageable
- How to write an agent and add management services dynamically
- How to access your resources from a remote management application
- The mechanism used to forward events and exceptions from agent to manager

Taken as a whole, these topics will demonstrate the complete development process for implementing a management solution in the Java programming language.

This book ends with a lesson devoted to the details of programming SNMP managers and agents (peers) using the Java Dynamic Management Kit.

Who Should Use This Book

This tutorial is aimed at developers who would like to learn how to instrument new or existing resources for management, write dynamic agents, or write management applications. Familiarity with Java programming is assumed. Some tutorials also rely on system and network management concepts: knowledge of these is helpful though not required.

This book is not intended to be an exhaustive reference. Management concepts and product features are covered in *Getting Started with the Java Dynamic Management Kit 4.2*, and the complete Javadoc™ API definitions are provided in the product's online documentation package.

Before You Read This Book

In order to build and run the sample programs in this tutorial or use the tool commands provided in the Java Dynamic Management Kit, you must have a complete installation of the product on your machine. Please refer to the *Java Dynamic Management Kit 4.2 Installation Guide and Release Notes* document for instructions on how to install the product components and configure your environment.

Before programming with the Java Dynamic Management Kit, you should be familiar with the concepts and tools used throughout these tutorials. The following books are part of the product documentation set:

- *Getting Started with the Java Dynamic Management Kit 4.2*
- *Java Dynamic Management Kit 4.2 Tools Reference*

These books are available online after you have installed the documentation package of the Java Dynamic Management Kit. The online documentation also includes the Javadoc API for the Java packages and classes, including those of the Java Management extensions. Using any web browser, open the homepage corresponding to your platform:

Operating Environment	**Homepage Location**
Solaris	*installDir*`/SUNWjdmk/jdmk4.2/`*JDKversion*`/index.html`
Windows NT	*installDir*`\SUNWjdmk\jdmk4.2\`*JDKversion*`\index.html`

In these file names, *installDir* refers to the base directory of your Java Dynamic Management Kit installation. In a default installation procedure, *installDir* is:

- `/opt` on the Solaris platform
- `C:\Program Files` on the Windows NT platform

The *JDKversion* is that of the Java Development Kit (JDK™) which you use and which you selected during installation. Its value can be either `1.1` or `1.2` when used in a directory, filename, or path.

These conventions are used throughout this book whenever referring to files or directories which are part of the installation.

Directories and Classpath

These tutorials are based on the example programs shipped with the Java Dynamic Management Kit. Each example is a set of Java source code files in a separate subdirectory. The following table gives the location of the main examples directory:

Operating Environment	**Examples Directory**
Solaris	*installDir*`/SUNWjdmk/jdmk4.2/`*JDKversion*`/examples`
Windows NT	*installDir*`\SUNWjdmk\jdmk4.2\`*JDKversion*`\examples`

Except where noted, the source code in this book is taken from these example programs. However, code fragments may be rearranged and comments may be changed. Program listings in the tutorials usually simplify comments and remove output statements for space considerations.

On the Solaris platform, you must have root access in order to write in the installed examples directory. For this reason, it may be necessary to copy all examples to a different location before compiling them. Throughout the rest of this book, we will use the term *examplesDir* to refer to the main examples directory in a location where you can compile and run them.

When either compiling or running the example programs, the jar file for the Java Dynamic Management Kit runtime libraries must be in your classpath:

JDK Version	**Classpath for Compiling or Running the Examples on Solaris**
1.1	`.:`*installDir*`/SUNWjdmk/jdmk4.2/1.1/lib/jdmkrt.jar:` *installDir*`/SUNWjdmk/jdmk4.2/1.1/lib/collections.jar`
1.2	`.:`*installDir*`/SUNWjdmk/jdmk4.2/1.2/lib/jdmkrt.jar`

The classpathes on the Windows NT platform are identical to these, with the forward slashes (`/`) replaced with back-slashes (\), and the colons (`:`) replaced with semi-colons (`;`).

These classpathes assumes that you are in the subdirectory of a particular example when compiling or running it. You specify the classpath on the command line of the `javac` and `java` tools with the `-classpath` option. The JDK version must match the version of the `javac` or `java` command that you are using.

Throughout the rest of this book, we will use the term *classpath* in command line examples to indicate that you must use the classpath indicated above. You may also define this classpath in an environment variable according to your platform and omit its definition on the command line.

In order to use the `proxygen` and `mibgen` tools provided with the Java Dynamic Management Kit, you should add the installation's binaries directory your environment's path. The following table give the location of this directory:

Operating Environment	**Binaries Directory**
Solaris	*installDir*`/SUNWjdmk/jdmk4.2/`*JDKversion*`/bin`
Windows NT	*installDir*`\SUNWjdmk\jdmk4.2\`*JDKversion*`\bin`

How This Book Is Organized

This book is organized like a *trail* of the *The Java Tutorial*. Each major part is a lesson covering a subject and each chapter covers a topic within that subject.

Part I, "Instrumentation through MBeans" shows various ways of making a resource manageable. Topics:

- "Standard MBeans"
- "Dynamic MBeans"
- "Model MBeans"

Part II, "Agent Applications" demonstrates the functionality of the MBean server at the heart of an agent. Topics:

- "The MBean Server in a Minimal Agent"
- "The HTML Protocol Adaptor"

- “The Base Agent”
- “The Notification Mechanism”

Part III, “Remote Management Applications” covers how the Java Dynamic Management Kit simplifies how a distant manager may access the resources in an agent. Topics:

- “Protocol Connectors”
- “MBean Proxies”
- “Notification Forwarding”
- “Access Control and Security”

Part IV, “Agent Services” demonstrates the various kinds of management intelligence which can be dynamically added to an agent. Topics:

- “The M-Let Class Loader”
- “The Relation Service”
- “Cascading Agents”
- “The Discovery Service”

Part VI, “SNMP Interoperability” shows how Java agents can also implement an SNMP agent and how to write a manager with the SNMP manager API. Topics:

- “Creating an SNMP Agent”
- “Developing an SNMP Manager”
- “Security Mechanisms in the SNMP Toolkit”
- “Implementing an SNMP Proxy”

Related Books

The Java Dynamic Management Kit relies on the management architecture of the Java Management extensions. The specification document, *Java Management Extensions Instrumentation and Agent Specification, v1.0* (Final Release, July 2000) is provided in the product documentation package under the filename `jmx_instr_agent.pdf`.

The structure of this book was inspired by that of the *The Java Tutorial*:

- Online version:
  `http://java.sun.com/docs/books/tutorial/index.html`
- Paperback reference:
  The Java Tutorial Second Edition: Object-Oriented Programming for the Internet (Java Series) by Mary Campione and Kathy Walrath; 2nd book and CD-ROM edition (March 1998) Addison-Wesley Pub. Co.; ISBN: 0201310074

Some topics in the lesson on SNMP refer to RFC “standards” for further information. The complete text of RFC papers can be found on the following site:

`http://sunsite.auc.dk/RFC/`

Disclaimer – This site is in no way affiliated with Sun Microsystems, Inc. and Sun makes no claim as to the accuracy or relevance of the data it contains.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks selected 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	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.
`AaBbCc123`	Class or object names, methods, parameters or any other element of the Java programming language	Instantiate the `MyBean` class.
`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	A placeholder: replace with the appropriate name or intended value	To delete a file, type **`rm`** *filename*.

Shell Prompts

The following table shows the default system prompts for the different platforms and shells.

TABLE P–2 Shell Prompts

Shell	Prompt
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	#
Windows NT system prompt	`C:\>`

Unless otherwise noted, the command examples in this book use the Korn shell.

PART I

Instrumentation through MBeans

Given a resource in the Java programming language, either an application, a service or an object representing a device, its instrumentation is the way that you expose its management interface. The management interface is the set of attributes and operations that are visible to managers wishing to interact with that resource. Therefore, instrumenting a resource makes it manageable.

This lesson covers the three ways to instrument a resource: by writing a standard MBean, by implementing a dynamic MBean, or by dynamically instantiating a configurable model MBean.

When you write a standard MBean, you follow certain design patterns so that the method names in your object exposes the attributes and operations to static introspection. Dynamic MBeans all implement a generic interface and may expose a rich description of their management interface. Model MBeans are MBean templates whose management interface and resource target are defined at runtime.

This lesson contains the following topics:

- "Standard MBeans" shows how to write a standard MBean by following the design patterns defined by the Java Management extensions. The example shows how an agent then accesses the attributes and operations.
- "Dynamic MBeans" shows how to implement the `DynamicMBean` interface in order to expose a coherent management interface. Running the example highlights the similarities and differences between dynamic and standard MBeans, with an analysis of performance issues.
- "Model MBeans" gives an example of how to create a model MBean, configure its behavior, set its target object, and then manage it in the same way as any other MBean.

CHAPTER **1**

Standard MBeans

A standard MBean is the simplest and fastest way to instrument a resource from scratch: attributes and operations are simply methods which follow certain design patterns. A standard MBean is composed of the `MBean` interface which lists the methods for all exposed attributes and operations, and of the class which implements this interface and provides the functionality of the resource.

The code samples in this topic are taken from the files in the `StandardMBean` example directory located in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

- “Exposing the `MBean` Interface” on page 23 demonstrates the design patterns for attributes and operations and gives some general rules for writing the `MBean` interface.
- “Implementing the MBean” on page 26 shows how the `MBean` interface is related to the code for the manageable resource.
- “Running the Standard MBean Example” on page 28 demonstrates the runtime behavior of a standard MBean.

Exposing the `MBean` Interface

Typically, you would first determine the management interface of your resource, that is the information needed to manage it. This information is expressed as attributes and operations. An attribute is a value of any type that a manager can get or set remotely. An operation is a method with any signature and any return type that the manager can invoke remotely.

Note - Attributes and operations are conceptually equivalent to properties and actions on JavaBeans objects. However, their translation into Java code is entirely different to accommodate the management functionality.

As specified by the Java Management extensions for instrumentation, all attributes and operations are explicitly listed in an `MBean` interface. This is a Java interface that defines the full management interface of an MBean. This interface must have the same name as the class that implements it, followed by the `MBean` suffix. Since the interface and its implementation are usually in different files, there are two files which make up a standard MBean.

For example, the class `SimpleStandard` (in the file `SimpleStandard.java`) will have its management interface defined in the interface `SimpleStandardMBean` (in the file `SimpleStandardMBean.java`).

CODE EXAMPLE 1–1 The `SimpleStandardMBean` Interface

```
public interface SimpleStandardMBean {

    public String getState() ;

    public void setState(String s) ;

    public Integer getNbChanges() ;

    public void reset() ;
}
```

Only `public` methods in the `MBean` interface are taken into consideration for the management interface. When present, non-public methods should be grouped separately, to make the code clearer for human readers.

Attributes

Attributes are conceptual variables that are exposed for management through getter and setter methods in the `MBean` interface:

- A *getter* is any public method whose name begins with `get` and which doesn't return void; it lets a manager read the value of the attribute, whose type is that of the returned object
- A public method whose name begins with `is` and which returns a boolean or `Boolean` object is also a getter, though a boolean attribute may have only one getter (it must be one form or the other)

- A *setter* is any public method whose name begins with `set` and which takes a single parameter; it lets a manager write a new value in the attribute, whose type is that of the parameter

Attribute types can be arrays of objects, but individual array elements cannot be accessed individually through the getters and setters. Use operations to access the array elements, as described below. The following code example demonstrates an attribute with an array type:

```
public String[] getMessages();
public void setMessages(String[] msgArray);
```

The name of the attribute is the literal part of the method name following `get`, `is`, or `set`. This name is case sensitive in all Java Dynamic Management Kit objects that manipulate attribute names. Using these patterns, we can determine the attributes exposed in the code sample above:

- `State` is a readable and writeable attribute of type String
- `NbChanges` is a read-only attribute of type Integer

The specification of the design patterns for attributes implies the following rules:

- Attributes may be read-only, write-only, or readable and writeable
- Attribute names cannot be overloaded: for any given attribute name there can be at most one setter and one getter, and if both are defined, they must use the same type

Operations

Operations are methods that management applications can call remotely on a resource. They can be defined with any number of parameters of any type and can return any type.

The design patterns for operations are simple: any public method defined in the `MBean` interface that is not an attribute getter or setter is an operation. For this reason, getters and setters are usually declared first in the Java code, so that all operations are grouped afterwards. The name of an operation is the name of the corresponding method.

The `SimpleStandardMBean` in the example defines one operation, `reset`, which takes no parameters and returns nothing.

While the following methods define valid operations (and not attributes), these types of names should not be used to avoid confusion:

```
public void getFoo();
public Integer getBar(Float p);
public void setFoo(Integer one, Integer two);
public String isReady();
```

For performance reasons, you may want to define operations for accessing individual elements of an array type attribute. In this case, use non-ambiguous operation names:

```
public String singleGetMessage(int index);
public void singleSetMessage(int index, String msg);
```

Note - The Java Dynamic Management Kit imposes no restrictions on attribute types, operation attribute types, and operation return types. However, the developer must insure that the corresponding classes are available to all applications manipulating these objects, and that they are compatible with the type of communication used. For example, attribute and operation types must be serializable in order to be manipulated remotely using the RMI or HTTP protocol.

Implementing the MBean

The second part of an MBean is the class that implements the `MBean` interface. This class encodes the expected behavior of the manageable resource in its implementation of the attribute and operation methods. Of course, the resource does not need to reside entirely in this class, the MBean implementation can rely on other objects.

Beyond the implementation of the corresponding `MBean` interface, there are two requirements on the MBean class:

- It must be a concrete class so that it can be instantiated
- It must expose at least one public constructor so that any other class can create an instance

Otherwise, the developer is free to implement the management interface in any way, provided of course that the object has the expected behavior. Here is the sample code that implements our `MBean` interface:

CODE EXAMPLE 1–2 The `SimpleStandard` Class

```
public class SimpleStandard implements SimpleStandardMBean {

     public String getState() {
        return state;
    }

    public void setState(String s) {
        state = s;
        nbChanges++;
    }

```

(continued)

```
    public Integer getNbChanges() {
        return new Integer(nbChanges);
    }

    public void reset() {
        state = "initial state";
        nbChanges = 0;
        nbResets++;
    }

    // This method is not a getter in the management sense because
    // it is not exposed in the "SimpleStandardMBean" interface.
    public Integer getNbResets() {
        return new Integer(nbResets);
    }

    // internal variables for exposed attributes
    private String      state = "initial state";
    private int         nbChanges = 0;

    // other private variables
    private int         nbResets = 0;
}
```

In this example there is no constructor. Since the Java compiler provides a public, no-argument constructor by default in such cases, this is a valid MBean.

As in this example, attributes are usually implemented as internal variables whose value is returned or modified by the getter and setter methods. However, an MBean may implement any access and storage scheme to fit particular management needs, provided getters and setters retain their read and write semantics. Methods in the MBean implementation may have side-effects, but it is up to the programmer to insure that these are safe and coherent within the full management solution.

As we shall see later, management applications never have a direct handle on an MBean. They only have an identification of an instance and the knowledge of the management interface. In this case, the mechanism for exposing attributes through methods in the `MBean` interface makes it impossible for an application to access the MBean directly. Internal variables and methods, and even public ones, are totally encapsulated and their access is controlled by the programmer through the implementation of the `MBean` interface.

Running the Standard MBean Example

The *examplesDir*/`StandardMBean` directory contains the `SimpleStandard.java` and `SimpleStandardMBean.java` files which make up the MBean. This directory also contains a simple agent application which instantiates this MBean, introspects its management interface and manipulates its attributes and operations.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/StandardMBean/
$ javac -classpath classpath *.java
```

To run the example, launch the agent class which will interact with the `SimpleStandard` MBean:

```
$ java -classpath classpath StandardAgent
```

Press `<Enter>` when the application pauses to step through the example. The agent application handles all input and output in this example and gives us a view of the MBean at runtime.

We will look at how agents work in "Dynamic Agents", but this example demonstrates how the `MBean` interface limits the view of what the MBean exposes for management. Roughly, the agent introspects the MBean interface at runtime to determine what attributes and operations are available. You then see the result of calling the getters, setters and operations.

The lesson on agents will also cover the topics of object names and exceptions which you see when running this example.

CHAPTER 2

Dynamic MBeans

A dynamic MBean implements its management interface programmatically, instead of through static method names. To do this, it relies on metadata classes which represent the attributes and operations exposed for management. Management applications then call generic getters and setters whose implementation must resolve the attribute or operation name to its intended behavior.

One advantage of this instrumentation is that you can use it to quickly make an existing resource manageable. The implementation of the `DynamicMBean` interface can provide an instrumentation wrapper for an existing resource.

Another advantage is that the metadata classes for the management interface can provide human-readable descriptions of the attributes, operations and MBean itself. This information could be displayed to a user on a management console to describe how to interact with this particular resource.

The code samples in this topic are taken from the files in the `DynamicMBean` example directory located in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

- “Exposing the Management Interface” on page 30 explains the `DynamicMBean` interface and its generic methods common to all dynamic MBeans.
- “Implementing a Dynamic MBean” on page 32 shows how to implement this interface to expose specific attributes and operations.
- “Running the Dynamic MBean Example” on page 41 demonstrates the runtime behavior of a dynamic MBean.

Exposing the Management Interface

In the standard MBean, attributes and operations are exposed statically in the names of methods in the MBean interface. Dynamic MBeans all share the same interface which defines generic methods to access attributes and operations. Since the management interface is no longer visible through introspection, dynamic MBeans must also provide a description of their attributes and operations explicitly.

The DynamicMBean Interface

The DynamicMBean class is a Java interface defined by the Java Management extensions. It specifies the methods that a resource implemented as a dynamic MBean must provide to expose its management interface. Here is an uncommented version of the code:

CODE EXAMPLE 2–1 The DynamicMBean Interface

```
public interface DynamicMBean {

    public Object getAttribute(String attribute) throws
        AttributeNotFoundException, MBeanException, ReflectionException;

    public void setAttribute(Attribute attribute) throws
        AttributeNotFoundException, InvalidAttributeValueException,
        MBeanException, ReflectionException ;

    public AttributeList getAttributes(String[] attributes);

    public AttributeList setAttributes(AttributeList attributes);

    public Object invoke(
        String actionName, Object params[], String signature[])
        throws MBeanException, ReflectionException ;

    public MBeanInfo getMBeanInfo();
 }
```

The getMBeanInfo method is the one which provides a description of the MBean's management interface. This method returns an MBeanInfo object which contains the metadata information about attributes and operations.

The attribute getters and setters are generic, since they take the name of the attribute which needs to be read or written. For convenience, dynamic MBeans must also define bulk getters and setters to operate on any number of attributes at once. These methods use the `Attribute` and `AttributeList` classes to represent attribute name-value pairs and lists of name-value pairs, respectively.

Since the names of the attributes are not revealed until runtime, the getters and setters are necessarily generic. In the same way, the `invoke` method takes the name of an operation and its signature, in order to invoke any method which might be exposed.

As a consequence of implementing generic getters, setters, and invokers, the code for a dynamic MBean is more complex than for a standard MBean. For example, instead of having a specific getter called by name, the generic getter must verify the attribute name and then encode the functionality to read each of the possible attributes.

The MBean Metadata Classes

A dynamic MBean has the burden of building the description of its own management interface. The JMX specification defines the Java objects used to completely describe the management interface of an MBean. Dynamic MBeans use these objects to provide a complete self description as returned by the `getMBeanInfo` method. Agents also use these classes to describe a standard MBean after it has been introspected.

As a group, they are referred to as the *MBean metadata classes* because they provide information about the MBean. This information includes the attributes and operations of the management interface but also the list of constructors for the MBean class and the notifications that the MBean may send. Notifications are event messages that are defined by the JMX architecture; they are fully covered in “The Notification Mechanism”.

Each element is described by its metadata object containing its name, a description string, and its characteristics. For example, an attribute has a type and is readable and/or writeable. The following table lists all MBean metadata classes:

Class Name	**Purpose**
`MBeanInfo`	Top-level object containing arrays of metadata objects for all MBean elements; also includes the name of the MBean’s Java class and a description string
`MBeanFeatureInfo`	Parent class from which all other metadata objects inherit a name and a description string
`MBeanOperationInfo`	Describes an operation: the return type, the signature as an array of parameters, and the impact (whether the operation just returns information or modifies the resource)
`MBeanConstructorInfo`	Describes a constructor by its signature
`MBeanParameterInfo`	Gives the type of a parameter in an operation or constructor signature

Class Name	Purpose
`MBeanAttributeInfo`	Describes an attribute: its type, whether it is readable, and whether it is writeable
`MBeanNotificationInfo`	Contains an array of notification type strings

Implementing a Dynamic MBean

A dynamic MBean consists of a class that implements the `DynamicMBean` interface coherently. By this, we mean a class which exposes a management interface whose description matches the attributes and operations which are accessible through the generic getters, setters and invokers.

Note - MBeans are not allowed to be both standard and dynamic. When a class is instantiated as an MBean, the agent checks the interfaces that it implements. If the class implements or inherits an implementation of *both* the corresponding MBean interface and the DynamicMBean interface, then an exception is raised and the MBean cannot be created.

Beyond this restriction, a dynamic MBean must also follow the same two rules as a standard MBean, namely:

- It must be a concrete class so that it can be instantiated
- It must expose at least one public constructor so that any other class can create an instance

Thereafter, a dynamic MBean class is free to declare any number of public or private methods and variables. None of these are visible to management applications, only the methods implementing the `DynamicMBean` interface are exposed for management. A dynamic MBean is also free to rely on other classes which may be a part of the manageable resource.

Dynamic Programming Issues

An MBean is a manageable resource that exposes a specific management interface. The name *dynamic MBean* refers to the fact that the interface is revealed at runtime, as opposed to through the introspection of static class names. The term *dynamic* is not meant to imply that the MBean can dynamically change its management interface. The management architecture defined by JMX and implemented in the Java Dynamic Management Kit does not support MBeans whose management interface is modified during runtime.

This is not an issue with standard MBeans which would need to be recompiled in order to change their interface. However, dynamic MBeans could be programmed so that their interface description and their generic getters, setters and the invoker have a different behavior at different times. In practice, this type of MBean could be created but it couldn't be managed after any change of interface.

As a rule, the value returned by the `MBeanInfo` method of a dynamic MBean, and the corresponding behavior of getters, setters and the invoker, must never change over the lifetime of a given instance of the MBean. However, it is permissible to have the same dynamic MBean class expose different management interfaces depending upon the instantiation conditions. This would be a valid MBean, since the agent architecture manages object instances, not class types. It would also be a very advanced MBean for a complex management solution, beyond the scope of this tutorial.

The `getMBeanInfo` Method

Since the MBean description should never change, it is usually created once at instantiation, and the `getMBeanInfo` method just returns its reference at every call. The MBean constructor should therefore build the `MBeanInfo` object from the MBean metadata classes such that it accurately describes the management interface. And since most dynamic MBeans will always be instantiated with the same management interface, building the `MBeanInfo` object is fairly straightforward.

The following code shows how the `SimpleDynamic` MBean defines its management interface, as built at instantiation and returned by its `getMBeanInfo` method:

CODE EXAMPLE 2–2 Implemention of the `getMBeanInfo` Method

```
// class constructor
public SimpleDynamic() {

    buildDynamicMBeanInfo();
}

// internal variables describing the MBean
private String dClassName = this.getClass().getName();
private String dDescription = "Simple implementation of a dynamic MBean.";

// internal variables for describing MBean elements
private MBeanAttributeInfo[] dAttributes = new MBeanAttributeInfo[2];
private MBeanConstructorInfo[] dConstructors = new MBeanConstructorInfo[1];
private MBeanOperationInfo[] dOperations = new MBeanOperationInfo[1];
private MBeanInfo dMBeanInfo = null;

// internal method
private void buildDynamicMBeanInfo() {
```

(continued)

```
    dAttributes[0] = new MBeanAttributeInfo(
        "State",                  // name
        "java.lang.String",       // type
        "State: state string.",   // description
        true,                     // readable
        true);                    // writable
    dAttributes[1] = new MBeanAttributeInfo(
        "NbChanges",
        "java.lang.Integer",
        "NbChanges: number of times the State string has been changed.",
        true,
        false);

    // use reflection to get constructor signatures
    Constructor[] constructors = this.getClass().getConstructors();
    dConstructors[0] = new MBeanConstructorInfo(
        "SimpleDynamic(): No-parameter constructor",  //description
        constructors[0]);                   // the contructor object

    MBeanParameterInfo[] params = null;
    dOperations[0] = new MBeanOperationInfo(
        "reset",                      // name
        "Resets State and NbChanges attributes to their initial values",
                                      // description
        params,                       // parameter types
        "void",                       // return type
        MBeanOperationInfo.ACTION);   // impact

    dMBeanInfo = new MBeanInfo(dClassName,
                               dDescription,
                               dAttributes,
                               dConstructors,
                               dOperations,
                               new MBeanNotificationInfo[0]);
}

// exposed method implementing the DynamicMBean.getMBeanInfo interface
public MBeanInfo getMBeanInfo() {

    // return the information we want to expose for management:
    // the dMBeanInfo private field has been built at instantiation time,
    return(dMBeanInfo);
}
```

Generic Attribute Getters and Setters

Generic getters and setters take a parameter that indicates the name of the attribute to read or write. There are two issues to keep in mind when implementing these methods:

- Attribute names must be correctly mapped to their corresponding internal representation
- Invalid attribute names and types should raise an exception, including when writing to a read-only attribute (and vice-versa)

The `getAttribute` method is the simplest, since only the attribute name must be verified:

CODE EXAMPLE 2–3 Implementation of the `getAttribute` Method

```
public Object getAttribute(String attribute_name)
    throws AttributeNotFoundException,
           MBeanException,
           ReflectionException {

    // Check attribute_name to avoid NullPointerException later on
    if (attribute_name == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException("Attribute name cannot be null"),
            "Cannot invoke a getter of " + dClassName +
                " with null attribute name");
    }

    // Call the corresponding getter for a recognized attribute_name
    if (attribute_name.equals("State")) {
        return getState();
    }
    if (attribute_name.equals("NbChanges")) {
        return getNbChanges();
    }

    // If attribute_name has not been recognized
    throw(new AttributeNotFoundException(
        "Cannot find " + attribute_name + " attribute in " + dClassName));
}

// internal methods for getting attributes
public String getState() {
    return state;
}

public Integer getNbChanges() {
    return new Integer(nbChanges);
}

// internal variables representing attributes
private String      state = "initial state";
private int         nbChanges = 0;
```

The `setAttribute` method is more complicated, since you must also insure that the given type can be assigned to the attribute and handle the special case for a null value:

CODE EXAMPLE 2–4 Implementation of the `setAttribute` Method

```
public void setAttribute(Attribute attribute)
    throws AttributeNotFoundException,
           InvalidAttributeValueException,
           MBeanException,
           ReflectionException {

    // Check attribute to avoid NullPointerException later on
    if (attribute == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException("Attribute cannot be null"),
            "Cannot invoke a setter of " + dClassName +
                " with null attribute");
    }
    // Note: Attribute class constructor ensures the name not null
    String name = attribute.getName();
    Object value = attribute.getValue();

    // Call the corresponding setter for a recognized attribute name
    if (name.equals("State")) {
        // if null value, try and see if the setter returns any exception
        if (value == null) {
            try {
                setState( null );
            } catch (Exception e) {
                throw(new InvalidAttributeValueException(
                    "Cannot set attribute "+ name +" to null"));
            }
        }
        // if non null value, make sure it is assignable to the attribute
        else {
            try {
                if ((Class.forName("java.lang.String")).isAssignableFrom(
                        value.getClass())) {
                    setState((String) value);
                }
                else {
                    throw(new InvalidAttributeValueException(
                        "Cannot set attribute "+ name +
                            " to a " + value.getClass().getName() +
                            " object, String expected"));
                }
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
    }

    // optional: recognize an attempt to set a read-only attribute
    else if (name.equals("NbChanges")) {
        throw(new AttributeNotFoundException(
            "Cannot set attribute "+ name +
                " because it is read-only"));
    }

    // unrecognized attribute name
```

(continued)

```
    else {
        throw(new AttributeNotFoundException(
            "Attribute " + name + " not found in " +
                this.getClass().getName()));
    }
}

// internal method for setting attribute
public void setState(String s) {
    state = s;
    nbChanges++;
}
```

Notice that the generic getter and setter methods usually hard-code information about the attributes. If a change in your management solution requires you to change your management interface, it will be harder to do with a dynamic MBean. In a standard MBean, each attribute and operation is a separate method, so unchanged attributes are unaffected. In a dynamic MBean, you must modify these generic methods that encode all attributes.

Bulk Getters and Setters

The `DynamicMBean` interface includes bulk getter and setter methods for reading or writing more than one attribute at once. These methods rely on the following classes:

Class Name	Purpose
`Attribute`	A simple object which contains the name string and value object of any attribute.
`AttributeList`	A dynamically extendable list of `Attribute` objects (extends `java.util.ArrayList`)

The `AttributeList` class extends the `java.util.ArrayList` class which is specific to Java 2. For this class and others that rely on similar sets and collection, the Java Dynamic Management Kit provides the `collections.jar` file for complete compatibility using any JDK version 1.1.*x*. See Directories and Classpath in the preface for more information.

The bulk getter and setter methods usually rely on the generic getter and setter, respectively. This makes them independent of the management interface, which can simplify certain modifications. In this case, their implementation consists mostly of error checking on the list of attributes. However, all bulk getters and setters must implement the following behavior: an error on any one attribute does not interrupt or invalidate the bulk operation on the other attributes.

If an attribute cannot be read, then its name-value pair does not figure in the list of results. If an attribute cannot be written, it will not be copied to the returned list of successful set operations. As a result, if there are any errors, the lists returned by bulk operators will not have the same length as the array or list passed to them. In any case, the bulk operators *do not* guarantee that their returned lists have the same ordering of attributes as the input array or list.

The `SimpleDynamic` MBean shows one way of implementing the bulk getter and setter methods:

CODE EXAMPLE 2–5 Implementation of the Bulk Getter and Setter

```
public AttributeList getAttributes(String[] attributeNames) {

    // Check attributeNames to avoid NullPointerException later on
    if (attributeNames == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException(
                "attributeNames[] cannot be null"),
            "Cannot invoke a getter of " + dClassName);
    }
    AttributeList resultList = new AttributeList();

    // if attributeNames is empty, return an empty result list
    if (attributeNames.length == 0)
            return resultList;

    // build the result attribute list
    for (int i=0 ; i<attributeNames.length ; i++){
        try {
            Object value = getAttribute((String) attributeNames[i]);
            resultList.add(new Attribute(attributeNames[i],value));
        } catch (Exception e) {
            // print debug info but continue processing list
            e.printStackTrace();
        }
    }
    return(resultList);
}

public AttributeList setAttributes(AttributeList attributes) {

    // Check attributesto avoid NullPointerException later on
    if (attributes == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException(
                "AttributeList attributes cannot be null"),
            "Cannot invoke a setter of " + dClassName);
    }
```

(continued)

```
    AttributeList resultList = new AttributeList();

    // if attributeNames is empty, nothing more to do
    if (attributes.isEmpty())
        return resultList;

    // try to set each attribute and add to result list if successful
    for (Iterator i = attributes.iterator(); i.hasNext();) {
        Attribute attr = (Attribute) i.next();
        try {
            setAttribute(attr);
            String name = attr.getName();
            Object value = getAttribute(name);
            resultList.add(new Attribute(name,value));
        } catch(Exception e) {
            // print debug info but keep processing list
            e.printStackTrace();
        }
    }
    return(resultList);
}
```

Generic Operation Invoker

Finally, a dynamic MBean must implement the `invoke` method so that operations in the management interface can be called. This method requires the same considerations as the generic getter and setter:

- Operations and their parameters should be mapped to their internal representation and the result must be returned
- Operation names and parameter types need to be verified
- These verifications are usually hard-coded, again making modifications to the management interface more delicate than in a standard MBean

The implementation in the `SimpleDynamic` MBean is relatively simple due to the one operation with no parameters:

CODE EXAMPLE 2–6 Implementation of the `invoke` Method

```
public Object invoke(
        String operationName, Object params[], String signature[])
    throws MBeanException, ReflectionException {

    // Check operationName to avoid NullPointerException later on
    if (operationName == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException(
                "Operation name cannot be null"),
            "Cannot invoke a null operation in " + dClassName);
    }

    // Call the corresponding operation for a recognized name
    if (operationName.equals("reset")){
        // this code is specific to the internal "reset" method:
        reset();     // no parameters to check
        return null; // and no return value
    } else {
        // unrecognized operation name:
        throw new ReflectionException(
            new NoSuchMethodException(operationName),
            "Cannot find the operation " + operationName +
                " in " + dClassName);
    }
}

// internal variable
private int         nbResets = 0;

// internal method for implementing the reset operation
public void reset() {
    state = "initial state";
    nbChanges = 0;
    nbResets++;
}

// Method not revealed in the MBean description and not accessible
// through "invoke" therefore it is only available for internal mgmt
public Integer getNbResets() {
    return new Integer(nbResets);
}
```

As it is written, the SimpleDynamic MBean correctly provides a description of its management interface and implements its attributes and operations. However, this example demonstrates the need for a strict coherence between what is exposed by the getMBeanInfo method and what can be accessed through the generic getters, setters, and invoker.

A dynamic MBean whose getMBeanInfo method describes an attribute or operation which cannot be accessed is not compliant with the Java Management extensions and is technically not a manageable resource. Similarly, a class could make attributes or operations accessible without describing them in the returned MBeanInfo object. Since MBeans should raise an exception when an undefined attribute or operation is accessed, this would, again, technically not be a compliant resource.

Running the Dynamic MBean Example

The *examplesDir*/`DynamicMBean` directory contains the `SimpleDynamic.java` file which makes up the MBean. The `DynamicMBean` interface is defined in the `javax.management` package provided in the runtime jar file (`jdmkrt.jar`) of the Java Dynamic Management Kit. This directory also contains a simple agent application which instantiates this MBean, calls its `getMBeanInfo` method to get its management interface and manipulates its attributes and operations.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform, you would type:

```
$ cd examplesDir/DynamicMBean/
$ javac -classpath classpath *.java
```

To run the example, launch the agent class which will interact with the `SimpleDynamic` MBean:

```
$ java -classpath classpath DynamicAgent
```

Press `<Enter>` when the application pauses to step through the example. The agent application handles all input and output in this example and gives us a view of the MBean at runtime.

This example demonstrates how the management interface encoded in the `getMBeanInfo` method is made visible in the agent application. We can then see the result of calling the generic getters and setters and the `invoke` method. Finally, the code for filtering attribute and operation errors is exercised, and we see the exceptions from the code samples as they are raised at runtime.

Comparison with the `SimpleStandard` Example

Now that we have implemented both types of MBeans we can compare how they are managed. We purposely created a dynamic MBean and a standard MBean with the same management interface so that we can do exactly the same operations on them. On the Solaris platform, we can compare the relevant code of the two agent applications with the `diff` utility (your output may vary):

```
$ cd examplesDir
$ diff ./StandardMBean/StandardAgent.java ./DynamicMBean/DynamicAgent.java
[...]
41c40
< public class StandardAgent {
---
> public class DynamicAgent {
49c48
<     public StandardAgent() {
---
>     public DynamicAgent() {
77c76
<       StandardAgent agent = new StandardAgent();
---
>       DynamicAgent agent = new DynamicAgent();
88c87
<       echo("\n>>> END of the SimpleStandard example:\n");
---
>       echo("\n>>> END of the SimpleDynamic example:\n");
113c112
<       String mbeanName = "SimpleStandard";
---
>       String mbeanName = "SimpleDynamic";
```

If the two agent classes had the same name, we see that the only programmatic difference would be the following:

```
113c112
<       String mbeanName = "SimpleStandard";
---
>       String mbeanName = "SimpleDynamic";
```

We can see that there is only one difference between the two example agents handling different types of MBeans: the name of the MBean class that is instantiated! In other words, standard and dynamic MBeans are indistinguishable from the agent's point of view. This is the power of the JMX architecture: managers interact with the attributes and operations of a manageable resource, and the specification of the agent hides any implementation differences between MBeans.

Since we know that the two MBeans are being managed identically, we can also compare their runtime behavior. In doing so, we can draw two conclusions:

- The dynamic MBean was programmed to have the same behavior as the standard MBean; the example output shows that this is indeed the case: despite the different implementations, the functionality of the resource is strictly the same
- The only functional difference between the two is that the agent can obtain the self-description strings encoded in the dynamic MBean: attributes and operations are associated with the explanation that the programmer provides for them

Note - There is no mechanism which allows a standard MBean to provide a self-description. The MBean server provides a default description string for each feature in a standard MBean, and these descriptions are identical for all standard MBeans.

Dynamic MBean Execution Time

In the introduction to this topic we presented two structural advantages of dynamic MBeans, namely the ability to wrap existing code to make it manageable and the ability to provide a self-description of the MBean and its features. Another advantage is that using dynamic MBeans can lead to faster overall execution time.

The performance gain depends on the nature of the MBean and how it is managed in the agent. For example, the `SimpleDynamic` MBean, as it is used, is probably not measurably faster than the `SimpleStandard` example in the Chapter 1 topic. When seeking improved performance, there are two situations which must be considered: MBean introspection, and management operations.

Since the dynamic MBean provides its own description, the agent doesn't need to introspect it as it would a standard MBean. Since introspection is done only once by the agent, this is a one-time performance gain during the lifetime of the MBean. In an environment where there are many MBean creations and where MBeans have a short lifetime, a slight performance increase can be measured.

However, the largest performance gain is in the management operations: calling the getters, setters and invoker. As we shall see in the next lesson ("Dynamic Agents"), the agent makes MBeans manageable through generic getters, setters, and invokers. In the case of standard MBeans, the agent must do the computations for resolving attribute and operation names according to the design patterns. Since dynamic MBeans necessarily expose the same generic methods, these are called directly by the agent. When a dynamic MBean has a simple management interface requiring simple programming logic in its generic methods, its implementation can show a better performance than the same functionality in a standard MBean.

CHAPTER 3

Model MBeans

A model MBean is a generic, configurable MBean which applications can use to instrument any resource dynamically. It is a dynamic MBean that has been implemented so that its management interface and its actual resource can be set programmatically. This allows any manager connected to a Java Dynamic Management agent to instantiate and configure a model MBean on the fly.

Model MBeans allow management applications to make resources manageable at runtime. The managing application must provide a compliant management interface for the model MBean to expose. It must also specify the *target objects* that actually implement the resource. Once it is configured, the model MBean will pass any management requests to the target objects and handle the result.

In addition, the model MBean provides a set of mechanisms for how management requests and their results are handled. For example, caching can be performed on attribute values. The management interface of a model MBean is augmented by *descriptors* which contain attributes for controlling these mechanisms.

The code samples in this topic are taken from the files in the `ModelMBean` example directory in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

- “The `RequiredModelMBean` Class” on page 46 gives an overview of model MBeans.
- “Model MBean Metadata” on page 46 explains how we can describe a model MBean’ management interface.
- “The Target Object(s)” on page 48 describes how a model MBean is associated with its resource.
- “Creating the Model MBean” on page 51 shows how to instantiate and register a model MBean.
- “Running the Model MBean Example” on page 52 shows how an agent interacts with a model MBean.

The `RequiredModelMBean` Class

The required Model MBean is mandated by the JMX specification for all compliant implementations. It is a dynamic MBean which lacks any predefined management interface. It contains a generic implementation which will transmit management request on it management interface to the target objects that define its managed resource.

The name of the required model MBean class is the same for all JMX-compliant implementation. Its full package and class name is `javax.management.modelmbean.RequiredModelMBean`. By instantiating this class, any application may use model MBeans.

In order to be useful, the instance of the required model MBean must be given a management interface and the target object of the management resource. In addition, the model MBean metadata must contain descriptors for configuring how the model MBean will respond to management requests. We will cover these steps in subsequent sections.

The MBean server does not make any special distinction for model MBeans. Internally they are treated as the dynamic MBeans that they are, and all of the model MBean's internal mechanisms and configurations are completely transparent to a management application. Like all other managed resources in the MBean server, the resources available through the model MBean can only be accessed through the attributes and operations defined in the management interface.

Model MBean Metadata

The metadata of a any MBean is the description of its management interface. The metadata of the model MBean is described by an instance of the `ModelMBeanInfo` class, which extends the `MBeanInfo` class.

Like all other MBeans, the metadata of a model MBean contains the list of attributes, operations, constructors, and notifications of the management interface. Model MBeans also describe their target object and their policies for accessing the target object. This information is contained in an object called a *descriptor*, defined by the `Descriptor` interface and implemented in the `DescriptorSupport` class.

There is one overall descriptor for a model MBean instance and one descriptor for each element of the management interface, that is for each attribute, operation, constructor, and notification. Descriptors are stored in the metadata object. As defined by the JMX specification all, of the classes for describing elements are extended so that they contain a descriptor. For example, the

ModelMBeanAttributeInfo extends the MBeanAttributeInfo and defines the methods getDescriptor and getDescriptor.

A descriptor is a set of named field and value pairs. Each type of metadata element has a defined set of fields that are mandatory, and users are free to add others. The field names reflect the policies for accessing target objects, and their values determine the behavior. For example the descriptor of an attribute contains the fields currencyTimeLimit and lastUpdatedTimeStamp which are used by the internal caching mechanism when performing a get or set operation.

In this way, model MBeans are manageable as any other MBean, but applications which are aware of model MBeans may interact with the additional features which they provide. The JMX specification defines the names of all required descriptor fields for each of the metadata element, and for the overall descriptor. The field names are also documented in the Javadoc API for the ModelMBean*Info classes.

In our example, our application defines a subroutine to build all descriptors and metadata objects which are needed to define the management interface of the model MBean.

CODE EXAMPLE 3–1 Defining Descriptors and MBeanInfo Objects

```
private void buildModelMBeanInfo(
                 ObjectName inMbeanObjectName, String inMbeanName) {
  try {

    // Create the descriptor and ModelMBeanAttributeInfo
    // for the 1st attribute
    //
    Descriptor stateDesc = new DescriptorSupport();
    stateDesc.setField("name","State");
    stateDesc.setField("descriptorType","attribute");
    stateDesc.setField("displayName","MyState");
    stateDesc.setField("getMethod","getState");
    stateDesc.setField("setMethod","setState");
    stateDesc.setField("currencyTimeLimit","20");

    dAttributes[0] = new ModelMBeanAttributeInfo(
                         "State",
                         "java.lang.String",
                         "State: state string.",
                         true,
                         true,
                         false,
                         stateDesc);

    [...] // create descriptors and ModelMBean*Info for
          // all attributes, operations, constructors
          // and notifications

    // Create the descriptor for the whole MBean
    //
```

(continued)

```
      mmbDesc = new DescriptorSupport( new String[]
                        { ("name="+inMbeanObjectName),
                          "descriptorType=mbean",
                          ("displayName="+inMbeanName),
                          "log=T",
                          "logfile=jmxmain.log",
                          "currencyTimeLimit=5"});

      // Create the ModelMBeanInfo for the whole MBean
      //
      private String dClassName = "TestBean";
      private String dDescription =
         "Simple implementation of a test app Bean.";

      dMBeanInfo = new ModelMBeanInfoSupport(
                           dClassName,
                           dDescription,
                           dAttributes,
                           dConstructors,
                           dOperations,
                           dNotifications);

      dMBeanInfo.setMBeanDescriptor(mmbDesc);

    } catch (Exception e) {
      echo("\nException in buildModelMBeanInfo : " +
            e.getMessage());
      e.printStackTrace();
    }
  }
```

The Target Object(s)

The object instance which actually embodies the behavior of the managed resource is called the target object. The last step of creating a model MBean is to give the MBean skeleton and its defined management interface a reference to the target object. Thereafter, management requests can be handled by model MBean, which will forward them to the target object and handle the response.

The following code example implements the `TestBean` class which is the simple managed resource in our example. Its methods provide the implementation for two attributes and one operation.

CODE EXAMPLE 3–2 Implementing the Managed Resource

```
public class TestBean
   implements java.io.Serializable
{

    // Constructor
    //
    public TestBean() {
        echo("\n\tTestBean Constructor Invoked: State " +
             state + " nbChanges: " + nbChanges +
             " nbResets: " + nbResets);

    }

    // Getter and setter for the "State" attribute
    //
    public String getState() {
        echo("\n\tTestBean: getState invoked: " + state);
        return state;
    }

    public void setState(String s) {
        state = s;
        nbChanges++;
        echo("\n\tTestBean: setState to " + state +
             " nbChanges: " + nbChanges);
    }


    // Getter for the read-only "NbChanges" attribute
    //
    public Integer getNbChanges() {
        echo("\n\tTestBean: getNbChanges invoked: " + nbChanges);
        return new Integer(nbChanges);
    }

    // Method of the "Reset" operation
    //
    public void reset() {
        echo("\n\tTestBean: reset invoked ");
        state = "reset initial state";
        nbChanges = 0;
        nbResets++;
    }

    // Other public method; looks like a getter,
    // but no NbResets attribute is defined in
    // the management interface of the model MBean
    //
    public Integer getNbResets() {
        echo("\n\tTestBean: getNbResets invoked: " + nbResets);
        return new Integer(nbResets);
    }

    // Internals
    //
```

(continued)

(Continuation)

```
    private void echo(String outstr) {
        System.out.println(outstr);
    }

    private String  state = "initial state";
    private int     nbChanges = 0;
    private int     nbResets = 0;

}
```

By default, the model MBean handles a managed resource that is contained in one object instance. This target is specified through the `setManagedResource` method defined by the `ModelMBean` interface. The resource can encompass several programmatic objects because individual attributes or operations can be handled by different target objects. This behavior is configured through the optional `targetObject` and `targetType` descriptor fields of each attribute or operation.

In our example, one of the operations is handled by an instance of the `TestBeanFriend` class. In the definition of this operation's descriptor, we set this instance as the target object. We then create the operation's `ModelMBeanOperationInfo` with this descriptor and add it to the list of operations in the metadata for our model MBean.

CODE EXAMPLE 3–3 Setting Other Target Objects

```
MBeanParameterInfo[] params = null;

Descriptor getNbResetsDesc = new DescriptorSupport(new String[]
                                    { "name=getNbResets",
                                      "class=TestBeanFriend",
                                      "descriptorType=operation",
                                      "role=operation"});

TestBeanFriend tbf = new TestBeanFriend();
getNbResetsDesc.setField("targetObject",tbf);
getNbResetsDesc.setField("targetType","objectReference");

dOperations[1] = new ModelMBeanOperationInfo(
                         "getNbResets",
                "getNbResets(): get number of resets performed",
                         params ,
                         "java.lang.Integer",
                         MBeanOperationInfo.INFO,
                         getNbResetsDesc);
```

(continued)

Creating the Model MBean

In order to insure coherence in an agent application, you should define the target object of an MBean before you expose it for management. This implies that you should call the `setManagedResource` method before registering the model MBean in the MBean server.

The following code example show how our application creates the model MBean. First it calls the subroutine give in Code Example 3–1 to build the descriptors and management interface of our model MBean. Then it instantiates the required model MBean class with this metadata. Finally it creates and sets the managed resource object before registering the model MBean.

CODE EXAMPLE 3–4 Setting the Default Target Object

```
ObjectName mbeanObjectName = null;
String domain = server.getDefaultDomain();
String mbeanName = "ModelSample";

try
{
     mbeanObjectName = new ObjectName(
                              domain + ":type=" + mbeanName);
} catch (MalformedObjectNameException e) {
     e.printStackTrace();
     System.exit(1);
}

// Create the descriptors and ModelMBean*Info objects
// of the management interface
//
ModelMBeanInfo dMBeanInfo = null;
buildModelMBeanInfo( mbeanObjectName, mbeanName );

try {
    // Instantiate javax.management.modelmbean.RequiredModelMBean
    RequiredModelMBean modelmbean =
        new RequiredModelMBean( dMBeanInfo );

    // Associate it with the resource (a TestBean instance)
```

(continued)

```
    modelmbean.setManagedResource( new TestBean(), "objectReference");

    // register the model MBean in the MBean server
    server.registerMBean( modelmbean, mbeanObjectName );

} catch (Exception e) {
     echo("\t!!! ModelAgent: Could not create the model MBean !!!");
     e.printStackTrace();
     System.exit(1);
}
```

Our model MBean is then available for management operations and remote requests, just like any other registered MBean.

Running the Model MBean Example

The *examplesDir*/`ModelMBean` directory contains the `TestBean.java` file which is the target object of the sample model MBean. This directory also contains a simple notification listener class and the agent application, `ModelAgent`, which instantiates, configures and manages a model MBean.

The model MBean itself is given by the `RequiredModelMBean` class defined in the `javax.management.modelmbean` package provided in the runtime jar file (`jdmkrt.jar`) of the Java Dynamic Management Kit.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform, you would type:

```
$ cd examplesDir/ModelMBean/
$ javac -classpath classpath *.java
```

To run the example, launch the agent class with the following command:

```
$ java -classpath classpath ModelAgent
```

Type return when the application pauses to step through the example. The agent application handles all input and output in this example and gives us a view of the MBean at runtime.

We can then see the result of managing the resource through its exposed attributes and operations. The agent also instantiates and registers a listener object for attribute change notifications sent by the model MBean. You can see the output of this listener whenever it receives a notification, after the application has called one of the attribute setters.

PART II Agent Applications

The agent is the central component of the JMX management architecture. An agent contains MBeans and hides their implementation behind a standardized management interface, it lets management applications connect and interact with all MBeans, and it provides filtering for handling large numbers of MBeans. JMX agents are dynamic because resources can be added and removed, connections can be closed and reopened with a different protocol, and services can be added and removed as management needs evolve.

In the previous lesson, “Instrumentation through MBeans”, we saw how to represent resources as MBeans. However, MBeans can represent any object whose functionality you need to manage. In particular, management services and remote connectivity are handled by objects which are also MBeans. This creates a very homogeneous model where an agent is a framework containing MBeans of different sorts and allowing them to interact.

The main component of an agent is the MBean server: it registers all MBeans in the agent and exposes them for management. The role of the MBean server is to be the liaison between any object available to be managed and any object with a management request. Usually resource MBeans are managed either by remote applications through connectivity MBeans or by local management service MBeans. This model allows a management service itself to be managed: connectors and services can also be created, modified or removed dynamically.

This lesson focuses on the functionality of the MBean server and the Java objects which are needed to create a simple agent. Details about programming managers and about using connectors and services will be covered later.

This lesson contains the following topics:

- “The MBean Server in a Minimal Agent” covers the interface of the MBean server which is used by all agents. We introduce the object name of an MBean which is its only reference in the MBean server. The only MBeans in the minimal agent are the communications MBeans, but this is enough to connect to the agent and manage it.

- "The HTML Protocol Adaptor" gives us a management view of the MBeans in an agent through a web browser. It lets us create MBeans, update their attributes, invoke their operations, and remove them dynamically in a running agent.
- "The Base Agent" is similar to the minimal agent but it shows how to manipulate MBeans programmatically through the instance of the MBean server. It covers the different ways of creating and interacting with MBeans in the MBean server. This topic also covers how to process the metadata objects that represent MBean information.
- "The Notification Mechanism" demonstrates the fundamentals of notification broadcasters and listeners where both are within the same agent. Since the MBean server delegate is a broadcaster, the example shows how to register a listener to process its events. The example also shows how to listen for attribute change notifications, a subclass of regular notifications that is defined by the JMX specification.

CHAPTER 4

The MBean Server in a Minimal Agent

An agent application is a program written in the Java language which contains an MBean server and some way of accessing its functionality. This would be a minimal agent, anything less couldn't be managed. In our example of a minimal agent, we will access the MBean server through the HTML protocol adaptor from a web browser.

In a real management solution, the agent could be instantiated and loaded with MBeans for all services and resources that it might need when launched. However, a minimal agent might also be used when resources and services are unknown at launch time. They would be instantiated dynamically at a later date by some management application connected to the agent. This flexibility shows how the Java Dynamic Management Kit lets you develop many different management solutions, depending on your intended deployment strategy.

For now we will focus on the functionality of the MBean server and how to interact with it through the HTML protocol adaptor. The code samples in this chapter are taken from the files in the `MinimalAgent` example directory located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Contents:

MBean Server Classes

Before writing an agent application, it is important to understand the functionality of the MBean server. It is actually an interface and a *factory* object defined by the agent specification level of the Java Management extensions. The Java Dynamic Management Kit provides an implementation of this interface and factory. The factory object finds or creates the MBean server instance, making it possible to substitute different implementations of the MBean server.

The `MBeanServer` Interface

The specification of the interface defines all operations that can be applied to resources and other agent objects through the MBean server. Its methods can be divided into three main groups:

- Methods for controlling MBean instances:
  - `createMBean`, or `instantiate` and `registerMBean` add a new MBean to the agent
  - `unregisterMBean` removes an MBean from the agent
  - `isRegistered` and `getObjectInstance` associate the class name with the MBean's management name
  - `addNotificationListener` and `removeNotificationListener` control event listeners for a particular MBean
  - `deserialize` is used to download new MBean classes
- Methods for accessing MBean attributes and operations; these methods are identical to those presented in "The `DynamicMBean` Interface" on page 30, except they all have an extra parameter for specifying the target MBean:
  - `getMBeanInfo`
  - `getAttribute` and `getAttributes`
  - `setAttribute` and `setAttributes`
  - `invoke`
- Methods for managing the agent as a whole:
  - `getDefaultDomain` (domains are a way of grouping MBeans in the agent)
  - `getMBeanCount` of all MBeans in an agent
  - `queryMBeans` and `queryNames` are used to find MBeans by name or by value

The MBean Server Implementation and Factory

The `MBeanServerImpl` class in the Java Dynamic Management Kit implements the `MBeanServer` interface. It is the class that will be instantiated in an agent. However, it is never instantiated directly by the agent application. Instead, you rely on the MBean server factory to return a new instance of the implementing class.

The `MBeanServerFactory` is a static class defined by the Java Management extensions that makes the agent application independent of the MBean server implementation. It resides in the Java virtual machine and centralizes all MBean server instantiation. It provides two static methods:

- `createMBeanServer` which returns a new MBean server instance
- `findMBeanServer` which returns a list of MBean servers in the Java virtual machine

You must use this class to create an MBean server so that other objects can obtain its reference by calling the `findMBeanServer` method. This method allows dynamically loaded objects to find the MBean server in an agent which has already been launched.

The Minimal Agent

The minimal agent contains only the essential components of a complete agent application. These are:

- An instance of the MBean server
- Some means of communication

The following code is the complete application from the `MinimalAgent.java` file:

CODE EXAMPLE 4–1 The Minimal Agent

```
import javax.management.ObjectInstance;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;

public class MinimalAgent {

    public static void main(String[] args) {

        // Instantiate the MBean server
        System.out.println("\nCreate the MBean server");
        MBeanServer server = MBeanServerFactory.createMBeanServer();

        // Create and start some way of communicating:
        //    - an HTML protocol adaptor
```

(continued)

```
        //    - an HTTP connector server
        //    - an RMI connector server
        // Any single one of these would be sufficient
        try {

            System.out.println(
                "\nCreate and start an HTML protocol adaptor");
            ObjectInstance html = server.createMBean(
                "com.sun.jdmk.comm.HtmlAdaptorServer", null);
            server.invoke(html.getObjectName(), "start",
                          new Object[0], new String[0]);

            System.out.println(
                "\nCreate and start an HTTP connector server");
            ObjectInstance http = server.createMBean(
                "com.sun.jdmk.comm.HttpConnectorServer", null);
            server.invoke(http.getObjectName(), "start",
                          new Object[0], new String[0]);

            System.out.println(
                "\nCreate and start an RMI connector server");
            ObjectInstance rmi = server.createMBean(
                "com.sun.jdmk.comm.RmiConnectorServer", null);
            server.invoke(rmi.getObjectName(), "start",
                          new Object[0], new String[0]);

        } catch(Exception e) {
            e.printStackTrace();
            return;
        }

        System.out.println(
"\nNow, you can point your browser to http://localhost:8082/");
        System.out.println(
"or start your management application to connect to this agent.\n");

    }
}
```

Here we analyze the most important lines of code in this example. We start with the instantiation of the MBean server:

```
MBeanServer server = MBeanServerFactory.createMBeanServer();
```

The MBean server is created through the static `MBeanServerFactory` object, and we store its object reference. Its true type is hidden by the factory object, which casts the returned object as an `MBeanServer` interface. The MBean server is the only functional class referenced directly in this application.

Then we just need some means of communicating (only the HTML adaptor is shown here):

```
ObjectInstance html = server.createMBean("com.sun.jdmk.comm.HtmlAdaptorServer", null);
server.invoke(html.getObjectName(), "start", new Object[0], new String[0]);
```

For each of the communications protocols we ask the MBean server to create the corresponding MBean. We must provide the class name of the MBean which we want the MBean server to instantiate. The MBean server instantiates the object, registers it for management, and returns its `ObjectInstance` reference (see "The `ObjectInstance` of an MBean" on page 63).

When an MBean is created through the MBean server, you can never obtain a direct programmatic reference on an MBean. The object instance is the only handle the caller has on the MBean. The MBean server hides the MBean object instance and only exposes its management interface.

We then ask the server to invoke the `start` operation of the HTML adaptor's MBean. The object instance gives us the MBean's object name which is what we use to identify the MBean (see "Object Names" on page 61). The other parameters correspond to the signature of the `start` operation which takes no parameters. This operation activates the adaptor or connector so that it will respond to remote management operations on its default port. We can now connect to the HTML protocol adaptor from a web browser.

Referencing MBeans

As demonstrated by the minimal agent, most agent applications interact with MBeans through the MBean server. It is possible for an object to instantiate an MBean class itself which it can then register in the MBean server. In this case, it may keep a programmatic reference to the MBean instance. All other objects can only interact with the MBean through its management interface exposed by the MBean server.

In particular, service MBeans and connectivity MBeans rely solely on the MBean server to access resources. The MBean server centralizes the access to all MBeans: it unburdens all other objects from having to keep numerous object references. To insure this function, the MBean server relies on object names to uniquely identify MBean instances.

Object Names

Each MBean object registered in the MBean server is identified by an object name. The same MBean class can have multiple instances, but each must have a unique name. The `ObjectName` class encapsulates an object name which is composed of a domain name and a set of key properties. The object name can be represented as a string in the following format:

DomainName`:`*property*=*value*`[`,*property2*=*value2*`]*`

The *DomainName*, the *properties* and their *values* can be any alpha-numeric string, so long as they don't contain any of the following characters: `: , = * ?`. All elements of the object name are treated as literal strings, meaning that they are case sensitive.

Domains

A domain is an abstract category that can be used to group MBeans arbitrarily. The MBean server lets you easily search for all MBeans with the same domain. For example, all connectivity MBeans in the minimal server could have been registered into a domain called `Communications`.

Since all object names must have a domain, the MBeans in an MBean server necessarily define at least one domain. When the domain name is not important, the MBean server provides a default domain name which you can use. By default, it is called the `DefaultDomain`, but you may specify a different default domain name when creating the MBean server from its factory.

Key Properties

A key is a property-value pair that can also have any meaning that you assign to it. An object name must have at least one key. Keys and their values are independent of the MBean's attributes: the object name is a static identifier which should identify the MBean, whereas attributes are the exposed, runtime values of the corresponding resource. Keys are not positional and can be given in any order to identify an MBean.

Keys usually provide the specificity for identifying a unique MBean instance. For example, a better object name for the HTML protocol adaptor MBean might be: `Communications:protocol=html,port=8082`, assuming the port will not change.

Usage

All MBeans must be given an object name that is unique. It can be assigned by the MBean's preregistration method, if the MBean supports preregistration (see the Javadoc API of the `MBeanRegistration` interface). Or it can be assigned by the object which creates or registers the MBean, which overrides the one given during pre-registration. However, if neither of these assign an object name, the MBean server will not create the MBean and raise an exception. Once an MBean is instantiated and registered, its assigned object name cannot be modified.

You are free to encode any meaning into the domain and key strings, the MBean server just handles them as literal strings. The contents of the object name should be determined by your management needs. Keys could be meaningless serial numbers if MBeans are always handled programmatically. On the other hand, the keys could be human-readable to simplify their translation to the graphical user interface of a management application. With the HTML protocol adaptor, object names are displayed directly to the user.

The `ObjectInstance` of an MBean

An object instance represents the complete reference of an MBean in the MBean server. It contains the MBean's object name and its Java class name. Object instances are returned by the MBean server when an MBean is created or in response to queries about MBeans. Since the object name and class name cannot change over the life of a given MBean, its returned object instance will always have the same value.

You cannot modify the class or object name in an object instance, this information can only be read. The object name is used to refer to the MBean instance in any management operation through the MBean server. The class name may be used to instantiate similar MBeans or introspect characteristics of the class.

A Minimalist Agent

The following code is one of the smaller agents you can write. It retains all of the functionality that we will need to connect to it with a web browser in the next topic ("The HTML Protocol Adaptor").

CODE EXAMPLE 4–2 A Minimalist Agent

```
import javax.management.ObjectInstance;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;

public class MinimalistAgent {

    public static void main(String[] args) {

        MBeanServer server = MBeanServerFactory.createMBeanServer();

        try {

            ObjectInstance html = server.createMBean(
                "com.sun.jdmk.comm.HtmlAdaptorServer", null);
```

(continued)

```
            server.invoke(html.getObjectName(), "start", null, null);

        } catch(Exception e) {
            e.printStackTrace();
            return;
        }
    }
}
```

Note - Only three classes are "imported" by this program and needed to compile it. However, the MBean server dynamically instantiates other classes such as the `HtmlAdaptorServer` which are needed at runtime. As a result, the Java Dynamic Management Kit runtime jar file (`jdmkrt.jar`) must be in the classpath of the Java virtual machine running the minimal agent.

The `MinimalistAgent` relies on the HTML adaptor, but we could have used any MBean that provides some way of accessing the MBean server. You could even use your own MBean that encodes some proprietary protocol, provided it makes all functionality of the MBean server available remotely.

It is important to realize that this minimalist agent is a fully functional agent that is every bit as powerful as any agent that may be deployed. Since we can connect to this agent, we can dynamically create new MBeans for it to manage, and classes that aren't available locally can be downloaded from the network (this is covered in "The M-Let Class Loader"). Because resources, services and other connections may be added on-the-fly, this agent can participate in any management scheme.

Of course, it is more efficient for the agent application to perform the initialization, including the creation of all MBeans that are known to be necessary. Typically, an agent application also needs to set up data structures or launch specific applications that its resources require. For example, it may establish a database session that an MBean will use to expose stored information. The agent application usually includes everything necessary for making the intended resources ready to be managed within the intended management solution.

However, there is no single management solution. Many different agent applications could perform the same management function, requiring more or less intervention after they are launched. And the flexibility of the Java Dynamic Management Kit means that there are many different management solutions to achieve the same goal. For example, an MBean could also establish the database session itself during its preregistration phase.

Running the Minimal Agent Example

The example code for the `MinimalAgent` application is located in the *examplesDir*/ `MinimalAgent` directory. As we saw, this agent application only has minimal output and is intended to be accessed for management through one of its communications MBeans. However, you will need to compile and launch the minimal agent if you continue on to the next topic.

Compile the `MinimalAgent.java` file in this directory with the `javac` command. For example, on the Solaris platform, you would type:

```
$ cd examplesDir/MinimalAgent/
$ javac -classpath classpath *.java
```

When we access the minimal agent through the HTML adaptor, we will instantiate the `SimpleStandard` and `SimpleDynamic` classes. Since we don't use a dynamic class loader, the agent will need these classes at runtime. You will need to have compiled the standard and dynamic MBean classes as described in "Running the Standard MBean Example" on page 28 and "Running the Dynamic MBean Example" on page 41. To run the example, update your classpath to find the MBeans and launch the agent class:

```
$ java -classpath classpath:../StandardMBean:../DynamicMBean MinimalAgent
```

This being a minimal agent, the example doesn't have much output. The MBean server is launched, and the three communication MBeans are created: it is now possible to connect to this agent. Management applications may connect through the HTTP and RMI connector servers, as described in "Connector Servers" on page 110.

The simplest way to communicate with the agent is through the HTML protocol adaptor. This adaptor provides a view of the agent and its MBeans through standard HTML pages which can be viewed on almost any web browser. To connect to the agent, load the following URL in your browser:

```
http://localhost:8082/
```

If you get an error, you may have to switch off proxies in your preference settings or substitute your machine name for `localhost`. Any browser on your local network can also connect to this agent by using your machine name in this URL. In the next topic, "The HTML Protocol Adaptor", we will go into the details of managing this agent from its web view.

CHAPTER 5

The HTML Protocol Adaptor

The HTML protocol adaptor provides a view of the agent and its registered MBeans through a basic interface on any web browser. It is the easiest way to access an agent since no further coding is necessary. For this reason, it can be useful for testing and debugging your MBeans in the minimal agent.

In fact, we will use your browser to “manage” the minimal agent and its MBeans. The HTML protocol adaptor outputs HTML pages which represent the agent and its MBeans. The adaptor also interprets the commands sent back by the buttons and fields appearing in your browser. It then interacts with the agent’s MBean server in order to get information about the MBeans that it has registered and operate on them.

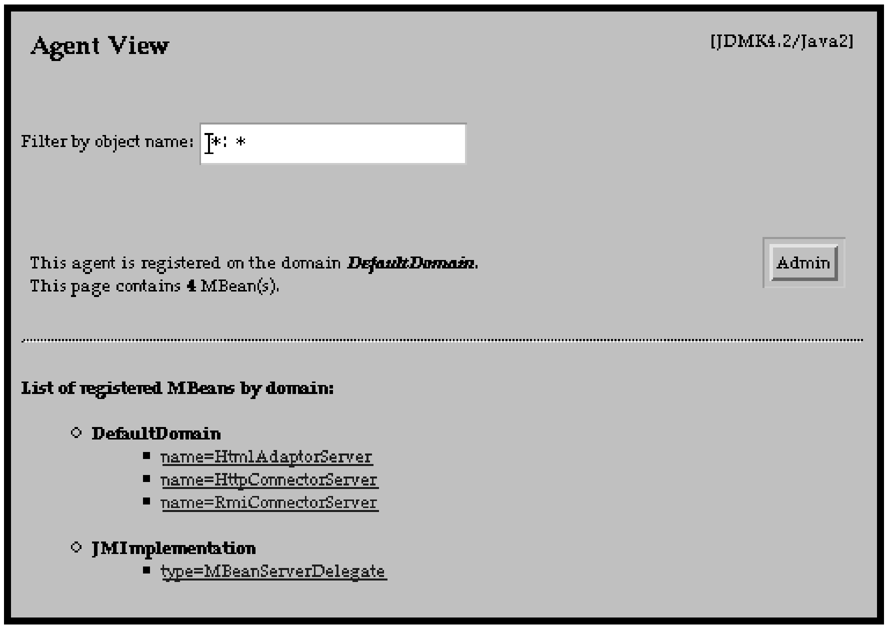
The HTML adaptor relies mostly on plain HTML. The only JavaScript™ that the generated pages contain are pop-up windows for displaying information. Browsers that are not JavaScript enabled might give an incompatibility message and won’t be able to display the information. Otherwise, the generated pages contain no further scripting (JavaScript, Visual Basic or other), no frames and no images that might slow down loading.

This topic relies on the minimal agent which you will need to launch first, as explained in “Running the Minimal Agent Example” on page 65. Once you can connect to the HTML protocol adaptor in the minimal agent, you are ready to go through these topics:

- “The Agent View” on page 68 is the main page for managing an agent through the HTML protocol adaptor.
- “The MBean View” on page 70 exposes an MBean’s management interface.
- The “Agent Administration” on page 74 lets you instantiate new MBeans.
- “Instantiating and Managing MBeans” on page 75 shows how to modify attributes and invoke operations.
- “Filtering MBeans” on page 78 is used to select the MBeans displayed in the agent view.

The Agent View

The first page displayed by the HTML adaptor is always the agent view. It originally contains a list of all registered MBeans. The following figure shows the agent view for the minimal agent. It contains four MBeans: the three communication MBeans, one of which is the HTML adaptor, and the MBean server delegate. The delegate is a special MBean covered in "The MBean Server Delegate" on page 69.


Agent View [JDMK4.2/Java2]

Filter by object name: *:*

This agent is registered on the domain ***DefaultDomain***.
This page contains **4** MBean(s).

List of registered MBeans by domain:

- **DefaultDomain**
  - name=HtmlAdaptorServer
  - name=HttpConnectorServer
  - name=RmiConnectorServer
- **JMImplementation**
  - type=MBeanServerDelegate

Figure 5–1 Initial Agent View of the Minimal Agent

The text field for filtering by object name lets you modify the list of displayed MBeans. The filter string is initially `*:*`, which gives all registered MBeans. Further use of the filter is covered in "Filtering MBeans" on page 78. The agent's registered domain tells you the name of the default domain in this agent. The number of MBeans on this page is the count of those listed beneath the separator line.

The "Admin" button is a link to the agent administration page which we will cover in "Agent Administration" on page 74.

The MBean List

The MBean list contains all MBeans whose object name matches the filter string. Object names can be filtered by their domain name and list of key-value pairs. In this list, MBeans are sorted and grouped by their domain name. Each MBean name listed is an active link to the page of the corresponding MBean view.

After its initialization, the contents of an agent are dynamic: new MBeans may be created and registered into new or existing domains and old MBeans may be removed. These changes can also affect the functionality of the agent: new agent services may be registered (or removed) as well. We will demonstrate examples of dynamic management in “Instantiating and Managing MBeans” on page 75.

The MBean Server Delegate

The MBean server delegate is an MBean that is automatically instantiated and registered by the MBean server when it is created. It provides information about the version of the Java Dynamic Management Kit which is running, and it represents the MBean server when sending notifications.

Notifications are events sent by MBeans, they are covered in detail in the lesson on “The Notification Mechanism.” Since the MBean server instance is not an MBean object, it relies on its delegate MBean to send notifications. The MBean server delegate sends notifications to inform interested listeners about such events as MBean registrations and de-registrations.

The exposed attributes of the delegate MBean provide vendor and version information about the MBean server. This can let a remote management application know which agent version is running and which version of the Java Runtime Environment it is using. The delegate MBean also provides a unique identification string for its MBean server.

▼ Viewing the MBean Server Delegate Information

1. **Click on the name of the delegate MBean to see its attributes. Version, vendor and identification information is listed in the table of attributes.**

2. **Click on the** `Back to Agent View` **link or use your browser’s *Previous page* function to return to the MBean list in the agent view.**

The MBean View

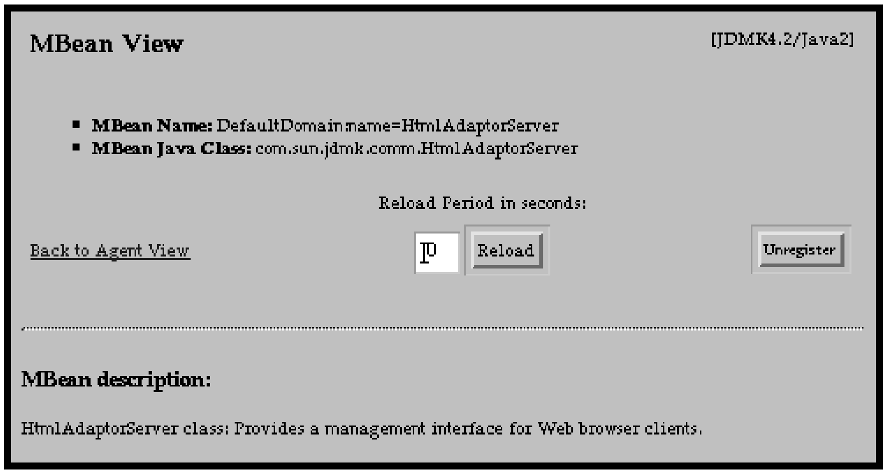
The MBean view has two functions: it presents the management interface of the MBean and it lets you interact with its instance. The management interface of an MBean is given through the name of the attributes, the operation signatures, and a self-description. You may interact with the MBean by reloading its attribute values, setting new values or invoking an operation.

▼ Preparation

1. **In the agent view, click on the object name of the HTML adaptor MBean:** `name=HTMLAdaptorServer` **in the default domain. This will bring up its MBean view.**

The Header and Description

As shown in the following figure, the top part of the page contains the description of the MBean and some controls for managing it:



MBean View [JDMK4.2/Java2]

- **MBean Name:** DefaultDomain:name=HtmlAdaptorServer
- **MBean Java Class:** com.sun.jdmk.comm.HtmlAdaptorServer

Reload Period in seconds: 0

MBean description:

HtmlAdaptorServer class: Provides a management interface for Web browser clients.

Figure 5–2 Description in the MBean View

The first two lines give the object instance (object name and class name) for this MBean. The MBean name is the full object name of this MBean instance, including the domain. The key-property pairs may or may not identify the MBean to a human reader, depending on the agent developer's intention. The MBean Java class is the full class name for the Java object of which this MBean is an instance.

The reload controls include a text field for entering a reload period and a manual "Reload" button. Originally, the reload period is set to zero indicating that the contents of the MBean view are not automatically refreshed. Clicking the reload button will force the page to reload, thereby updating all of the attribute values displayed. If you have entered a reload period, clicking the button will begin automatic reloading with the given period. The reload period must be at least five seconds.

Note - You should use the reload button of the MBean view instead of the browser's reload-page button. After some operations, such as applying changes to attribute values, the browser's button will repost the form data, inadvertently performing the same operation again. To avoid undesirable side effects, always use the reload button provided in the MBean view.

▼ Setting the Reload Period

1. **Enter a reload period of five and click the "Reload" button. Every five seconds the page will blink as it reloads.**

2. **In another browser window, open another connection to the HTML adaptor at** `http://localhost:8082/`**. Observe the new values for the** `ActiveClientCount` **and** `LastConnectedClient` **attributes in the original window. Due to the way the adaptor works, you may have to try several connections before you see the attribute values change.**

The reload period is reset to zero every time you open an MBean view.

The "Unregister" button is a shortcut for removing this MBean from the agent. Unregistering is covered in "Instantiating and Managing MBeans" on page 75.

The MBean description text provides some information about the MBean. Because standard MBeans are statically defined, they cannot describe themselves, and the MBean server provides a generic text. Dynamic MBeans are required to provide their own description string at runtime according to the JMX specification. Except for the class name, this is the only way to tell standard and dynamic MBeans apart in the MBean view.

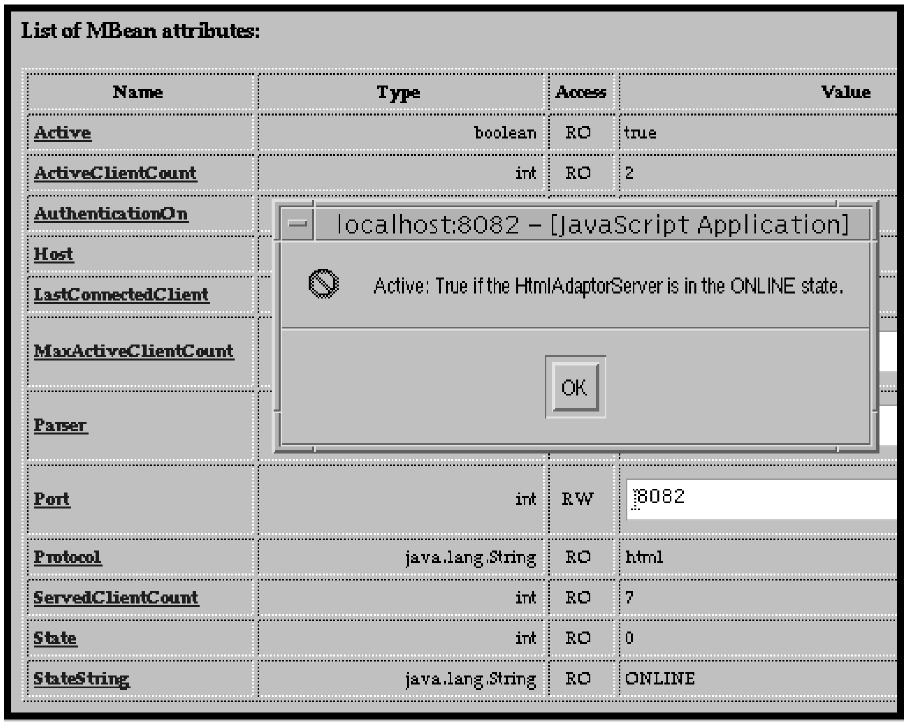
The Table of Attributes

The second part of the MBean view is a table containing all attributes exposed by the MBean. For each attribute, this table lists its name, its Java type, its read-write access and a string representation of its current value.

While MBean attributes may be of any type, not all types can be displayed in the MBean view. The HTML adaptor is limited to basic data types that can be displayed and entered as strings. Read-only attributes whose type support the `toString` method are also displayed. Enumerated types that are concrete subclasses of `com.sun.jdmk.Enumerated` are displayed as a menu with a pop-up selection list. Boolean attributes are represented as true-false radio buttons. Finally, attributes with array types are represented by a link to a page which displays the array values in a table. If the attribute is writeable, you may enter values for the array elements to set them.

For the complete list of supported types, see the Javadoc API of the `HtmlAdaptorServer` class. If an attribute type is not supported, this is mentioned in place of its value. If there was an error when reading an attribute's value, the table contains the name of the exception that was raised and the message it contains.

The name of each attribute is a link that pops up a dialog box containing the description for this attribute. Like the MBean description, attribute descriptions can only be provided by dynamic MBeans. The MBean server inserts a generic message for standard MBean attributes. The following figure shows the attributes of the HTML adaptor with a description of the `Active` attribute:



List of MBean attributes:

Name	Type	Access	Value
Active	boolean	RO	true
ActiveClientCount	int	RO	2
AuthenticationOn			
Host			
LastConnectedClient			
MaxActiveClientCount			
Parser			
Port	int	RW	8082
Protocol	java.lang.String	RO	html
ServedClientCount	int	RO	7
State	int	RO	0
StateString	java.lang.String	RO	ONLINE

localhost:8082 – [JavaScript Application]

Active: True if the HtmlAdaptorServer is in the ONLINE state.

Figure 5–3 MBean Attributes with a Description Dialog

▼ Viewing Attribute Descriptions

1. **Click on the attribute names of the HTML adaptor to read their description. Since the HTML adaptor is implemented as a dynamic MBean, its attribute descriptions are meaningful.**

Note - Due to the use of JavaScript commands in the generated HTML, these pop-up windows might not be available on browsers that are not JavaScript-enabled.

Writable attributes have a text field for entering new values. To set the value of a writable attribute, you would enter or replace its current value in the text field and click the "Apply" button at the bottom of the attributes table.

You should not try to modify the attributes of the HTML protocol adaptor here, we will see why in "Instantiating and Managing MBeans" on page 75.

Because there is only one "Apply" button for all attributes, this button has a particular behavior: it systematically invokes the setter for all writeable attributes, whether or not their field has actually been modified. This may impact the MBean if setters have side effects, such as counting the number of modifications as in the `SimpleStandard` and `SimpleDynamic` examples given in "Instrumentation through MBeans."

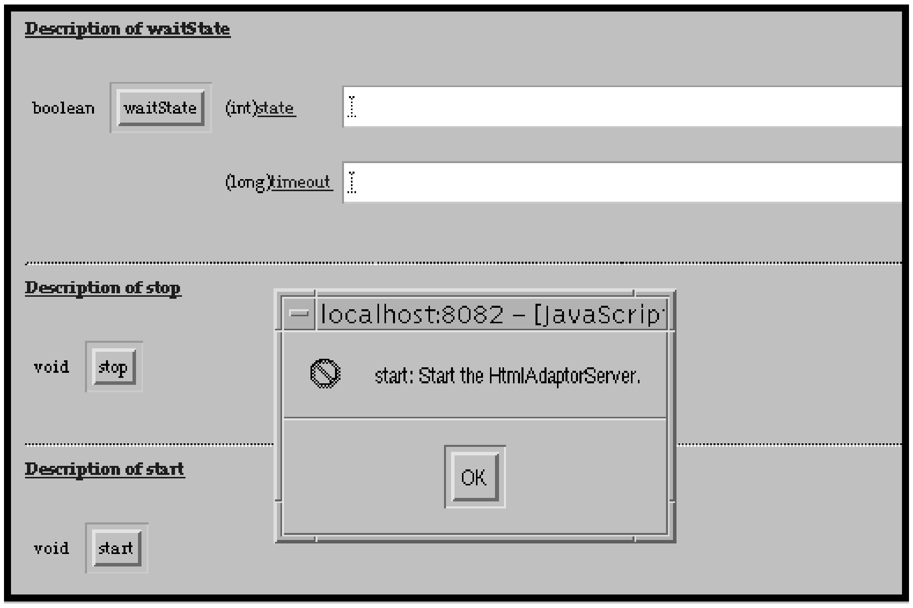
The HTML adaptor detects attributes which are `ObjectName` types and provides a link to the MBean view of the corresponding MBean. This link is labeled `view`, and it is located just under the displayed value of the object name. Since MBeans often need to reference other MBeans, this provides a quick way of navigating through MBean hierarchies.

The Operations

The last part of the MBean view contains all of the operations exposed by the MBean. Each operation in the list is presented like a method signature: there is a return type, then a button with the operation name, and if applicable, a list of parameters, each with their type as well.

As with the table of attributes, the list of operations contains only those involving types that can be represented as strings. The return type must support the `toString` method and the type of each parameter must be one of basic data types supported by the HTML adaptor. For the complete list, see the Javadoc API of the `HtmlAdaptorServer` class.

Above each operation is a link to its description. Parameter names are also active links which pop up a window with a description text. Again, descriptions are only meaningful when provided by dynamic MBeans. The following figure shows some of the operations exposed by the HTML adaptor MBean and a description of the `Start` operation.



start: Start the HtmlAdaptorServer.

Figure 5–4 MBean Operations with a Description Dialog (Partial View)

We will not invoke any operations on this MBean until a brief explanation in “Instantiating and Managing MBeans” on page 75.

To invoke an operation, you would fill in any and all parameter values in the corresponding text fields and then click the operation’s button. The HTML adaptor would then display a page with the result of the operation: the return value if successful or the reason the operation was unsuccessful.

Agent Administration

The agent administration page contains a form for entering MBean information when creating or unregistering MBeans. You may also instantiate an MBean through one of its public constructors. In order to instantiate an MBean, its class must be available in the agent application’s classpath. Optionally, you may specify a different class loader if the agent contains other class loader MBeans.

▼ Preparation

1. **Go back to the agent view by clicking the link near the top of the MBean view page. Then click on the “Admin” button in the agent view to bring up the agent administration page in your browser window.**

The first two fields, “Domain” and “Keys” are mandatory for all administrative actions. The “Domain” field initially contains the string representing the agent’s default domain. Together, these fields define the object name, whether for a new MBean to be created or the name of an existing MBean to unregister. The “Java Class” is the full class name of the object to be instantiated as a new MBean. This field is ignored when unregistering an MBean.

Using the drop-down menu, you may select one of three actions on this page:

- Create - Instantiates the given Java class of an MBean and registers the new instance in the MBean server. If successful, the MBean will then appear in the agent view. The class must have a public constructor without parameters in order to be created in this way.
- Unregister - Unregisters an MBean from the MBean server so that it is no longer available in the agent. The class instance is not explicitly deleted, though if no other references to it exist, it will be garbage collected.
- Constructors - Displays the list of public constructors at the bottom of the administration page for the given Java class. This lets you provide parameters to a specific constructor and create the MBean in this manner. This is the only way to create MBeans which do not have a no-parameter constructor.

When you click the “Send Request” button, the HTML adaptor processes the action and updates the bottom of the page with the action results. You may have to scroll down to see the result. The text fields are not cleared after a request so that you can do multiple operations. The “Reset” button will return the fields to their last posted value after you have modified them.

Instantiating and Managing MBeans

Sometimes, launching an MBean requires several steps: this is particularly the case for agent services which require some sort of configuration. For example, you can instantiate another HTML adaptor for connecting to a different port. Usually, this would be done programmatically in the agent application, but we need to do it through the browser for the minimal agent.

▼ Creating a New HTML Adaptor MBean

1. **On the agent administration page, fill in the fields as follows:**

Domain:	**`Communications`**
Keys:	**`protocol=html,port=8088`**
Java Class:	**`com.sun.jdmk.comm.HtmlAdaptorServer`**
Class Loader:	*leave blank*

   Note: In previous versions of product, specifying the port number in the object name would initialize communication MBeans. Starting with the Java Dynamic Management Kit 4.0, the names and contents of key properties no longer have any significance for any components of the product. We must set the port in other ways.

2. **Make sure the selected action is "Create" and send the request. If you scroll down the page, you should see if your request was successful.**

   We can't connect to this HTML adaptor quite yet, we need to configure it first.

3. **Go to the new HTML adaptor's MBean view with the provided link.**

   We couldn't modify any of the adaptor's attributes before because the implementation is designed so that they can't be modified while it is online. Our new HTML adaptor is instantiated in the stopped state (the `StateString` attribute indicates "OFFLINE"), so we can change its attributes.

4. **Set the `Port` attribute to "8088" and `MaxActiveClientCount` to "2", then click the "Apply" button. If the page is reloaded and the new values are displayed, the attribute write operation was successful. You may also click the attribute names to get an explanation for them.**

5. **Scroll down the MBean view to the `Start` operation and click its button. This brings up a new page to tell us the operation was successful. If you go back to the MBean view with the provided link, you can see that the `StateString` is now indicating `ONLINE`.**

6. **Now you should be able to access your minimal agent through a browser on port 8088. Try going to a different machine on the same network and connecting to the URL:**

   `http://`*agentHostName*`:8088/`

where *agentHostName* is the name or IP address of the machine where you launched the `MinimalAgent`. If you reload the MBean view of the new HTML adaptor on either browser, the name of this other machine should be the new value of the `LastConnectedClient` attribute.

Through this other connection, you could stop, modify or remove the HTML adaptor MBean using port 8082. In that case, your original browser will have to use `http://localhost:8088/` as well to connect. Instead, we will manage the minimal agent from this other machine.

▼ Instantiating MBeans with Constructors

1. **From the browser on the other machine, go to the administration page. Fill in the fields as follows and request the constructors:**

`Domain:`	**`Standard_MBeans`**
`Keys:`	**`name=SimpleStandard,number=1`**
`Java Class:`	**`SimpleStandard`**
`Class Loader:`	*leave blank*

   The list of constructors for the `SimpleStandard` class is given at the bottom of the page. The MBean name is also given: this is the object name that will be assigned to the MBean when using one of the listed constructors. As you can see, the `SimpleStandard` class only has one constructor that takes no parameters.

2. **Click on the "Create" button: the result will be appended to the bottom of the page. Scroll down and go to the MBean view with the provided link.**

   Since it is a standard MBean, all of its description strings are generic: this shows the necessity of programming meaningful attribute names.

3. **In the agent view on the original browser window, click in the filter field and hit "Return" to refresh the agent view. Click on the new MBean's name and set its reload period to 15. Back on the other machine, type in a different string for the `State` attribute and click "Apply".**

   On the original machine, you should see the MBean's attributes get updated when the MBean view is periodically reloaded.

4. **On the other machine, click the `reset` operation button at the bottom of the MBean view page. This brings up the operation result page which indicates the success of the operation.**

This page also gives the return value of the operation when it is not void. If you go back to the MBean view, you will see the result of the operation on the attributes. You should also see it on the original machine after it reloads.

The browser on the other machine is no longer needed, and we can remove the HTML adaptor on port 8088.

▼ Unregistering MBeans

1. **Go to the administration page and fill in the object name of the HTML adaptor we want to remove (you don't need its Java class to unregister it):**

`Domain:`	**`Communications`**
`Keys:`	**`protocol=html,port=8088`**
`Java Class:`	*leave blank*
`Class Loader:`	*leave blank*

2. **Select "Unregister" from the drop down menu and click the "Send Request" button. The result then appears at the bottom of the page.**

   You can also unregister an MBean directly from its MBean view: just click the "Unregister" button on the upper right hand-side of the page.

Filtering MBeans

Since an agent can manage hundreds of MBeans, the agent view provides a filtering mechanism for the list that is displayed. An object name with wildcard characters is used as the filter, and only those MBeans which match are counted and displayed.

Filters restrict the set of MBeans listed in the agent view. This may not be particularly useful for our small agent, but it can help you find MBeans among hundreds in a complex agent. In addition, management applications use the same filter syntax when requesting an agent's MBeans through the programmatic interface of a connector. The filtering lets managers get either lists of MBean names or find a particular MBean instance.

Filters are entered as partial object names with wildcard characters or as a full object name for which to search. Here are the basic substitution rules for filtering:

1. You may search for partial *domain* names:
   the asterisk (`*`) stands for any number (including zero) of any characters;
   the question mark (`?`) stands for any one character

2. An empty domain name stands for the default domain string; an empty key list is illegal
3. Keys are atomic: you must search for the full `property=value` key, you may not search for a partial property name or an incomplete value
4. The asterisk (`*`) may be used to terminate the key list, where it stands for any number of any keys (complete property-value pairs)
5. You must match all keys exactly: use the form `property=value,*` to search for one key in names with multiple keys
6. Keys are unordered when filtering: giving one or more keys (and an asterisk) in any order finds all object names which contain that subset of keys

▼ Instructions

1. **Go to the administration page, and create three more of the standard MBeans. Modify only the `number` value in their object name so that they are numbered sequentially. In the same way, create four dynamic MBeans starting with:**

`Domain:`	**`Dynamic_MBeans`**
`Keys:`	**`name=SimpleDynamic,number=1`**
`Java Class:`	**`SimpleDynamic`**
`Class Loader:`	*leave blank*

2. **Go back to the agent view which should display all of the new MBeans.**

3. **Enter the following filter strings to see the resulting MBean list:**

`Standard_MBeans:*`	Gives all of the standard MBeans we created
`*_MBeans:*`	Gives all of the standard and dynamic MBeans we created
`DefaultDomain:`	Not allowed by rule 2
`:*`	Lists all MBeans in the default domain
`*:name=Simple*,*`	Not allowed by rule 3
`*:name=SimpleStandard`	Allowed, but list is empty (rule 5)
`*:name=*`	Not allowed by rule 3
`*_??????:number=2,*`	Gives the second standard and dynamic MBean we created

`Communications:port=8088,protocol=html`	Gives the one MBean matching the domain and both (unordered) keys
empty string	allowed: special case equivalent to `*:*`

Notice how the MBean count is updated with each filter: this count gives the number of MBeans that were found with the current filter, which is the number of MBeans appearing on the page. It is not the total number of MBeans in the agent, unless the filter is `*:*`.

4. **When you are ready to stop the minimal agent example, go to the window where you launched its class and type `<Control-C>`.**

CHAPTER 6

The Base Agent

The base agent demonstrates the programming details of writing an agent application. We will cover how to access the MBean server, ask it to create MBeans, and then interact with those MBeans. Everything that we could do through the HTML adaptor can be done through the code of your agent application.

Interacting programmatically with the MBean server gives you more flexibility in your agent application. This topic covers the three main ways to create MBeans through the MBean server, how to interact with MBeans, and how to unregister them.

The base agent is functionally equivalent to the minimal agent, but instead of writing the smallest agent, we will demonstrate good defensive programming with error checking. We will create the same three connectivity MBeans and do some of the same management operations that we did through the browser interface.

The program listings in this tutorial show only functional code: comments and output statements have been modified or removed for space considerations. However, all management functionality has been retained for the various demonstrations. The complete source code is available in the `BaseAgent` and `StandardMBean` example directories located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Contents:

- "Managing MBeans" on page 88 demonstrates the same management operations that we did through the HTML protocol adaptor.
- "Filtering MBeans" on page 89 shows how to get various lists of MBeans from the MBean server.
- "Running the Base Agent Example" on page 91 demonstrates its runtime behavior.

The Agent Application

The base agent is a stand-alone application with a `main` method, but it also has a constructor so that it may be instantiated dynamically by another class.

CODE EXAMPLE 6–1 Constructor for the Base Agent

```
public BaseAgent() {
    //
 Enable the TRACE level according to the level set in system properties
    try {
        Trace.parseTraceProperties();
    }
    catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }

    echo("\n\tInstantiating the MBean server of this agent...");
    myMBeanServer = MBeanServerFactory.createMBeanServer();

    // Retrieves ID of the MBean server from the delegate
    try {
        System.out.println("ID = "+ myMBeanServer.getAttribute(
            new ObjectName(ServiceName.DELEGATE), "MBeanServerId"));
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

In the first statement of the constructor, we enable tracing messages for the agent. The tracing lets us see internal information at runtime, which is useful for debugging. See "Setting Trace Messages" on page 91 for specifying tracing parameters on the command line. Then we create the MBean server through its static factory class (see "The MBean Server Implementation and Factory" on page 59). Its reference is stored in a variable with class-wide scope so that all internal methods have access to the MBean server. Finally, we retrieve the MBean server identification string, for informational purposes only.

After the MBean server is initialized, we are going to create the same three communications MBeans that we saw in the minimal agent.

Creating an MBean (Method 1)

The methods of the MBean server let you create an MBean in three different ways. The base agent demonstrates all three ways, and we will discuss the advantages of each.

One way of creating an MBean consists of first instantiating its class and then registering this instance in the MBean server. Registration is the internal process of the MBean server which takes a manageable resource's MBean instance and exposes it for management.

Bold text in this and the following code samples highlights the important statements that vary between the three methods.

CODE EXAMPLE 6–2 Creating an MBean (Method 1)

```
// instantiate the HTML protocol adaptor object to use the default port
HtmlAdaptorServer htmlAdaptor = new HtmlAdaptorServer();

try {
    // We know that the HTML adaptor provides a default object name
    ObjectInstance htmlAdaptorInstance =
        myMBeanServer.registerMBean(htmlAdaptor, null);
    echo("CLASS NAME  = " + htmlAdaptorInstance.getClassName());
    echo("OBJECT NAME = " + htmlAdaptorInstance.getObjectName().toString());
} catch(Exception e) {
    e.printStackTrace();
    System.exit(0);
}

// Now we need to explicitly start the html protocol adaptor
htmlAdaptor.start();

while (htmlAdaptor.getState() == CommunicatorServer.STARTING) {
    sleep(1000);
}
echo("STATE = " + htmlAdaptor.getStateString());
[...]
```

In this first case, we instantiate the `HtmlAdaptorServer` class and keep a reference to this object. We then pass it to the `registerMBean` method of the MBean server to make our instance manageable in the agent. During the registration, the instance can also obtain a reference to the MBean server, something it requires to function as a protocol adaptor.

In the minimal agent, we saw that the HTML adaptor gives itself a default name in the default domain. Its Javadoc API confirms this, so we can safely let it provide a default name. We print the object name in the object instance returned by the registration to confirm that the default was used.

Once the MBean is registered, we can perform management operations on it. Because we kept a reference to the instance, we don't need to go through the MBean server to manage this MBean. This lets us call the `start` and `getStateString` methods directly. The fact that these methods are publicly exposed is particular to the implementation: the HTML adaptor is a dynamic MBean, so without any prior knowledge of the class, we would have to go through its `DynamicMBean` interface.

In a standard MBean you would directly call the implementation of its management interface. Since the HTML adaptor is a dynamic MBean, the `start` method is just a shortcut for the `start` operation. For example, we could start the adaptor and get its `StateString` attribute with the following calls:

```
htmlAdaptor.invoke("start", new Object[0], new String[0]);
echo("STATE = " + (String)htmlAdaptor.getAttribute("StateString"));
```

This type of shortcut is not specified by the Java Management extensions, nor is its functionality necessarily identical to that of the `start` operation exposed for management. In the case of the HTML adaptor, its Javadoc API confirms that it is identical, and in other cases, it is up to the MBean programmer to guarantee this functionality if it is offered.

Creating an MBean (Method 2)

The second way to create an MBean is the single `createMBean` method of the MBean server. In this case, the MBean server instantiates the class and registers it all at once. As a result, the caller never has a direct reference to the new object.

CODE EXAMPLE 6–3 Creating an MBean (Method 2)

```
ObjectInstance httpConnectorInstance = null;
try {
    String httpConnectorClassName = "com.sun.jdmk.comm.HttpConnectorServer";
    // Let the HTTP connector server provides its default name
    httpConnectorInstance =
```

(continued)

(Continuation)

```
        myMBeanServer.createMBean(httpConnectorClassName, null);
} catch(Exception e) {
    e.printStackTrace();
    System.exit(0);
}
// We need the object name to refer to our MBean
ObjectName httpConnectorName = httpConnectorInstance.getObjectName();
echo("CLASS NAME  = " + httpConnectorInstance.getClassName());
echo("OBJECT NAME = " + httpConnectorName.toString());

// Now we demonstrate the bulk getter of the MBean server
try {
    String att1 = "Protocol";
    String att2 = "Port";
    String attNames[]= {att1, att2};
    AttributeList attList =
        myMBeanServer.getAttributes(httpConnectorName, attNames);
    Iterator attValues = attList.iterator();
    echo("\t" + att1 + "\t=" + ((Attribute) attValues.next()).getValue());
    echo("\t" + att2 + "\t=" + ((Attribute) attValues.next()).getValue());

} catch (Exception e) {
    e.printStackTrace();
    System.exit(0);
}

// Now we explicitly start the HTTP connector server
try {
    myMBeanServer.invoke(httpConnectorName, "start",
                         new Object[0], new String[0]);

    // waiting to leave starting state...
    while (new Integer(CommunicatorServer.STARTING).equals
           (myMBeanServer.getAttribute(httpConnectorName,"State"))) {
        sleep(1000);
    }
    echo("STATE = " +
         myMBeanServer.getAttribute(httpConnectorName, "StateString"));
} catch (Exception e) {
    e.printStackTrace();
    System.exit(0);
}
[...]
```

The advantage of this method for creating MBeans is that the instantiation and registration are done in one call. In addition, if we have registered any class loaders in the MBean server, they will automatically be used if the class is not available locally. See “The M-Let Class Loader” for more information on class loading.

The major difference is that we no longer have a reference to our MBean instance. The object instance that was only used for display purposes in the previous example now gives us the only reference we have on the MBean: its object name.

What can be seen as a drawback of this method is that all management of the new MBean must now be done through the MBean server. For the attributes of the MBean, we need to call the generic getter and setter of the MBean server, and for the operations we need to call the `invoke` method. When the agent needs to access the MBean, having to go through the MBean server adds some complexity to the code. It does have the advantage of not relying on any shortcuts provided by the MBean, making the code more portable and reusable.

However, the `createMBean` method is ideal for quickly launching new MBeans that the agent application doesn't need to manipulate. In just one call, the new objects are instantiated and exposed for management.

Creating an MBean (Method 3)

The last way of creating an MBean relies on the `instantiate` method of the MBean server. In addition, we use a non-default constructor to instantiate the class with a different behavior.

CODE EXAMPLE 6–4 Creating an MBean (Method 3)

```
CommunicatorServer rmiConnector = null;
Object[] params = {new Integer(8086)};
String[] signature = {new String("int")};
try {
    String RmiConnectorClassName = "com.sun.jdmk.comm.RmiConnectorServer";
    // specify the RMI port number to use as a parameter to the constructor
    rmiConnector = (CommunicatorServer)myMBeanServer.instantiate(
                       RmiConnectorClassName, params, signature);
} catch(Exception e) {
    e.printStackTrace();
    System.exit(0);
}

try {
    // Let the RMI connector server provides its default name
    ObjectInstance rmiConnectorInstance =
        myMBeanServer.registerMBean(rmiConnector, null);

    // Confirm the class and default object name
    echo("CLASS NAME  = " + rmiConnectorInstance.getClassName());
    echo("OBJECT NAME = " + rmiConnectorInstance.getObjectName().toString());
} catch(Exception e) {
    e.printStackTrace();
```

(continued)

```
    System.exit(0);
}

// Now we explicitly start the RMI connector server
rmiConnector.start();

// waiting to leave starting state...
while (rmiConnector.getState() == CommunicatorServer.STARTING) {
    sleep(1000);
}
echo("STATE = " + rmiConnector.getStateString());

// Check that the RMI connector server is started
if (rmiConnector.getState() != CommunicatorServer.ONLINE) {
    echo("Cannot start the RMI connector server");
    System.exit(0);
}
[...]
```

As in the first example of MBean creation, we instantiate and register the MBean in separate steps. First, we instantiate the class using the `instantiate` method of the MBean server. This method lets you specify the constructor you wish to use when instantiating. Note that we could also have specified a constructor to the `createMBean` method in the previous example.

To specify a constructor, you give an array of objects for the parameters and an array of strings which defines the signature. If these arrays are empty or null, the MBean server will try to use the default no-parameter constructor. If the class does not have a public no-parameter constructor, you must specify the parameters and signature of a valid, public constructor.

In our case, we specify an integer parameter to set the port through one of the constructors of the `RmiConnectorServer` class. Then, we register the MBean with the `registerMBean` method of the MBean server, as in the first example.

The advantage of this creation method is that the instantiate method of the MBean server also supports class loaders. If any are registered in the MBean server, they will automatically be used if the class is not available locally. See “The M-Let Class Loader” for more information on class loading.

Since we don’t take advantage of the class loaders here, we could have just called the class’ constructor directly. The main advantage is that, like the first method of MBean creation, we retain a direct reference to the new object. The direct reference lets us again use the MBean’s shortcut methods explicitly.

There are other combinations of instantiating and registering MBeans for achieving the same result. For example, we could use the default constructor and then set the `port` attribute of the MBean before starting it. Other combinations are left as an exercise to the reader.

Managing MBeans

In "Creating an MBean (Method 2)" on page 84, we rely totally on the MBean server to create and access an MBean. The code example in that section demonstrates how to get attributes and invoke operations through the MBean server. Here we will concentrate on the usage of MBean metadata classes when accessing MBeans representing resources.

We will rely on the `StandardAgent` and `DynamicAgent` classes presented in Instrumentation through MBeans. As mentioned in "Comparison with the `SimpleStandard` Example" on page 41, the two are nearly identical. We examine the method for displaying MBean metadata that is common to both: the same code works for any registered MBean, whether standard or dynamic.

CODE EXAMPLE 6–5 Processing MBean Information

```
private MBeanServer server = null; // assigned by MBeanServerFactory

private void printMBeanInfo(ObjectName name) {

    echo("Getting the management information for  " + name.toString() );
    MBeanInfo info = null;

    try {
        info = server.getMBeanInfo(name);
    } catch (Exception e) {
        e.printStackTrace();
        return;
    }
    echo("\nCLASSNAME: \t"+ info.getClassName());
    echo("\nDESCRIPTION: \t"+ info.getDescription());

    echo("\nATTRIBUTES");
    MBeanAttributeInfo[] attrInfo = info.getAttributes();
    if (attrInfo.length>0) {
        for(int i=0; i<attrInfo.length; i++) {
            echo(" ** NAME: \t"+ attrInfo[i].getName());
            echo("    DESCR: \t"+ attrInfo[i].getDescription());
            echo("    TYPE: \t"+ attrInfo[i].getType() +
                 "\tREAD: "+ attrInfo[i].isReadable() +
                 "\tWRITE: "+ attrInfo[i].isWritable());
        }
    } else echo(" ** No attributes **");

    echo("\nCONSTRUCTORS");
```

(continued)

```
    MBeanConstructorInfo[] constrInfo = info.getConstructors();
    // Note: the class necessarily has at least one constructor
    for(int i=0; i<constrInfo.length; i++) {
        echo(" ** NAME: \t"+ constrInfo[i].getName());
        echo("    DESCR: \t"+ constrInfo[i].getDescription());
        echo("    PARAM: \t"+ constrInfo[i].getSignature().length +
                 " parameter(s)");
    }

    echo("\nOPERATIONS");
    MBeanOperationInfo[] opInfo = info.getOperations();
    if (opInfo.length>0) {
        for(int i=0; i<constrInfo.length; i++) {
            echo(" ** NAME: \t"+ opInfo[i].getName());
            echo("    DESCR: \t"+ opInfo[i].getDescription());
            echo("    PARAM: \t"+ opInfo[i].getSignature().length +
                     " parameter(s)");
        }
    } else echo(" ** No operations ** ");

    echo("\nNOTIFICATIONS");
    MBeanNotificationInfo[] notifInfo = info.getNotifications();
    if (notifInfo.length>0) {
        for(int i=0; i<constrInfo.length; i++) {
            echo(" ** NAME: \t"+ notifInfo[i].getName());
            echo("    DESCR: \t"+ notifInfo[i].getDescription());
        }
    } else echo(" ** No notifications **");
}
```

The `getMBeanInfo` method of the MBean server gets the metadata of an MBean's management interface and hides the MBean's implementation. This method returns an `MBeanInfo` object which contains the MBean's description. We can then get the lists of attributes, operations, constructors, and notifications to display their descriptions. Recall that the dynamic MBean provides its own meaningful descriptions and that those of the standard MBean are default strings provided by the introspection mechanism of the MBean server.

Filtering MBeans

The base agent does very little filtering because it does very little management. Usually, filters are applied programmatically in order to get a list of MBeans to which some operations will apply. There are no management operations in the

MBean server which apply to a list of MBeans: you must loop through your list and apply the desired operation to each MBean.

Before exiting the agent application, we do a query of all MBeans so that we can unregister them properly. MBeans should be unregistered before being destroyed since they may need to perform some actions before or after being unregistered. See the Javadoc API of the `MBeanRegistration` interface for more information.

CODE EXAMPLE 6–6 Unregistering MBeans

```
public void removeAllMBeans() {

    try {
        Set allMBeans = myMBeanServer.queryNames(null, null);
        for (Iterator i = allMBeans.iterator(); i.hasNext();) {
            ObjectName name = (ObjectName) i.next();

            // Don’t unregister the MBean server delegate
            if ( ! name.toString().equals( ServiceName.DELEGATE ) ) {
                echo("Unregistering " + name.toString() );
                myMBeanServer.unregisterMBean(name);
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
    }
}
```

We use the `queryNames` method because we only need the object names to operate on MBeans. The null object name as a filter gives us all MBeans in the MBean server. We then iterate through the resulting set and unregister each one, except for the MBean server delegate. As described in “The MBean Server Delegate” on page 69, the delegate is also an MBean and so it will be returned by the query. However, if we unregister it, the MBean server can no longer function and remove the rest of our MBeans.

We recognized the delegate by its standard name which is given by the static field `ServiceName.DELEGATE`. The `ServiceName` class provides standard names and other default properties for communications and service MBeans. It also provides the version strings that are exposed by the delegate MBean. Note that, since the delegate is the only MBean created directly by the MBean server, it is the only one whose name cannot be overridden during its registration. This is why the delegate always has the same object name, and we are always sure to detect it.

Running the Base Agent Example

The *examplesDir*/`BaseAgent/` directory contains the source file of the `BaseAgent` application. Compile the `BaseAgent.java` file in this directory with the `javac` command. For example, on the Solaris platform, you would type:

```
$ cd examplesDir/BaseAgent/
$ javac -classpath classpath *.java
```

Again, we don't need the MBean classes at compile time, but they will be needed at runtime, since we don't use a dynamic class loader. You will need to have compiled the standard and dynamic MBean classes as described in "Running the Standard MBean Example" on page 28 and "Running the Dynamic MBean Example" on page 41. If you wish to load any other class in the base agent, you must include its directory or jar file in the classpath. To run the example, update your classpath to find the MBeans and launch the agent class:

```
$ java -classpath classpath:../StandardMBean:../DynamicMBean BaseAgent
```

Setting Trace Messages

Since the base agent enables internal tracing (see Code Example 6–1), you can also set the trace level and trace output on the command line. The tracing mechanism is covered in the Java Dynamic Management Kit 4.2 Tools Reference guide and in the Javadoc API of the `Trace` class. The simplest way to get the default tracing is to specify the *filename* for a trace log on the `java` command line:

```
$ java -classpath classpath -DTRACE_OUTPUT=filename BaseAgent
```

Agent Output

Besides any trace information, this agent displays output for the three types of MBean creation.

When the connection MBeans have been created, it is possible to connect to the agent through one of the protocols. Management applications may connect through the HTTP and RMI connector servers, as described in "Connector Servers" on page 110. If you connect to the base agent through the HTML adaptor, you could go through the same procedures as with the minimal agent.

When you are done, type "Enter" to remove all MBeans from the agent and exit the agent application.

CHAPTER 7

The Notification Mechanism

This topic presents the mechanisms for sending and receiving notifications by demonstrating them locally on the agent-side. MBeans for either resources or services are the source of notifications, called *broadcasters*. Other MBeans or other objects that want to receive the notifications register with one or more broadcasters, they are called the *listeners*.

Notification mechanisms are demonstrated through two sample broadcasters: the MBean server delegate which notifies listeners of MBean creation and de-registration, and an MBean which sends attribute change notifications.

The code samples are taken from the files in the `Notification2` example directory located in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

- “Overview” on page 94 presents the `Notification` object used to send generic events, identified by their notification type string.
- “MBean Server Delegate Notifications” on page 95 explains the concepts of notification broadcasters and listeners through the simple mechanism of an MBean sending notifications and a listener object receiving them.
- “Attribute Change Notifications” on page 99 provide an example of sub-classing the `Notification` object to provide additional information to the listener.
- “Running the Agent Notification Example” on page 104 will let you trigger attribute change notifications through the HTML adaptor.

Overview

The ability for resources and other entities to signal an event to their managing applications is a key functionality of management architectures. As specified in the Java Management extensions, notifications in the Java Dynamic Management Kit provide a generic event mechanism whereby one listener can receive all events sent by a broadcaster.

All notifications in the Java Dynamic Management Kit rely on the `Notification` class which itself inherits from Java event classes. A string called the *notification type* inside a `Notification` object gives the nature of the event, and other fields provide additional information to the recipient. This ensures that all MBeans, the MBean server, and remote applications may send and receive `Notification` objects and its subclasses, regardless of their inner type.

You may define new notification objects only by subclassing the `Notification` class. This ensures that the custom notifications will be compatible with the notification mechanism. New notification classes may be used to convey additional information to custom listeners, and generic listenters will still be able to access the standard `Notification` fields. However, since there are already fields provided for user data, subclassing is discouraged in the JMX architecture so that notification objects remain as universal as possible.

Listeners usually interact with notification broadcasters indirectly through the MBean server. The interface of the MBean server lets you associate a listener with any broadcaster MBean, thereby giving you dynamic access to any of the broadcasters that are registered. In addition, the MBean metadata provided through the MBean server contains the list of notification types that the MBean broadcasts.

The following diagram of an agent application summarizes how listeners register with broadcasters and then receive notifications.

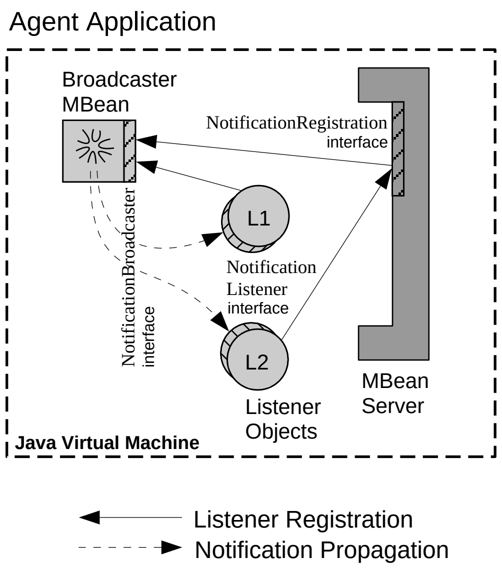


Figure 7–1 Listener Registration and Notification Propagation

MBean Server Delegate Notifications

The MBean server delegate object is an MBean that is automatically created and registered when an MBean server is started. It preserves the management model by serving as the management interface for the MBean server. The delegate exposes read-only information such as the vendor and version number of the MBean server. More importantly for this topic, it sends the notifications that relate to events in the MBean server: all MBean registrations and deregistrations generate a notification.

The `NotificationBroadcaster` Interface

A class must implement the `NotificationBroadcaster` interface to be recognized as a source of notifications in the JMX architecture. This interface provides the methods for adding or removing a notification listener to or from the broadcaster. When the broadcaster sends a notification, it must send it to all listeners that are currently registered through this interface.

This interface also specifies a method which returns information about all notifications which may be sent by the broadcaster. This method returns an array of `MBeanNotificationInfo` objects, each of which provides a name, a description string, and the type string of the notification.

As detailed in the Javadoc API, the `MBeanServerDelegate` class implements the `NotificationBroadcaster` interface. We know from the JMX specification that it sends notifications of the following types:

- `JMX.mbean.registered`
- `JMX.mbean.unregistered`

Note - Although broadcaster objects are almost always MBeans, they should not expose the methods of the `NotificationBroadcaster` interface. That is, the `MBean` interface of a standard MBean should never extend the `NotificationBroadcaster` interface. As we shall see in the next topic, "Notification Forwarding", the `remoteMBeanServer` interface of connector clients provides the methods needed to register for and receive notifications remotely.

The `NotificationListener` Interface

Listeners are the other players in the notification game. They must implement the `NotificationListener` interface, and they are registered in the notification broadcasters to receive the notifications. The listener interface defines a *handler* method that will receive all notifications of the broadcaster where the listener is registered. We say that a listener is *registered* when it has been added to the broadcaster's list of notification recipients; this is completely independent of any registration of either object in the MBean server.

Like the broadcaster, the listener is generic, meaning that it can handle any number of different notifications. Its algorithm usually involves determining the type of the notification and taking the appropriate action. A listener can even be registered with several broadcasters and handle all of the notifications that may be sent.

Note - The handler is a *callback* method that the broadcaster will invoke with the notification object it wishes to send. As such, it will execute in the broadcaster's thread and should therefore execute rapidly and return promptly. The code of the handler should rely on other threads to execute long computations or blocking operations.

In our example, the listener is a trivial class that has a constructor and the handler method. Our handler simply prints out the nature of the notification and the name of the MBean to which it applied. Other listeners on the agent side might themselves be MBeans that process the event and update the state of their resource or the quality of their service in response. For example, the relation service must know when any MBeans participating in relations are unregistered; it does this by listening to MBean server delegate notifications.

CODE EXAMPLE 7–1 The Listener for MBean Server Delegate Notifications

```
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.MBeanServerNotification;

public class AgentListener implements NotificationListener {
    [...]

    // Implementation of the NotificationListener interface
    //
    public void handleNotification(Notification notification,
                                   Object handback) {

        // Process the different types of notifications fired by the
        // MBean server delegate.
        String type = notification.getType();

        System.out.println(
            "\n\t>> AgentListener handles received notification:" +
            "\n\t>> --------------------------------------------");
        try {
            if (type.equals(
                    MBeanServerNotification.REGISTRATION_NOTIFICATION)) {
                System.out.println("\t>> \"" +
                    ((MBeanServerNotification)notification).getMBeanName() +
                    "\" has been registered in the server");
            }
            else if (type.equals(
                    MBeanServerNotification.UNREGISTRATION_NOTIFICATION)) {
                System.out.println("\t>> \"" +
                    ((MBeanServerNotification)notification).getMBeanName() +
                    "\" has been unregistered from the server\n");
            }
            else {
                System.out.println("\t>> Unknown event type (?)\n");
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

In most cases, the notification object passed to the handler method is an instance of the `Notification` class. This class provides the notification type as well as a time-stamp, a sequence number, a message string and user data of any type. All of these are provided by the broadcaster to pass any needed information to its listeners. Because listeners are usually registered through the MBean server, they only know the broadcaster by its object name: this is given by the `getSource` method of the notification object.

Note - The notification model does not assume that notifications will be received in the same order that they are sent. If notification order is critical to your application, your broadcaster should set the sequence numbers appropriately, and your listeners should sort the received notifications.

The MBean server delegate sends `MBeanServerNotification` objects which are subclasses of the `Notification` class. This subclass provides two constants to identify the notification types sent by the delegate and a method which gives the object name of the MBean which was registered or de-registered. Our notification handler uses these to print out the type of operation and the object name to which the operation applies.

Adding a Listener Through the MBean Server

Now that we have identified the objects involved, we need to add the listener to the notification broadcaster. Our example does this in the `main` method of the agent application:

CODE EXAMPLE 7–2 Registering for MBean Server Delegate Notifications

```
AgentListener agentListener = null;
[...]

echo("\nAdding the MBean server delegate listener...");
try {
    agentListener = new AgentListener();
    myAgent.myMBeanServer.addNotificationListener(
        new ObjectName(ServiceName.DELEGATE),
        agentListener, null, null);
} catch(Exception e) {
    e.printStackTrace();
    System.exit(0);
}
echo("done");
```

In our example, the agent application adds the `AgentListener` instance to the delegate MBean, which is known to be a broadcaster. The object name of the MBean server delegate is given by the `DELEGATE` constant in the `ServiceName` class. The listener is added through the `addNotificationListener` method of the MBean server: this method preserves the management architecture by adding listeners to MBeans while referring only the MBean object names.

If an MBean implements the listener interface and needs to receive certain notifications, it can add itself to a broadcaster. For example, an MBean could use its pre-registration method in order to add itself as a notification listener or it could expose a method that takes the object name of the notification broadcaster MBean. In both cases, its notification handler method would have to be designed to process all expected notification types.

The last two parameters of the `addNotificationListener` methods of both the `MBeanServer` and the `NotificationBroadcaster` interfaces define a filter and a handback object, respectively. Filter objects are defined by the `NotificationFilter` interface and provide a callback method that the broadcaster will invoke before calling the notification handler. If the filter is defined by the entity which adds the listener, it prevents the handler from receiving unwanted notifications.

Handback objects are added to a broadcaster along with a listener and are returned to the designated handler with every notification. The handback object is completely untouched by the broadcaster and can be used to transmit context information from the entity which adds the listener to the handler method. The functionality of filters and handback objects is not covered in this tutorial; please refer to the JMX specification for their full description.

Attribute Change Notifications

In this second part of the notification example, we demonstrate attribute change notifications that may be sent by MBeans. An MBean designer may choose to send notifications whenever the value of an attribute changes or is changed. The designer is free to implement this mechanism in any manner, according to the level of consistency required by the management solution.

The JMX specification only provides a subclass of notifications that should be used to represent the attribute change events: the `AttributeChangeNotification` class.

The `NotificationBroadcasterSupport` Class

The broadcaster in our example is a very simple MBean with only one attribute. The setter method for this attribute triggers a notification whenever the value actually changes. This policy is specific to our example, you might want to design an MBean that sends an attribute change every time the setter is called, regardless of whether or not the value is modified. In the same spirit, the fact that the reset operation changes the value of the attribute but doesn't send a notification is specific to our example; your management needs may vary.

Here is the code for our `SimpleStandard` MBean class (the code for its `MBean` interface has been omitted):

CODE EXAMPLE 7–3 The Broadcaster for Attribute Change Notifications

```
import javax.management.NotificationBroadcasterSupport;
import javax.management.MBeanNotificationInfo;
import javax.management.AttributeChangeNotification;

public class SimpleStandard
    extends NotificationBroadcasterSupport
    implements SimpleStandardMBean {

    /* "SimpleStandard" does not provide any specific constructors.
     * However, "SimpleStandard" is JMX compliant with regards to
     * constructors because the default constructor SimpleStandard()
     * provided by the Java compiler is public.
     */

    public String getState() {
        return state;
    }

    // The attribute setter chooses to send a notification only if
    // the value is modified
    public void setState(String s) {
        if (state.equals(s))
            return;
        AttributeChangeNotification acn = new AttributeChangeNotification(
            this, 0, 0, null, "state", "String", state, s);
        sendNotification(acn);
        state = s;
        nbChanges++;
    }

    [...]
    // The reset operation chooses not to send a notification even though
    // it changes the value of the state attribute
    public void reset() {
        state = "initial state";
        nbChanges = 0;
        nbResets++;
    }

    // Provide details about the notification type and class that is sent
    public MBeanNotificationInfo[] getNotificationInfo() {

        MBeanNotificationInfo[] ntfInfoArray = new MBeanNotificationInfo[1];

        String[] ntfTypes = new String[1];
        ntfTypes[0] = AttributeChangeNotification.ATTRIBUTE_CHANGE;

        ntfInfoArray[0] = new MBeanNotificationInfo( ntfTypes,
            "javax.management.AttributeChangeNotification",
            "Attribute change notification for the ’State’ attribute.");
        return ntfInfoArray;
    }

    private String      state = "initial state";
```

(continued)

(Continuation)

```
    private int         nbChanges = 0;
    private int         nbResets = 0;
}
```

You might be wondering how this MBean actually sends its notifications, or even how it implements the `NotificationBroadcaster` interface, for that matter. The answer to both is: by extension of the `NotificationBroadcasterSupport` class.

This class implements the `NotificationBroadcaster` interface in order to provide all the mechanisms for adding and removing listeners and sending notifications. It manages an internal list of listeners and their handback objects and updates this list whenever listeners are added or removed. In addition, the `NotificationBroadcasterSupport` class provides the `sendNotification` method to send a notification to all listeners currently on its list.

By extending this object, our MBean inherits all of this behavior. Subclassing `NotificationBroadcasterSupport` is a quick and convenient way to implement notification broadcasters. We don't even have to call a superclass constructor because it has a default constructor. We only need to override the `getNotificationInfo` method to provide details about all of the notifications that may be sent.

The Attribute Change Listener

Like our listener for MBean server notifications, our listener for attribute change notifications is a trivial class consisting of just the handler method.

CODE EXAMPLE 7–4 The Listener for Attribute Change Notifications

```
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.AttributeChangeNotification;

public class SimpleStandardListener implements NotificationListener {
    [...]

    // Implementation of the NotificationListener interface
    //
    public void handleNotification(Notification notification,
                                   Object handback) {

        // Process the different types of notifications fired by the
```

(continued)

```
        // simple standard MBean.
        String type = notification.getType();

        System.out.println(
            "\n\t>> SimpleStandardListener received notification:" +
            "\n\t>> --------------------------------------------");
        try {
            if (type.equals(AttributeChangeNotification.ATTRIBUTE_CHANGE)) {

  System.out.println("\t>> Attribute \"" +
      ((AttributeChangeNotification)notification).getAttributeName()
      + "\" has changed");
  System.out.println("\t>> Old value = " +
      ((AttributeChangeNotification)notification).getOldValue());
  System.out.println("\t>> New value = " +
      ((AttributeChangeNotification)notification).getNewValue());

            }
            else {
                System.out.println("\t>> Unknown event type (?)\n");
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Again, we are handling a subclass of the `Notification` class, this one specific to attribute change notifications. The `AttributeChangeNotification` class provides methods for extracting the information about the attribute, notably its name, its type and its values before and after the modification. Our handler does nothing more than display these to the user. If this handler were part of an MBean in a larger management solution, it would undoubtedly want to take some action, depending upon the change in value of the attribute.

As demonstrated by the broadcaster's code (see Code Example 7–3), the subclass can easily be instantiated and sent instead of a `Notification` object. Its constructor provides parameters for initializing all of the attribute-related values. In our example, we do not use significant values for the `sequenceNumber` and `timeStamp` parameters because our listener has no need for them. This is one great advantage of the Java Dynamic Management Kit: you only need to implement the level of functionality that you require for your management solution.

Adding a Listener Directly to an MBean

There is nothing that statically indicates that our MBean sends attribute change notifications. In our case it is a design decision, meaning that we know that the listener will receive attribute change notifications because we wrote the MBean that way. At runtime, the MBean server exposes the list of notifications in this MBean's metadata object, allowing a manager that is interested in attribute changes to register the appropriate listener.

Being confined to the agent, our example is much simpler. First we instantiate and register our simple MBean with the agent's MBean server. Then, because we have designed them to work together, we can add our listener for attribute changes to our MBean. Since we have kept a direct reference to the MBean instance, we can call its `addNotificationListener` method directly, without going through the MBean server.

CODE EXAMPLE 7–5 Registering for Attribute Change Notifications

```
SimpleStandard simpleStd = null;
ObjectName simpleStdObjectName = null;
SimpleStandardListener simpleStdListener = null;
[...]

try {
    simpleStdObjectName =
        new ObjectName("simple_mbean:class=SimpleStandard");
    simpleStd = new SimpleStandard();
    myAgent.myMBeanServer.registerMBean(simpleStd, simpleStdObjectName);
} catch(Exception e) {
    e.printStackTrace();
    System.exit(0);
}

echo("\nAdding the simple standard MBean listener...");
try {
    simpleStdListener = new SimpleStandardListener();
    simpleStd.addNotificationListener(simpleStdListener, null, null);
} catch(Exception e) {
    e.printStackTrace();
    System.exit(0);
}
echo("done");
```

There are several major implications to adding our listener directly to the MBean instance:

- Notification objects, or in this case subclasses, will contain a direct reference to the broadcaster object. This means that their `getSource` method will return a reference to the broadcaster instead of its object name. Our listener is unaffected by this issue since it never calls this method.

- This listener will need to be removed directly from the MBean instance. A listener added directly to the broadcaster object cannot be removed through the MBean server's methods, and vice versa.

The rest of the Agent object's code performs the setup of the agent's MBean server and various input and output for running the example. Similar agents were already presented in detail in the lesson on "Agent Applications".

Running the Agent Notification Example

Now that we have seen all of our notification broadcaster objects and all of our listener handlers, we are ready to run the example.

The *examplesDir*/`Notification2` directory contains all of the files for the simple MBean, the listener objects, and the agent. When launched, the agent application adds the MBean server delegate listener first, so that a notification can be seen for the creation of the MBean. Attribute change notifications are triggered by modifying attributes through the HTML adaptor.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/Notification2/
$ javac -classpath classpath *.java
```

To run the example, launch the agent application:

```
$ java -classpath classpath Agent
```

▼ Triggering Notifications in the Agent Example

1. **After the agent application has started and added the MBean server delegate listener, press `<Enter>` to create the simple MBean.**

   Before the next printout of the agent application, you should see the text generated by the `AgentListener` class. Its handler method has been called with an MBean creation notification, and it prints out the object name of our new MBean.

2. **Now that the simple MBean is registered and the `SimpleStandardListener` has been added as a listener, you can trigger attribute change notifications by modifying the `State` attribute through the HTML adaptor.**

   Load the following URL in your browser:

   ```
   http://localhost:8082/
   ```

   If you get an error, you may have to switch off proxies in your preference settings or substitute your machine name for `localhost`. Any browser on your local network can also connect to this agent by using your machine name in this URL.

   In the attribute table of our MBean view, enter a new value for the `State` attribute and click the "Apply" button. Every time you do this, you should see the output of the attribute change listener in the terminal window where you launched the agent.

3. **When you are finished with the attribute change notifications, press `<Enter>` in the agent's terminal window to remove our simple MBean.**

   Again, you should see the output of the MBean server delegate listener. This time it has detected that our MBean has been de-registered from the MBean server.

4. **Press `<Enter>` again to stop the agent application.**

PART III Remote Management Applications

In the lesson on “Agent Applications”, we saw how to access and manage a Java Dynamic Management agent through the HTML protocol adaptor. Protocol adaptors provide a view of an agent through some communication protocol. In this lesson, we present protocol connectors and proxy MBeans for managing agents programmatically.

The Java Dynamic Management Kit goes beyond the scope of the JMX specification to define the APIs needed to develop distributed management applications in the Java programming language. These remote applications establish connections with agents through protocol connectors over RMI, HTTP or HTTPS. The connector client object exposes a remote version of the MBean server interface. The connector server object in the agent transmits management requests to the MBean server and forwards any replies.

Connectors allow you to develop a management application which is both protocol independent and location independent. Once the connection is established, the communication layer is transparent, and the manager can issue requests, as if it were directly invoking the MBean server. Using proxy objects which represent MBeans, the design of the management application is even simpler and development time is reduced.

This homogeneity of the API makes it possible to develop portable management applications which can run either in an agent or in a remote management application. This simplifies the development and testing of applications, and it also allows functionality to evolve along with the management solution. As your agent and manager platforms evolve, management policies can be implemented at higher levels of management, and intelligent monitoring and processing logic can be moved down into agents.

Connectors and proxies provide an abstraction of the communication layer between agent and manger, but they also provide mechanisms for simplifying the management task. Notification forwarding with configurable caching and pull policies lets you dynamically optimize bandwidth usage. The connector heartbeat monitors a connection, applies a retry policy, and automatically frees the resources when there is a communication failure. Finally, the connector server in an agent can act as a watchdog and refuse unauthorized requests based on the type of operation.

This lesson contains the following topics:

- "Protocol Connectors" establish a connection between a management application written in the Java programming language and a Java Dynamic Management agent. Agents are identified by an address and port number, all other details of the communication are hidden. Once the connection is established, the remote management application may access the MBeans in the agent. All connectors also implement the heartbeat mechanism to monitor the connection.
- "MBean Proxies" represent MBeans so that they are easier to access. A proxy object exposes the interface of its MBean for direct invocation. It then encodes the management request which it forwards to the MBean server and returns any response to its caller. Specific proxies can be generated for standard MBeans, but dynamic MBeans can only be accessed through generic proxies, similar to their `DynamicMBean` interface.
- "Notification Forwarding" extends the concept of notification listeners to the manager side. This topic covers how manager-side listeners can register with agent-side broadcasters. The example then shows how the connector client and server interact to provide both pull and push modes for forwarding notifications from the agent to the manager.
- "Access Control and Security" presents the security features that can be enabled for a given connection. The HTTP-based connectors include a password mechanism that will refuse unauthorized connections. Context checking and the general filtering mechanism can implement a complex algorithm for allowing or disallowing management requests to an agent. Finally, the HTTPS connector encrypts and protects data as it transits over the network between connector client and server components.

CHAPTER **8**

Protocol Connectors

Protocol connectors provide a point-to-point connection between a Java Dynamic Management agent and a management application. Each connector relies on a specific communication protocol, but the API that is available to the management application is identical for all connectors and is entirely protocol-independent.

A connector consists of a connector server component registered in the agent and a connector client object instance in the management application. The connector client exposes a remote version of the MBean server interface. Each connector client represents one agent to which the manager wishes to connect. The connector server replies to requests from any number of connections and fulfills them through its MBean server. Once the connection is established, the remoteness of the agent is transparent to the management application, except for any communication delays.

Connectors rely on the Java serialization package to transmit data as Java objects between client and server components. Therefore, all objects needed in the exchange of management requests and responses must be instances of a serializable class. However, the data encoding and sequencing are proprietary, and the raw data of the message contents in the underlying protocol are not exposed by the connectors.

All connectors provided with the Java Dynamic Management Kit implement a heartbeat mechanism to automatically detect when a connection is lost. This allows the manager to be notified when the communication is interrupted and when it is restored. If the connection is not reestablished, the connector automatically frees the resources that were associated with the lost connection.

The code samples in this topic are taken from the files in the `SimpleClients` and `HeartBeat` example directories located in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

- “Connector Servers” on page 110 presents the agent-side component of a protocol connector.

- "Connector Clients" on page 113 presents the manager-side component and how it establishes a connection.
- "The Heartbeat Mechanism" on page 121 explains how a connector monitors a connection and demonstrates how to run the heartbeat example.

Connector Servers

The connector server on the agent side listens for management requests issued through a corresponding connector client. It transmits these requests to its MBean server and forwards any response back to the management application. The connector server also forwards notifications, when the management application has registered to receive them through its connector client (see "Notification Forwarding" for more details).

There is a connector server for each of the protocols supported in the Java Dynamic Management Kit: RMI, HTTP and HTTPS. They all inherit from the `CommunicatorServer` class which is the superclass for all protocol adaptors and connector servers. This class defines the methods needed to control the port and communication settings that are common to all. Each connector server class then provides specific controls, such as the service name for RMI and authentication information for HTTP. This example covers the RMI connector server, and the HTTP authentication mechanism is covered in "Password-Based Authentication" on page 162.

A connector server listens for incoming requests from its corresponding connector client, decodes that request and encodes the reply. Several connector clients may establish connections with the same connector server, and the connector server can handle multiple requests simultaneously. There only needs to be one connector server MBean per protocol to which the agent needs to respond. However, several connector servers for the same protocol can coexist in an agent for processing requests on different ports.

Instantiating an RMI Connector Server

The RMI connector server is an MBean, so we instantiate its class and register it in the MBean server. This operation could also be performed remotely, for example if a management application wishes to access an agent through an alternate protocol.

CODE EXAMPLE 8–1 Instantiating the RMI Connector Server

```
// Instantiate an RMI connector server with default port
//
CommunicatorServer rmiConnector = new RmiConnectorServer();
```

(Continuation)

```
try {
    // We let the RMI connector server provides its default name
    ObjectName rmiConnectorName = null;
    ObjectInstance rmiConnectorInstance =
        myMBeanServer.registerMBean( rmiConnector, rmiConnectorName );

} catch(Exception e) {
    e.printStackTrace();
}
```

Other constructors for the `RmiConnectorServer` class have parameters for specifing the port and the RMI service name that the connector server will use. The default constructor assigns port 1099 and “`name=RmiConnectorServer`”, as given by the static variables `RMI_CONNECTOR_PORT` and `RMI_CONNECTOR_SERVER`, respectively, of the `ServiceName` class. Both attributes can also be accessed through the getter and setter methods of the `RmiConnectorServer` class.

Each connector uses different parameters that are specific to its protocol. For example, the HTTP connector does not need a service name. The default values for all parameters are given by the static variables of the `ServiceName` class.

Registering a connector server as an MBean implies that its MBean server will handle the remote requests that it receives. However, you may specify a different object for fulfilling management requests through the `setMBeanServer` method that the `RmiConnectorServer` class inherits from the `CommunicatorServer` class. For security reasons, this method is not exposed in the RMI connector server MBean, so it must be called from the agent application.

Registering the connector server as an MBean is optional: for example, you may not want it exposed for management. In this case, you must use the `setMBeanServer` method to specify an object which implements the `MBeanServer` interface so that it can fulfill management requests.

Connector States

Like all communicator servers, the RMI connector server has a connection state which is identified by the static variables of the `CommunicatorServer` class:

- `OFFLINE` – Stopped and not responding.
- `STARTING` – In a transitional state and not yet responding.
- `ONLINE` – Able to respond to management requests.

- `STOPPING` – In a transitional state and no longer responding.

All connector servers are `OFFLINE` after their instantiation, so they must be started explicitly. Then, you must wait for a connector server to come `ONLINE` before it can respond to connections on its designated port.

CODE EXAMPLE 8–2 Starting the RMI Connector Server

```
// Explicitly start the RMI connector server
//
rmiConnector.start();

// waiting for it to leave starting state...
while (rmiConnector.getState() == CommunicatorServer.STARTING) {
    try {
        Thread.sleep( 1000 );
    } catch (InterruptedException e) {
        continue;
    }
}
```

Instead of blocking the application thread, you may register a listener for attribute change notifications concerning the `State` attribute of the connector server MBean. All connector servers implement this notification which contains both old and new values of the attribute (see "Attribute Change Notifications" on page 99). Listeners in the agent can then asynchronously detect when the state changes from `STARTING` to `ONLINE`.

Note - During the `STARTING` state, the RMI connector server registers its RMI service name with the RMI registry on the local host for its designated port. If no registry exists, one is instantiated for the given port. Due to a limitation of the JDK software, creating a second registry in the same Java VM will fail. As a workaround, before starting an RMI connector server on a new, distinct port number in an agent, you must run the `rmiregistry` command from a terminal on the same host. This limitation is specific to the RMI connector: the HTTP protocols do not require a registry.

The `stop` method is used to take a connector server offline. The `stop` method is also called by the `preDeregister` method that all connector servers inherit. Stopping a connector server will interrupt all requests that are pending and close all connections that are active. When a connection is closed, all of its resources are cleaned up, including all notification listeners, and the connector client may be notified by a heartbeat notification (see "The Heartbeat Mechanism" on page 121). A connection that is closed can no longer be reopened, the connector client must establish a new connection when the connector server is restarted.

The `setPort` method that all connector servers inherit from the `CommunicatorServer` class allows you to change the port on which management requests will be expected. You can only change the port when the connector server is offline, so it must be explicitly stopped and then restarted. The same rule applies to the `setServiceName` method which is specific to the RMI connector server. These methods are also exposed in the MBean interface, along with `start` and `stop`, allowing a remote management application to configure the connector server through a different connection.

Connector Clients

The manager application interacts with a connector client in order to access an agent through an established connection. Through its implementation of the `RemoteMBeanServer` interface, a connector client provides methods for handling the connection and for accessing the agent. This interface specifies nearly all of the same methods as the `MBeanServer` interface, meaning that an agent is fully manageable from a remote application.

Through the connector, the management application sends management requests to the MBeans located in a remote agent. Components of the management application access remote MBeans by calling the methods of the connector client for getting and setting attributes and invoking operations on the MBeans. The connector client then returns the result, providing a complete abstraction of the communication layer.

The `RemoteMBeanServer` Interface

All connector clients implement the `RemoteMBeanServer` interface in order to expose the methods needed to access and manage the MBeans in a remote agent. This interface allows all management operations that would be possible directly in the agent application. In fact, the methods of the connector client for accessing MBeans remotely have exactly the same name and signature as their equivalent methods in the `MBeanServer` interface.

The following methods are defined identically in both the `MBeanServer` and the `RemoteMBeanServer` interfaces:

TABLE 8–1 The Set of Shared Methods

♦ `void`	**`addNotificationListener`**`(ObjectName name, NotificationListener listener, NotificationFilter filter, java.lang.Object handback)`
`ObjectInstance`	**`createMBean`**`(*)` – All four overloaded forms of this method
♦ `java.lang.Object`	**`getAttribute`**`(ObjectName name, java.lang.String attribute)`
♦ `AttributeList`	**`getAttributes`**`(ObjectName name, java.lang.String[] attributes)`
`java.lang.String`	**`getDefaultDomain`**`()`
`java.lang.Integer`	**`getMBeanCount`**`()`
♦ `MBeanInfo`	**`getMBeanInfo`**`(ObjectName name)`
`ObjectInstance`	**`getObjectInstance`**`(ObjectName name)`
♦ `java.lang.Object`	**`invoke`**`(ObjectName name, java.lang.String operationName, java.lang.Object[] params, java.lang.String[] signature)`
`boolean`	**`isInstanceOf`**`(ObjectName name, java.lang.String className)`
`boolean`	**`isRegistered`**`(ObjectName name)`
`java.util.Set`	**`queryMBeans`**`(ObjectName name, QueryExp query)`
`java.util.Set`	**`queryNames`**`(ObjectName name, QueryExp query)`
♦ `void`	**`removeNotificationListener`**`(ObjectName name, NotificationListener listener)`
♦ `void`	**`setAttribute`**`(ObjectName name, Attribute attribute)`
♦ `AttributeList`	**`setAttributes`**`(ObjectName name, AttributeList attributes)`
♦ `void`	**`unregisterMBean`**`(ObjectName name)`

♦ These methods are defined in the `ProxyHandler` interface; see the next topic: “Local and Remote Proxies” on page 131.

Components of a management application which rely solely on this common subset of methods may be instantiated in either the agent or the manager application. Such components are location independent and may be reused either locally or remotely as managemement solutions evolve. This symmetry also allows the design of advanced management architectures where functionality may be deployed either in the agent or in the manager, depending on runtime conditions.

The other, unshared methods of the RemoteMBeanServer interface are used to establish and monitor the connection. In the following section, we will establish a connection and access MBeans directly through the connector client. In “The Heartbeat Mechanism” on page 121, we will see how to monitor a connection and detect when it is lost.

Establishing a Connection

The target of a connection is identified by a protocol-specific implementation of the ConnectorAddress interface. This object contains all the information that a connector client needs to establish a connection with the target agent. All address objects specify a host name and port number. An RMI address adds the required service name, and HTTP-based addresses have an optional authenication field (see “Password-Based Authentication” on page 162). In addition, the ConnectorType string identifies the protocol without needing to introspect the address object

In our example, the target of the connection is an active RMI connector server identified by an RmiConnectorAddress object. We use the default constructor to instantiate a default address object, but otherwise, these parameters can be specified in a constructor or through setter methods. The default values of the information contained in the RmiConnectorAddress object are the following:

- The ConnectorType identifies the protocol that is used; its value is “SUN RMI” for the RmiConnectorAddress class.
- The default RMI port is 1099, as given by the static variable RMI_CONNECTOR_PORT in the ServiceName class.
- The Host is the name of the machine where the target agent is running; by default, its value is the localhost.
- The Name attribute specifies the RMI registry service name of the adaptor server. Its default value is “name=RmiConnectorServer”, which is the value of the RMI_CONNECTOR_SERVER static variable in the ServiceName class.

The RmiConnectorAddress object is used as the parameter to the connect method of the RmiConnectorClient instance. This method will try to establish the connection and will throw an exception if there is a communication or addressing error. Otherwise, when the connect method returns, the connector client will be ready to perform management operations on the designated agent.

CODE EXAMPLE 8–3 Establishing a Connection

```
echo("\t>> Instantiate the RMI connector client...");
connectorClient = new RmiConnectorClient();

echo("\t>> Instantiate a default RmiConnectorAddress object...");
RmiConnectorAddress address = new RmiConnectorAddress();
```

(continued)

```
// display the default values
echo("\t\tTYPE\t= "   + address.getConnectorType());
echo("\t\tPORT\t= "   + address.getPort());
echo("\t\tHOST\t= "   + address.getHost());
echo("\t\tSERVER\t= " + address.getName());
echo("\t<< done <<");

echo("\t>> Connect the RMI connector client to the agent...");
try {
     connectorClient.connect( address );

} catch(Exception e) {
     echo("\t!!! RMI connector client could not connect to the agent !!!");
     e.printStackTrace();
     System.exit(1);
}
```

Managing MBeans Remotely

Once the connection to an agent is established, the management application can access that agent's MBeans through the RemoteMBeanServer interface of the connector client. Invoking these methods has exactly the same effect as invoking the equivalent methods directly on the MBean server instance.

It is possible to restrict access to certain methods of the MBean server when they are called through the connector client, but this is performed by a security mechanism in the connector server: see "Context Checking" on page 165 for more details.

Creating and Unregistering MBeans in the Agent

We use the createMBean method to instantiate and register an object from its class name. This class must already be available in the agent application's classpath, or you can use the createMBean method which takes the name of a class loader (see "M-Let Loading from a Manager (Java 2)" on page 194 for more details).

CODE EXAMPLE 8–4 Creating and Unregistering an MBean Remotely

```
private void doWithoutProxyExample(String mbeanName) {

    try {

        // build the MBean ObjectName instance
        //
        ObjectName mbeanObjectName = null;
        String domain = connectorClient.getDefaultDomain();
        mbeanObjectName = new ObjectName( domain + ":type=" + mbeanName );

        // create and register an MBean in the MBeanServer of the agent
        //
        echo("\nCurrent MBean count in the agent = "+
             connectorClient.getMBeanCount());
        echo("\n>>> CREATE the " + mbeanName +
             " MBean in the MBeanServer of the agent:");
        String mbeanClassName = mbeanName;

        ObjectInstance mbeanObjectInstance =
        connectorClient.createMBean( mbeanClassName, mbeanObjectName );

        echo("\tMBEAN CLASS NAME      = " +
             mbeanObjectInstance.getClassName() );
        echo("\tMBEAN OBJECT NAME     = " +
             mbeanObjectInstance.getObjectName() );
        echo("\nCurrent MBean count in the agent = "+
             connectorClient.getMBeanCount() );

        [...] // Retrieve MBeanInfo and access the MBean (see below)

        // unregister the MBean from the agent
        //
        echo("\n>>> UNREGISTERING the "+ mbeanName +" MBean");
        connectorClient.unregisterMBean(
            mbeanObjectInstance.getObjectName() );

        [...]

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The above sample shows the use of other calls to the remote agent, such as `getDefaultDomain` and `getMBeanCount` which have the same purpose as in an agent application.

Accessing MBean Attributes and Operations

Once you can access the object names for MBeans in the agent, you can know their management interface from their `MBeanInfo` object. The following code is actually called in between the MBean creation and deregistration shown in the previous code sample.

CODE EXAMPLE 8–5 Retrieving the MBeanInfo Object

```
ObjectName mbeanObjectName = mbeanObjectInstance.getObjectName();

echo("\n>>> Getting the management information of the MBean");
MBeanInfo info = null;
try {

    info = connectorClient.getMBeanInfo( mbeanObjectName );

} catch (Exception e) {
    echo("\t!!! Could not get MBeanInfo object for "+ mbeanObjectName );
    e.printStackTrace();
    return;
}

// display content of MBeanInfo object
//
echo("\nCLASSNAME: \t"+ info.getClassName());
echo("\nDESCRIPTION: \t"+ info.getDescription());
echo("\nATTRIBUTES");
MBeanAttributeInfo[] attrInfo = info.getAttributes();
if ( attrInfo.length>0 ) {
    for( int i=0; i<attrInfo.length; i++ ) {
        echo(" ** NAME: \t"+ attrInfo[i].getName());
        echo("    DESCR: \t"+ attrInfo[i].getDescription());
        echo("    TYPE: \t"+ attrInfo[i].getType() +
             "\tREAD: "+ attrInfo[i].isReadable() +
             "\tWRITE: "+ attrInfo[i].isWritable());
    }
} else echo(" ** No attributes **");

[...]
```

It is then straightforward to perform management operations on MBeans through the connector client. As in an agent, we call the generic getters, setters and invoke methods with the object name of the MBean, the name of the attribute or operation, and any parameters. As in the methods of the MBean server, we need to use the `Attribute` and `AttributeList` classes to pass attributes as name-value pairs.

CODE EXAMPLE 8–6 Accessing an MBean Through the Connector Client

```
try {
    // Getting attribute values
    String State = (String)
        connectorClient.getAttribute( mbeanObjectName, "State" );
    Integer NbChanges = (Integer)
        connectorClient.getAttribute( mbeanObjectName, "NbChanges" );
    echo("\tState     = \"" + State + "\"");
    echo("\tNbChanges = " + NbChanges);

    // Setting the "State" attribute
    Attribute stateAttr = new Attribute("State", "new state from client");
    connectorClient.setAttribute(mbeanObjectName, stateAttr);

    // Invoking the "reset" operation
    Object[] params = new Object[0];
    String[] signature = new String[0];
    connectorClient.invoke(mbeanObjectName, "reset", params, signature);

} catch (Exception e) {
    e.printStackTrace();
    return;
}
```

All other MBean access methods are available in the same manner, such as bulk getters and setters, and the query methods.

Creating and Accessing Dynamic MBeans

In the first run of the example, the management application creates, manages and deregisters a standard MBean in the remote agent. However, standard and dynamic MBeans are designed to be managed through the same methods, both in the `MBeanServer` and in the `RemoteMBeanServer` interfaces.

As shown in Code Example 8–4, the subroutine of the example application takes only a single class name parameter. The first time this subroutine is called, the example passes the class name of a standard MBean, and the second time, that of a dynamic MBean. For the example to run, the two MBeans must have an identical management interface. By extension of this special case, we see that the connector client can manage dynamic MBeans through the same methods as it manages standard MBeans, without making any distinction between the two.

Running the Simple Client Example

The *examplesDir*/`SimpleClients` directory contains all of the files for three sample managers, a base agent, and some MBeans to manage. In this topic, we run the `ClientWithoutProxy` application which demonstrates simple operations on MBeans through an RMI connector.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/SimpleClients/
$ javac -classpath classpath *.java
```

We will not need all the files for this topic, but they will be used in the next topic, “MBean Proxy Objects”. In this demonstration, we only need the `BaseAgent` and `ClientWithoutProxy` applications, as well as the standard and dynamic MBeans.

▼ Instructions

1. **Launch the agent in another terminal window on the *same* host with the following command:**

   ```
   $ java -classpath classpath BaseAgent
   ```

   The agent creates an HTML protocol adaptor and an RMI connector server to which the client application will establish a connection, and then it waits for management operations.

2. **Wait for the agent to be completely initialized, then launch the manager with the following command:**

   ```
   $ java -classpath classpath ClientWithoutProxy
   ```

   The client application creates the RMI connector client and establishes the connection to base agent.

3. **Press `<Enter>` in the manager window to step through the example.**

   The management application instantiates both types of MBeans, looks at their metadata, and performs management operations on them. The results of each step are displayed in the terminal window.

4. **At any time, you can view the agent through its HTML adaptor and interact with the MBeans created by the management application.**

   For example, immediately after the manager creates an MBean, you could modify its attributes and see this change reflected when the connector client access the new values.

5. **Press `<Enter>` in both windows to stop the agent and manager applications.**

The Heartbeat Mechanism

The heartbeat mechanism monitors the connection between a manager and an agent and automates the cleanup procedure when the connection is lost. This allows both the manager and the agent to release resources that were allocated for maintaining the connection.

The mechanism is entirely contained in the connector client and connector server components, no additional objects are involved. In addition, connector clients send notifications that the manager application can receive to be aware of changes in the status of a connection.

All connector clients of the Java Dynamic Management Kit implement the `HeartBeatClientHandler` interface to provide a heartbeat mechanism. This means that agent-manager connections over RMI, HTTP, and HTTPS can be monitored and controlled in the same way. A manager application could even use the same notification handler for all connector clients where the heartbeat mechanism is activated.

Configuring the Heartbeat

To monitor the connection, the connector client sends periodic heartbeats (*ping* requests) to the connector server which acknowledges them by sending a reply (*ping* responses). If either heartbeat goes missing, the components of the connector will retry until either the connection is reestablished or the number of retries has been exhausted.

In a connector client, the methods specified by the `HeartBeatClientHandler` interface set the heartbeat period and the number of retries that will be attempted. You should determine these parameters empirically to implement the desired connection monitoring behavior, taking into account the network conditions and topology between the hosts of your manager and agent applications.

In the following code example, the management application configures the heartbeat mechanism before the connection to an agent is established.

CODE EXAMPLE 8–7 Configuring the Heartbeat in the Connector Client

```
// CREATE a new RMI connector client
//
echo("\tInstantiate the RMI connector client...");
connectorClient = new RmiConnectorClient();

// SET heartbeat period to 1 sec. Default value is 10 secs
```

(continued)

```
//
echo("\tSet heartbeat period to 1 second...");
connectorClient.setHeartBeatPeriod(1000);

// SET heartbeat number of retries to 2. Default value is 6 times
//
echo("\tSet heartbeat number of retries to 2 times...");
connectorClient.setHeartBeatRetries(2);
```

Using the same methods, the heartbeat configuration may also be modified at any time, even after the connection has been established. By default, the heartbeat mechanism is activated in a connector with a 10 second heartbeat and 6 retries, meaning that a connection which cannot be reestablished within a minute will be assumed to be lost.

Setting the number of heartbeat retries to zero will cause a lost connection to be signalled immediately after the heartbeat fails. Setting the heartbeat period to zero will deactivate the mechanism and prevent any further connection failures from being detected.

No specific configuration is necessary on the agent-side connector server: it automatically responds to the heartbeat messages. These heartbeat messages contain the current heartbeat settings from the connector client which also configure the connector server. In this way, both client and server will apply the same retry policy, and when the configuration is updated in the connector client, it is immediately reflected in the connector server. The connector server may handle multiple connections from different management application, each with its specific heartbeat configuration.

The connector server will apply its retry policy when the next expected heartbeat message is not received within the heartbeat period. From that moment, the connector server will begin a timeout period which lasts 20% longer than the number of retries times the heartbeat period. This corresponds to the time during which the connector client will attempt to resend the heartbeat, with a safety margin to allow for communication delays. If no further heartbeat is received in that timeout, the connection is determined to be lost.

The heartbeat ping messages also contain a connection identifier so that connections are not erroneously reestablished. If a connector server is stopped, thereby closing all connections, and then restarted between two heartbeats or before the client’s timeout period has elapsed the server will respond to heartbeats from a previous connection. However, the connector client will detect that the identifier has changed and will immediately declare that the connection is lost, regardless of the number of remaining retries.

During the timeout period, the notification push mechanism in the connector server will be suspended to avoid losing notifications (see "Notification Forwarding"). Similarly, while the connector client is retrying the heartbeat, it must suspend the notification pull mechanism if it is in effect.

When a connection is determined to be lost, both the connector client and server free any resources that were allocated for maintaining the connection. For example, the connector server will unregister all local listeners and delete the notification cache needed to forward notifications. Both components also return to a coherent, functional state, ready to establish or accept another connection.

The state of both components after a connection is lost is identical to the state which is reached after the `disconnect` method of the connector client is invoked. In fact, the `disconnect` method is invoked internally by the connector client when a connection is determined to be lost, and the equivalent, internal method is called in the connector server when its timeout elapses without recovering a heartbeat.

Receiving Heartbeat Notifications

The connector client also sends notifications which signal any changes in the state of the connection. These notifications are instances of the `HeartBeatNotification` class. The `HeartBeatClientHandler` interface includes methods specifically for registering for heartbeat notifications. These methods are distinct from those of the `NotificationRegistration` interface that a connector client implements for transmitting agent-side notifications (see "Registering Manager-Side Listeners" on page 146).

CODE EXAMPLE 8–8 Registering for Heartbeat Notifications

```
// Register this manager as a listener for heartbeat notifications
// (the filter and handback objects are not used in this example)
//
echo("\tAdd this manager as a listener for heartbeat notifications...");
connectorClient.addHeartBeatNotificationListener(this, null, null);
```

Instances of heartbeat notifications contain the connector address object from the connection that generated the event. This allows the notification handler to listen to any number of connectors and retrieve all relevant information about a specific connection when it triggers a notification. The `HeartBeatNotification` class defines constants to identify the possible notification type strings for heartbeat events:

- `CONNECTION_ESTABLISHED` – Emitted when the `connect` method succeeds.
- `CONNECTION_RETRYING` – After the heartbeat fails and if the number of retries is not zero, this notification type is emitted *once* when the first retry is sent.

- CONNECTION_REESTABLISHED – Emitted if the heartbeat recovers during one of the retries.
- CONNECTION_LOST – Emitted after the heartbeat and all retries, if any, have failed or when a heartbeat contains the wrong connection identifier, indicating that the connector server has been stopped and restarted.
- CONNECTION_TERMINATED – Emitted when the disconnect method successfully terminates a connection and frees all the resources it used; therefore, this notification type is received after both a user–terminated connection and after a connection is lost.

Once they are established, connections can go through any number of retrying-reestablished cycles, and then be terminated by the user or determined to be lost and terminated automatically. When the heartbeat mechanism is deactivated by setting the heartbeat period to zero, only heartbeat notifications for normally established and normally terminated connections will continue to be sent. In that case, connections may be lost but they will not be detected nor signaled by a notification.

The following diagram shows the possible sequence of heartbeat notifications during a connection. Retries are enabled when the getHeartBeatRetries method returns an integer greater than zero.

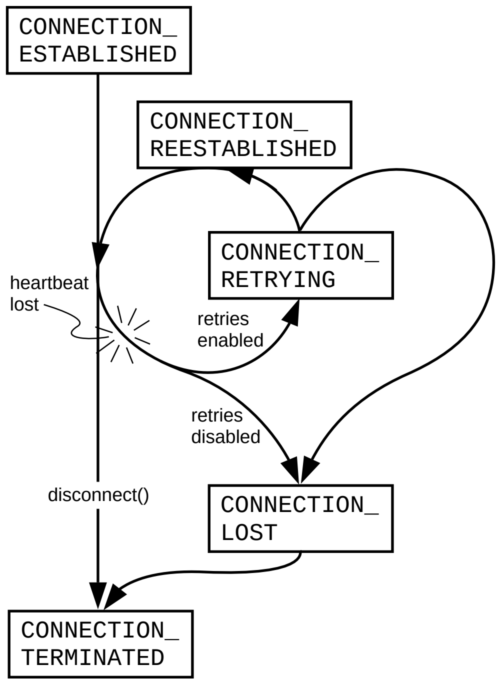


Figure 8–1 Sequencing of Heartbeat Notifications

The following example shows the source code for the notification handler method in our manager class. The handler prints out the notification type and the RMI address associated with the connector that generated the notification:

CODE EXAMPLE 8–9 Heartbeat Notification Handler

```
public void handleNotification(Notification notification, Object handback){

    echo("\n>>> Notification has been received...");
    echo("\tNotification type = " + notification.getType());

    if (notification instanceof HeartBeatNotification) {
        ConnectorAddress notif_address =
            ((HeartBeatNotification)notification).getConnectorAddress();

        if (notif_address instanceof RmiConnectorAddress) {
            RmiConnectorAddress rmi_address =
                (RmiConnectorAddress) notif_address;

            echo("\tNotification connector address:");
            echo("\t\tTYPE   = " + rmi_address.getConnectorType());
            echo("\t\tHOST   = " + rmi_address.getHost());
            echo("\t\tPORT   = " + rmi_address.getPort());
            echo("\t\tSERVER = " + rmi_address.getName());
         }
    }
}
```

In the agent application, the connector server does not emit any notifications about the state of its connections. The HTTP protocol-based connectors do provide a count of active clients, but there is no direct access to heartbeat information in an agent's connector servers.

Running the Heartbeat Example

The *examplesDir*/`HeartBeat` directory contains all of the files for the `Agent` and `Client` applications which demonstrate the heartbeat mechanism through an RMI connector.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/HeartBeat/
$ javac -classpath classpath *.java
```

To demonstrate the various communication scenarios, we run the example three times: once to see a normal termination, once to see how the manager reacts to a lost connection, and once to see how the agent reacts to a lost connection.

▼ Normal Termination

1. **Launch the agent on another host or in another terminal window with the following command:**

   ```
   $ java -classpath classpath Agent
   ```

   The agent only creates the RMI connector server to which the client application will establish a connection, and then it waits for management operations.

2. **Wait for the agent to be completely initialized, then launch the manager with the following command, where *hostname* is the name of the machine running the agent. The RMI connector in this example uses port 1099 by default. If you launched the agent on the same machine, you can omit the *hostname* and the port number:**

   ```
   $ java -classpath classpath Client hostname 1099
   ```

   The client application creates the RMI connector client, configures its heartbeat, and registers a notification listener, as seen in the code examples. When the connection is established, the listener outputs the notification information in the terminal window.

3. **Press `<Enter>` in the manager window to call the `disconnect` method on the connector client and stop the `Client` application.**

   In the terminal window, the heartbeat notification listener outputs the information for the normal connection termination before the application ends.

4. **Leave the agent application running for the next scenario.**

▼ Connector Client Reaction

1. **Launch the `Client` application again with the same command as before:**

```
$ java -classpath classpath Client hostname 1099
```

2. **When the connection is established, type** `<Control-C>` **in the *agent's* window to stop the connector server and the agent application. This simulates a broken communication channel as seen by the connector client.**

   Less than a second later, when the next heartbeat fails, the heartbeat retrying notification is displayed in the manager's terminal window. Two seconds later, after both retries have failed, the lost connection and terminated connection notifications are displayed.

3. **Press** `<Enter>` **in the manager window to exit the** `Client` **application.**

▼ Connector Server Reaction

1. **Launch the agent in debug mode on another host or in another terminal window with the following command:**

```
$ java -classpath classpath -DLEVEL_DEBUG Agent
```

2. **Wait for the agent to be completely initialized, then launch the** `Client` **application again with the same command as before:**

```
$ java -classpath classpath Client hostname 1099
```

   When the connection is established, you should see the periodic heartbeat messages in the debug output of the agent.

3. **This time, type** `<Control-C>` **in the *client's* window to stop the connector client and the manager application. This simulates a broken communication channel as seen by the connector server in the agent.**

   After the heartbeat retry timeout elapses in the agent, you should see the lost connection message in the heartbeat debugging output of the agent.

4. **Type** `<Control-C>` **in the agent window to stop the agent application and end the example.**

CHAPTER **9**

MBean Proxies

In the previous topic, we saw how to access a remote agent and interact with its MBeans through connectors. The Java Dynamic Management Kit provides additional functionality which makes the remoteness of an agent and the communication layer even more transparent: *proxy objects* for registered MBeans.

A proxy is an object instance which represents an MBean, which mirrors the methods of the MBean, and whose methods are invoked directly by the caller. The proxy transmits requests to the MBean, through the MBean server, possibly through a connector, and returns any responses to the caller. Proxy objects can also register listeners for notifications that the MBean may emit.

The advantage of a proxy object is that it allows applications to have an instance of an object which represents an MBean, instead of accessing the MBean's management interface through methods of the MBean server or through a connector client. This can simplify both the conceptual design of a management system and the source code needed to implement that system.

Note - While the concept of proxy objects remains unchanged from previous versions of the Java Dynamic Management Kit, the binding mechanisms have been modified in version 4.2.

The old proxy creation and binding methods were not thread-safe and are now deprecated. Callers are now responsible for instantiating and binding the proxy objects that they wish to use; see "The `Proxy` Interface" on page 132 for details.

In addition, the new design allows proxies to be bound on both the agent and manager sides for a more symmetric usage.

The code samples in this topic are taken from the files in the `SimpleClients` example directory located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Contents:

- "The Proxy Mechanism" on page 130 describes how proxy objects are implemented and the interfaces on which they rely.
- "Standard MBean Proxies" on page 133 shows how to generate proxy objects for standard MBeans and run the corresponding example.
- "Generic Proxies" on page 138 describes the proxy objects for dynamic MBeans and shows how to run the corresponding example.
- Finally, "Proxies for Java DMK Components" on page 142 explains how to use the pre-generated proxy objects provided with the product.

The Proxy Mechanism

Proxy objects simplify the interactions between an application and the MBeans it wants to manage. The purpose of a proxy is to invoke the methods that access the attributes and operations of an MBean, through its MBean server. These method calls can be rather tedious to construct at every invocation, so the proxy performs this task for the caller.

For example to invoke an operation on an MBean, an application must call the `invoke` method of the MBean server and provide the MBean's object name, the operation name string, an array of parameter objects, and a signature array. The proxy is a class which codes this whole sequence, meaning that the application can call the `reset` method directly on the proxy instance. And because the code of a proxy class is deterministic, it can be generated automatically using the `proxygen` tool.

Conceptually, a proxy instance makes the MBean server and a protocol connector completely transparent. Except for MBean registration and connector connection phases, all management requests on MBeans can be fully served through proxies, with identical results. However, all functionality of the Java Dynamic Management Kit is available without using proxies, so their usage is never mandatory.

By definition, a proxy object has the same interface as its MBean: the proxy can be manipulated as if it were the MBean instance, except that all requests are transmitted through the MBean server to the actual MBean instance for processing. A standard MBean proxy exposes getters, setters, and operation methods. It may also register listeners for all notifications broadcast by their corresponding MBean. A dynamic MBean proxy, also known as a *generic proxy*, exposes generic methods which are identical to those of the `DynamicMBean` interface.

In addition, standard proxies are commonly called *proxy MBeans*, because they are themselves MBeans. They are generated with an MBean interface, and can therefore be registered as standard MBeans in an MBean server. This feature allows one agent to expose resources whose MBeans are actually located in another agent. An equivalent functionality is covered in the topic of "Cascading Agents" in the lesson on "Agent Services".

Local and Remote Proxies

Proxies may also be bound to objects called *handlers* which necessarily implement the `ProxyHandler` interface. The methods of this interface are a subset of MBean server methods, as listed in Table 8–1 (see “The `RemoteMBeanServer` Interface” on page 113). These are the only methods that a proxy needs to in order to access its corresponding MBean and fulfill all management requests.

In the Java Dynamic Management Kit, the `RemoteMBeanServer` interface extends the `ProxyHandler` interface, meaning that proxy objects may be bound to any of the connector clients. These are called *remote proxies*, because they are instantiated in an application which is distant from the agent and its MBean instances.

As a new feature in version 4.2 of the product, the implementation of the `MBeanServer` interface also implements the `ProxyHandler` interface, so that proxy objects may be bound to the MBean server itself. These are called *local proxies* because they are located in the same application as their corresponding MBeans. Local proxies help preserve the management architecture by providing the simplicity of performing direct method calls on MBeans, while still routing all operations through the MBean server.

The symmetry of remote and local proxies complements the symmetry that allows management components to execute either in an agent or in a management application. Provided that all proxy classes are available, management components which use MBean proxies may be instantiated in an agent and rely on the MBean server or may be instantiated in a remote manager where they interact with a connector client. Except for communication delays, the results of all operations will be identical, and the same notifications will be received, whether obtained through a local proxy or a remote proxy (see “Adding a Listener Through the Connector” on page 148).

The following diagram shows local proxies which are instantiated in an agent and bound to the MBean server, and the same classes instantiated as remote proxies in a management application and bound to the connector client. Management components located in either the agent or management application can interact with the local or remote proxies, respectively. Management components may also access the MBean server or the connector client directly, regardless of whether proxies are being used.

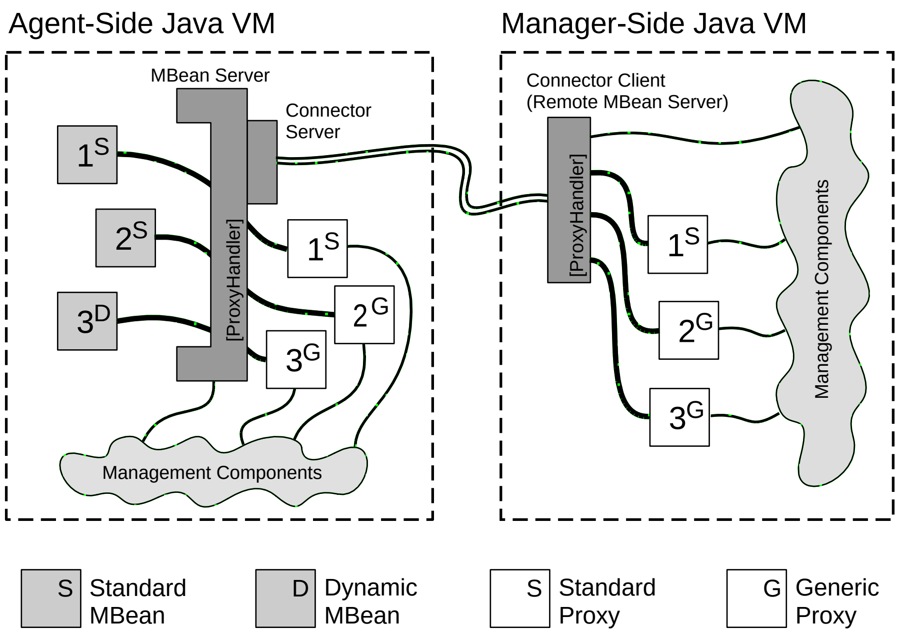


Figure 9–1 Interacting with Local and Remote Proxies

This diagram shows all possible relations between management components, proxies and their MBeans. Standard proxies can only represent a specific standard MBean, and generic proxies may represent any standard or dynamic MBean. In a typical management scenario, the management components will be located in only one application, and for simplicity, they will rarely instantiate more than one type of proxy.

Throughout the rest of this topic, we do not distinguish between local and remote proxies. A proxy object, either standard or generic, is used in exactly the same way regardless of whether it is instantiated locally or remotely.

The `Proxy` Interface

In addition to the methods for accessing the attributes and operations of its MBean, all proxies implement the methods of the `Proxy` interface. This interface contains the methods for binding the proxy instance to the proxy handler which can fulfill its requests. The `setServer` method binds the proxy to the handler object. Setting the server to `null` effectively unbinds the proxy object. The result of the `getServer` method can indicate whether or not a proxy is bound and if so, it will return a reference to the handler object.

Because of the new design of the proxy mechanism, many methods in the `Proxy` interface are deprecated as of version 4.2 of the product. The functionality of the deprecated methods is preserved in the three non-deprecated methods, listed in the following table. See the Javadoc API of the Proxy interface for details.

TABLE 9–1 Non-Deprecated Methods of the Proxy Interface

ObjectInstance	**getMBeanObjectInstance**()
ProxyHandler	**getServer**()
void	**setServer**(ProxyHandler server)

Standard proxies may also implement the NotificationBroadcasterProxy interface if their corresponding MBean is a notification proxy. This interface contains the same addNotificationListener and removeNotificationListener methods that the MBean implements from the NotificationBroadcaster interface.

Applications that use proxies therefore have two ways to detect notification broadcasters. The first way relies on the implementation of the NotificationBroadcasterProxy interface which can be detected in the proxy's class inheritance. The second and more standard way is to look at the notifications listed in the MBean's metadata obtained by the getMBeanInfo method either from the server or through the proxy.

Generic proxies do not implement the NotificationBroadcasterProxy interface, so callers must use the MBean metadata for detecting broadcasters. In addition, generic proxies cannot register notification listeners, callers must do this directly through the server.

Standard MBean Proxies

A standard MBean proxy class is specific to its corresponding MBean class. Furthermore, a proxy instance is always bound to the same MBean instance, as determined by the object name passed to the proxy's constructor. Binding the proxy to its ProxyHandler object can be done through a constructor or set dynamically through the methods of the Proxy interface.

The methods of a standard proxy have exactly the same signature as those of the corresponding standard MBean. Their only task is to construct the complete management request, which necessarily includes the object name, and to transmit it to the MBean server or connector client. They also return any result directly to the caller.

Because the contents of all proxy methods are determined by the management interface of the MBean, the proxy classes can be generated automatically.

Generating Proxies for Standard MBeans

The proxygen tool provided with the Java Dynamic Management Kit takes the class files of an MBean and its MBean interface, and generates the Java source code files of its corresponding proxy object and proxy MBean interface. You then need to compile the two proxy files with the javac command and include the resulting class files in your application's classpath.

The proxygen command is fully documented in the *Java Dynamic Management Kit 4.2 Tools Reference* guide, and in the Javadoc API for the ProxyGen class. Its command line options allow you to generate read-only proxies where all setter methods are suppressed and to define a package name for your proxies. For the purpose of the examples, we generate the default proxies without any of these options: see the section on "Running the Standard Proxy Example" on page 137.

The following code sample shows part of the code generated for the SimpleStandard MBean used in the SimpleClients examples.

CODE EXAMPLE 9–1 Code Generated for the SimpleStandardProxy Class

```
public java.lang.String getState()
  throws InstanceNotFoundException, AttributeNotFoundException,
  ReflectionException, MBeanException {

    return ((java.lang.String)server.getAttribute(
                objectInstance.getObjectName(), "State"));
  }

public  void setState(java.lang.String value)
  throws InstanceNotFoundException, ReflectionException,
  AttributeNotFoundException,InvalidAttributeValueException,
  MBeanException {

    server.setAttribute(objectInstance.getObjectName(),
                        new Attribute("State",value));
}

public void reset()
  throws InstanceNotFoundException, ReflectionException,
  MBeanException {

  Object result;
  result= server.invoke(objectInstance.getObjectName(), "reset",
                        null, null);
}
```

You are free to modify the generated code if you wish to customize the behavior of your proxies. However, customization is not recommended if your MBean interfaces are still evolving, because all modifications will need to be redone every time the proxies are generated.

Using Standard MBean Proxies

Once the proxies are generated and available in your application's classpath, their usage is straightforward. For each of the proxy objects it wishes to use, the application needs to instantiate its proxy class and then bind it to a `ProxyHandler` object. The application is responsible for creating and binding all of the proxies that it wishes to use, and it must unbind and free them when they are no longer needed.

Note - In previous versions of the Java Dynamic Management Kit, the connector client handled the creation of proxy instances and insured that only one proxy object could exist for each MBean. As of this version (4.2) of the product, connector clients no longer instantiate nor control proxy objects. The corresponding methods of the `RemoteMBeanServer` interface are now deprecated.

Similarly, the previous binding methods in the Proxy interface are deprecated in favor of the new `setServer` and `getServer` methods. This change is necessary so that proxies may be bound to any `ProxyHandler` object, to allow for both local and remote proxies.

All parameters for binding the proxy can be given in its constructor, which makes it very simple to instantiate and bind a proxy in one step.

CODE EXAMPLE 9–2 Instantiating and Binding a Proxy In One Step

```
String mbeanName = "SimpleStandard";

// build the MBean ObjectName instance
ObjectName mbeanObjectName = null;
String domain = connectorClient.getDefaultDomain();
mbeanObjectName = new ObjectName( domain + ":type=" + mbeanName );

// create the MBean in the MBeanServer of the agent
String mbeanClassName = mbeanName;
ObjectInstance mbeanObjectInstance =
    connectorClient.createMBean( mbeanClassName, mbeanObjectName );

// create and bind a proxy MBean on the client side
// that corresponds to the MBean just created in the agent
Proxy mbeanProxy = new SimpleStandardProxy(
                           mbeanObjectInstance, connectorClient );

echo("\tPROXY CLASS NAME  = " +
      mbeanProxy.getClass().getName() );
echo("\tMBEAN OBJECT NAME = " +
      mbeanProxy.getMBeanObjectInstance().getObjectName() );
```

(continued)

```
echo("\tCONNECTOR CLIENT  = " +
      mbeanProxy.getServer().getClass().getName() );
```

If the class name of your proxy is not known at compile time, you will have to instantiate its class dynamically. In the following code, we obtain the proxy class name which corresponds to an MBean, and we call its first constructor. This must be the constructor which takes an `ObjectInstance` identifying the MBean, and we must dynamically build the call to this constructor. We then call the `setServer` method to bind the new proxy instance.

CODE EXAMPLE 9–3 Instantiating and Binding a Proxy Class Dynamically

```
// Get the class name of the MBean’s proxy
Class proxyClass = Class.forName(
    connectorClient.getClassForProxyMBean( mbeanObjectInstance ));

// Find the constructor with takes an ObjectInstance parameter
Class[] signature = new Class[1];
signature[0] = Class.forName("javax.management.ObjectInstance");
Constructor proxyConstr = proxyClass.getConstructor( signature );

// Call the constructor to instantiate the proxy object
Object[] initargs = new Object[1];
initargs[0] = mbeanObjectInstance;
Proxy proxy2 = (Proxy) proxyConstr.newInstance( initargs );

// Bind the proxy
proxy2.setServer( connectorClient );

echo("\tPROXY CLASS NAME  = " +
      proxy2.getClass().getName());
echo("\tMBEAN OBJECT NAME = " +
      proxy2.getMBeanObjectInstance().getObjectName());
echo("\tCONNECTOR CLIENT  = " +
      proxy2.getServer().getClass().getName());

// We no longer need proxy2, so we unbind it
proxy2.setServer( null );
```

Once a proxy is bound, you can access the attributes and operations of its MBean through direct calls to the proxy object, as shown in the following example.

CODE EXAMPLE 9–4 Accessing a Standard MBean Through Its Proxy

```
try {
    // cast mbeanProxy to SimpleStandardProxy, so we can
    // call its MBean specific methods
    SimpleStandardProxy simpleStandardProxy =
        (SimpleStandardProxy) mbeanProxy;

    [...]

    // Change the "State" attribute
    simpleStandardProxy.setState("new state from client");

    // Get and display the new attribute values
    echo("\tState     = \"" + simpleStandardProxy.getState() + "\"");
    echo("\tNbChanges = " + simpleStandardProxy.getNbChanges());

    // Invoke the "reset" operation
    simpleStandardProxy.reset();
    [...]

    // We are done with the MBean, so we
    // unbind the proxy and unregister the MBean
    simpleStandardProxy.setServer( null );
    connectorClient.unregisterMBean( mbeanObjectName );

} catch (Exception e) {
    echo("\t!!! Error accessing proxy for " +
          mbeanProxy.getMBeanObjectInstance().getObjectName() );
    e.printStackTrace();
}
```

Running the Standard Proxy Example

The *examplesDir*/`SimpleClients` directory contains all of the files for the `ClientMBeanProxy` application which demonstrates the use of standard MBean proxies.

If you haven't done so already, compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/SimpleClients/
$ javac -classpath classpath *.java
```

Before running the example, you must also generate the proxy MBeans classes and compile them as well. From the same directory as above, type the following commands:

```
$ installDir/SUNWjdmk/jdmk4.2/JDKversion/proxygen SimpleStandard
$ javac -classpath classpath SimpleStandardProxy.java
```

▼ Instructions

1. **Launch the base agent in a terminal window with the following command:**

   ```
   $ java -classpath classpath BaseAgent
   ```

   The agent creates the RMI connector server to which the client application will establish a connection, and then it waits for management operations.

2. **Wait for the agent to be completely initialized, then, in another window on the same host, launch the management application with the following command:**

   ```
   $ java -classpath classpath ClientMBeanProxy
   ```

3. **Press** `<Enter>` **in the manager window to step through the example.**

   As seen in the code examples, the client application instantiates the proxy objects to access the MBean it has created in the base agent.

4. **Press** `<Enter>` **one last time to exit the manager application, but leave the base agent running for the next example.**

Generic Proxies

Because dynamic MBeans only expose their management at runtime, it is impossible to generate a specific proxy object for them. Instead, we use the `GenericProxy` object which can be bound to any dynamic MBean, and whose generic methods take the name of the attribute or operation being accessed. Therefore, to access a dynamic MBean through generic proxy you invoke exactly the same methods as those of the `DynamicMBean` interface.

Just as the MBean server's generic methods can access both standard and dynamic MBeans, generic proxies can also be bound to standard MBeans. You lose the specificity and simplicity of a standard proxy, but a generic proxy is always available in any Java Dynamic Management application, and it never needs regenerating.

The management application in this example shows how generic proxies can be used to access both standard and dynamic MBeans. The application contains the following subroutine which takes the class name of an MBean, creates that MBean in the agent, and instantiates a generic proxy to access the MBean.

In fact, the subroutine instantiates two generic proxies for the MBean, one using the `GenericProxy` class constructor which also binds the proxy, the other bound in a second, separate call to its `setServer` method. This demonstrates that it is possible to have two distinct proxy instances coexisting simultaneously for the same MBean.

CODE EXAMPLE 9–5

```
private void doGenericProxyExample( String mbeanName ) {

    try {
        // build the MBean ObjectName instance
        ObjectName mbeanObjectName = null;
        String domain = connectorClient.getDefaultDomain();
        mbeanObjectName = new ObjectName( domain +
                                          ":type=" + mbeanName);

        // create the MBean in the MBeanServer of the agent
        String mbeanClassName = mbeanName;
        ObjectInstance mbeanObjectInstance =
            connectorClient.createMBean( mbeanClassName, mbeanObjectName );

        // create and bind a generic proxy instance for the MBean
        Proxy proxy = new GenericProxy(
                              mbeanObjectInstance, connectorClient );

        echo("\tPROXY CLASS NAME  = " +
              proxy.getClass().getName());
        echo("\tMBEAN OBJECT NAME = " +
              proxy.getMBeanObjectInstance().getObjectName());
        echo("\tCONNECTOR CLIENT  = " +
              proxy.getServer().getClass().getName());

        // An alternate way is to first instantiate the generic proxy,
        // and then to bind it to the connector client:
        Proxy proxy2 = new GenericProxy( mbeanObjectInstance );
        proxy2.setServer( connectorClient );

        echo("\tPROXY CLASS NAME  = " +
              proxy2.getClass().getName());
        echo("\tMBEAN OBJECT NAME = " +
              proxy2.getMBeanObjectInstance().getObjectName());
        echo("\tCONNECTOR CLIENT  = " +
              proxy2.getServer().getClass().getName());

        // we no longer need proxy2, so we unbind it
        proxy2.setServer(null);

        [...] // Accessing the MBean through its generic proxy (see below)
```

(continued)

(Continuation)

```
        // When done with the MBean, we unbind the proxy
        // and unregister the MBean
        //
        proxy.setServer(null);
        connectorClient.unregisterMBean( mbeanObjectName );

    } catch (Exception e) {
        echo("\t!!! Error instantiating or binding proxy for " +
              mbeanName );
        e.printStackTrace();
    }
}
```

The standard and dynamic MBean classes used in this example have exactly the same management interface, and therefore, we can use the same code to access both of them. The manager application does this by calling the above subroutine twice, once with the class name of the standard MBean, once with that of the dynamic MBean:

```
manager.doGenericProxyExample("SimpleStandard");
manager.doGenericProxyExample("SimpleDynamic");
```

Because the two MBeans have the same behavior, they will produce the same results when accessed through their proxy. The only difference is that the dynamic MBean can expose a description of its management interface in its `MBeanInfo` object. As expected, accessing a standard MBean through a generic proxy also produces the same result as when it is accessed through a standard proxy (compare the following with Code Example 9–4).

CODE EXAMPLE 9–6 Accessing an MBean Through its Generic Proxy

```
try {

    // cast Proxy to GenericProxy
    GenericProxy genericProxy = (GenericProxy) proxy;

    // Get the MBean's metadata through the proxy
    MBeanInfo info = genericProxy.getMBeanInfo();

    // display content of the MBeanInfo object
    echo("\nCLASSNAME: \t"+ info.getClassName() );
    echo("\nDESCRIPTION: \t"+ info.getDescription() );
    [...] // extract all attribute and operation info
```

(continued)

```
    // Change the "State" attribute
    Attribute stateAttr = new Attribute( "State", "new state from client");
    genericProxy.setAttribute( stateAttr );

    // Get and display the new attribute values
    String state =
        (String) genericProxy.getAttribute("State");
    Integer nbChanges =
        (Integer) genericProxy.getAttribute("NbChanges");
    echo("\tState     = \"" + state + "\"");
    echo("\tNbChanges = " + nbChanges);

    // Invoke the "reset" operation
    Object[] params = new Object[0];
    String[] signature = new String[0];
    genericProxy.invoke("reset", params, signature );

} catch (Exception e) {
    echo("\t!!! Error accessing proxy for " +
          proxy.getMBeanObjectInstance().getObjectName() );
    e.printStackTrace();
}
```

The above code listing shows how the generic methods are called with the names of attributes and operations, and how required parameters can be constructed.

Running the Generic Proxy Example

The `ClientGenericProxy` application, also in the *examplesDir*/`SimpleClients` directory, demonstrates the use of generic proxies.

If you haven't done so already, compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/SimpleClients/
$ javac -classpath classpath *.java
```

Because generic proxies do not need to be generated, this example does not need the proxygen tool. The `GenericProxy` class is available in the usual classpath for the product's runtime libraries.

▼ Instructions

1. **If it is not already running on your host, launch the base agent in a terminal window with the following command:**

   ```
   $ java -classpath classpath BaseAgent
   ```

   The agent creates the RMI connector server to which the client application will establish a connection, and then it waits for management operations.

2. **Wait for the agent to be completely initialized, then launch the management application in another window on the same host:**

   ```
   $ java -classpath classpath ClientGenericProxy
   ```

3. **Press `<Enter>` in the manager window to step through the example.**

   As seen in the code examples, the client application instantiates generic proxy objects to access both a standard and dynamic MBean that it creates in the base agent. The only difference between the two is the user-provided information available in the dynamic MBean's metadata.

4. **Press `<Enter>` in both windows to exit the base agent and manager applications.**

Proxies for Java DMK Components

Most components of the Java Dynamic Management Kit product are MBeans and can therefore also be managed through local or remote proxies. Nearly all are standard MBeans, so their corresponding standard proxies are provided with the product. The Java source code for all component proxy classes can be found in the `JdmkProxyMBeans` directory located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Note - The HTML protocol adaptor is implemented as a dynamic MBean and therefore cannot have a standard proxy. You must use a generic proxy if you wish to access the HTML adaptor through a proxy object.

Of course, all other Java DMK MBean components may also be accessed through generic proxies, although their standard proxies provide more abstraction of the MBean server and a greater simplification of your application's code.

The proxy classes have been generated by the `proxygen` tool with full read-write access of all attributes. See the chapter on the `proxygen` tool in the *Java Dynamic Management Kit 4.2 Tools Reference* guide.

Proxy Packages

Like all other classes, proxies may contain a `package` statement. The package for a component proxy class depends upon the package of the component:

- The proxy classes for Java DMK components in the `javax.management` package and its `javax.management.*` subpackages do not have a package statement.

  Their class files may be located in any directory which can then be added to your application's classpath.

- The proxy classes for all other components belong to the same class as the component itself. For example, the proxy classes for the `RmiConnectorServer` component are declared in the `com.sun.jdmk.comm` package.

  Their class files should be contained in the corresponding file hierarchy, whose root can be added to your application's classpath. Therefore, the class files of the `RmiConnectorServerProxy` should be located in a directory called *packageRoot*`/com/sun/jdmk/comm/`.

Compiling the Proxy Classes

To use the standard proxies for the product components, you must first compile them with the `javac` command and then place the classes you need in the classpath of your agent or management application. If you compile the proxy classes individually, be sure to compile the proxy's MBean interface before its corresponding proxy class.

Because of the package statement in proxy classes, we recommend using the following commands:

```
$ cd examplesDir/JdmkProxyMBeans/
$ javac -d packageRoot -classpath classpath *ProxyMBean.java *Proxy.java
```

In this command, the *classpath* must contain the current directory and the classpath of the Java DMK runtime libraries (usually in *installDir*`/SUNWjdmk/jdmk4.2/`*JDKversion*`/lib/jdmkrt.jar`). The `-d` option of the `javac` compiler creates the necessary directories in the given *packageRoot* for each class's package. The *packageRoot* is usually the current directory (`.`), or you may directly specify a target directory in your application’s classpath.

CHAPTER **10**

Notification Forwarding

In this topic, we expand the notification mechanism to the manager side, looking at how remote applications receive notifications. Notifications are forwarded to a manager through existing structures of the Java Dynamic Management architecture: MBeans, the MBean server, and the connectors. Notably, this implies that the notification mechanism is designed to forward only notifications from registered MBeans on the agent side to proper listeners on the manager side.

As with the other management operations, listening for notifications is nearly as simple to do in a management application as it is to do locally in an agent. The interface of the connector client hides all communication issues, so that listeners may be registered through the connector or directly with existing proxy MBeans.

The code samples in this topic are taken from the files in the `Notification` example directory located in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

Registering Manager-Side Listeners

Like the other structures of the Java Dynamic Management Kit, the notification mechanism is designed to be homogeneous from the agent to the manager side. For this reason, notification objects and listener interfaces in manager applications are identical to those on the agent side.

The symmetry of the interfaces also means that code for listeners can easily be reused without modification in either agent or manager applications. Listeners in managers are similar to those in agents, and they could even be identical objects in some management solutions. However, chances are that manager-side listeners will want to receive different notifications and take different actions than their agent-side peers.

The Agent-Side Broadcaster

The notification broadcasters are MBeans registered in an agent's MBean server to which our management application will need to connect. Only notifications sent by registered MBeans can be forwarded to manager applications, and a manager-side listener can receive them only by registering through a connector client or a proxy object.

Other notification broadcasters may exist independently in the manager application, but listeners will need to register directly with these local broadcasters. Nothing prevents a listener object from registering with both a connector client or proxy for remote notifications and with a local broadcaster.

The code example below shows how the sample `NotificationEmitter` MBean will send notifications (the code for its `MBean` interface has been omitted). It extends the `NotificationBroadcasterSupport` class to reuse all of its listener registration facilities. It only contains one operation which can be called by our manager to trigger any number of notifications.

CODE EXAMPLE 10–1 The Agent-Side Broadcaster MBean

```
import javax.management.MBeanNotificationInfo;
import javax.management.NotificationBroadcasterSupport;
import javax.management.Notification;

public class NotificationEmitter
    extends NotificationBroadcasterSupport
    implements NotificationEmitterMBean {

    // Just to make the inheritance explicit
    public NotificationEmitter() {
        super();
```

(continued)

```
    }

    // Provide details about the notification type and class that is sent
    public MBeanNotificationInfo[] getNotificationInfo() {

        MBeanNotificationInfo[] ntfInfoArray = new MBeanNotificationInfo[1];

        String[] ntfTypes = new String[1];
        ntfTypes[0] = myType;

        ntfInfoArray[0] = new MBeanNotificationInfo( ntfTypes,
            "javax.management.Notification",
            "Notifications sent by the NotificationEmitter");
        return ntfInfoArray;
    }

    // The only operation: sends any number of notifications
    // whose sequence numbers go from 1 to "nb"
    public void sendNotifications( Integer nb ) {

        for (int i=1; i<=nb.intValue(); i++) {
            sendNotification(new Notification(myType, this, i));
        }
    }
    private String myType = "notification.my_notification";
}
```

Our MBean invents a notification type string and exposes this information through the `getNotificationInfo` method. To demonstrate the forwarding mechanism, we are more interested in the sequence number: this will allow us to identify the notifications as they are received in the manager.

This MBean demonstrates that the broadcaster has total control over the contents of its notifications. Constructors for the `Notification` object allow you to specify all of the fields, even ones such as the time stamp. In this example, we control the sequence number, and our chosen policy is to reset the sequence number to 1 with every call to the operation. Of course, you are free to choose the notification contents, including the time-stamping and sequence-numbering policies that fit your management solution.

Note - Due to possible loss in the communication layer and the inherent indeterminism of thread execution, the notification model does not guarantee that remote notifications will be received nor that their sequence will be preserved. If notification order is critical to your application, your broadcaster should set the sequence numbers appropriately, and your listeners should sort the received notifications.

The Manager-Side Listener

In our simple example, the `Client` class itself is the listener object. Usually, a listener would be a separate instance of a special listener class and depending on the complexity of the manager, there might be several classes of listeners, each for a specialized category of notifications.

CODE EXAMPLE 10–2 The Manger-Side Listener

```
public class Client implements NotificationListener {

    [...] // Constructor omitted

    // Implementation of the NotificationListener interface
    //
    public void handleNotification(Notification notif, Object handback) {

        System.out.println("Client: received a notification of type "
            + notif.getType() + "\nwith the sequence number "
            + notif.getSequenceNumber());
    }
    [...]  // main omitted
}
```

As explained in the notification mechanism "Overview" on page 94, a listener on the agent side is typically an MBean which receives notifications about the status of other MBeans and then processes or exposes this information in some manner. Only if a key value or some management event is observed will this information be passed to a listening manager, probably by sending a different notification.

In this manner, the notification model reduces the communication that is necessary between agents and managers. Your management solution determines how much decisional power resides in the agent and when situations are escalated. These parameters will affect your design of the notification flow between broadcasters, listeners, agents, and managers.

The usual role of a manager-side listener is to process the important information in a notification and take the appropriate action. As we shall see, our notification example is much simpler. Our goal is not to construct a real-world example, but to demonstrate the mechanisms that are built into the Java Dynamic Management Kit.

Adding a Listener Through the Connector

By extension of the `ClientNotificationHandler` interface, the `RemoteMBeanServer` interface exposes methods for adding and removing listeners. The signatures of these methods are identical to those of the agent-side

MBeanServer interface. The only difference is that they are implemented in the connector client classes which make the communication protocol transparent.

Our manager application uses the RMI protocol connector. After creating the connector client object, we use the methods of its RemoteMBeanServer interface to create our broadcaster MBean and then register as a listener to this MBean's notifications.

CODE EXAMPLE 10–3 Adding a Listener through the Connector

```
// Use RMI connector on port 8086 to communicate with the agent
System.out.println(">>> Create an RMI connector client");
RmiConnectorClient connectorClient = new RmiConnectorClient();

// agentHost was read from the command line or defaulted to localhost
RmiConnectorAddress rmiAddress = new RmiConnectorAddress(
    agentHost, 8086, com.sun.jdmk.ServiceName.RMI_CONNECTOR_SERVER);
connectorClient.connect(rmiAddress);

// Wait 1 second for connecting
Thread.sleep(1000);

// Create the MBean in the agent
ObjectName mbean = new ObjectName ("Default:name=NotificationEmitter");
connectorClient.createMBean("NotificationEmitter", mbean);

// Now add ourselves as the listener (no filter, no handback)
connectorClient.addNotificationListener(mbean, this, null, null);
```

You can see how similar this code is to the agent application by comparing it with the code example for "Adding a Listener Through the MBean Server" on page 98.

If you have generated and instantiated proxy MBeans for your broadcaster MBeans, you can also register through the addNotificationListener method that they expose. When generating proxy classes with the proxygen tool, MBeans which implement the NotificationBroadcaster interface will have proxy classes which implement the NotificationBroadcasterProxy interface.

Again, the method signatures defined in a proxy MBean are identical to those of the MBeanServer or NotificationBroadcasterClient interfaces for adding or removing listeners: see the code example for "Adding a Listener Directly to an MBean" on page 103. Listeners added through a proxy MBean will received the same notifications as listeners added to the same MBean through the interface of the connector client.

Note - Following the Java programming model, the connector client limits its resource usage by only running one thread to notify all of its listeners. This thread calls all of the handler callback methods that have been added through this connector. Therefore, the callbacks should return quickly and use safe programming to avoid crashing the connector client.

Push Mode

Because the broadcaster and the listener are running on separate machines or in separate virtual machines on the same host, their notifications must be forwarded from one to the other. The mechanism for doing this is completely transparent to the users of the Java Dynamic Management Kit components.

Briefly, the connector client instructs the connector server to add its own agent-side listener to the designated broadcaster using the methods of the MBeans server. Then, the connector server implements a buffering cache mechanism to centralize notifications before serializing them to be forwarded to the connector client. By design, it is the connector client in the manager application that controls the buffering and forwarding mechanism for a connection.

The following diagram summarizes the notification forwarding mechanism and its actors. In particular, it shows how the connector server uses internal listener instances to register locally for MBean notifications, even if this mechanism is completely hidden from the user. The path of listener registration through calls to the `addNotificationListener` method of the various interfaces is paralleled by the propagation of notifications through calls to the listeners' `handleNotification` method.

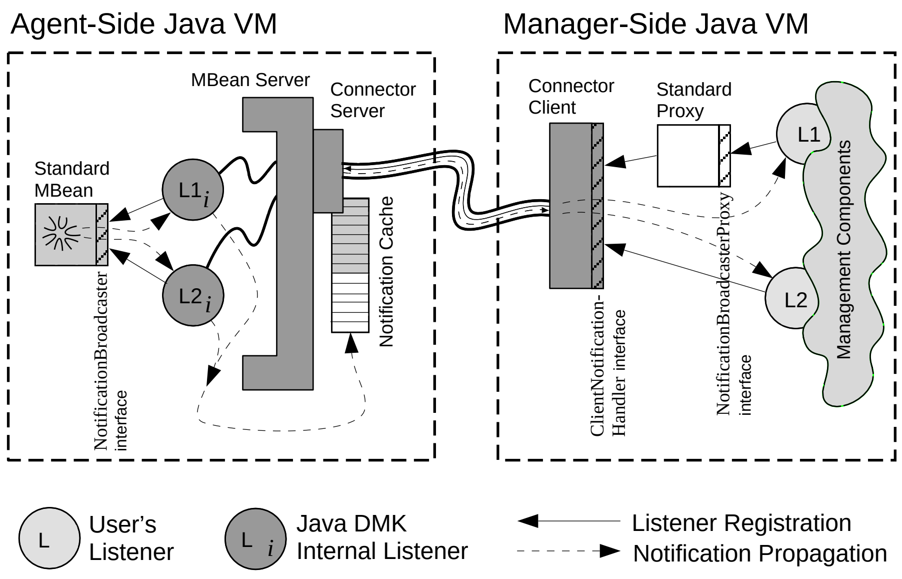


Figure 10–1 Notification Forwarding Internals

Neither the broadcaster nor the listener need to implement any extra methods, or even be aware that the other party is remote. Only the designer needs to be aware of communication issues such as delays: you can't expect the listener to be invoked instantaneously after a remote broadcaster sends a notification.

The forwarding mechanism allows you to configure how and when notifications are forwarded. This allows you to optimize the communication strategy between your agents and managers. There are two basic modes for notification forwarding: push mode and pull mode. A notification in the connector server's cache is either pushed to the manager at the agent's initiative, or pulled by the manager at its own initiative.

The push mode for notification forwarding is the simplest because it implements the expected behavior of a notification. When a notification is sent from an MBean to its listener, it is immediately pushed to the manager-side where the listener's handler method is called. There is no delay in the caching, and if the communication layer is quick enough, the listener is invoked almost immediately after the notification is sent.

Push mode is the default forwarding policy of a newly instantiated connector client.

In our manager example, we explicitly set the connector client in push mode and then trigger the agent-side notifications.

CODE EXAMPLE 10–4 Switching to the Notification Push Mode

```
System.out.println("\n>>> Set notification forward mode to PUSH.");
connectorClient.setMode(ClientNotificationHandler.PUSH_MODE);

System.out.println(">>> Have our MBean broadcast 10 notifications...");
```

(continued)

```
params[0] = new Integer(10);
signatures[0] = "java.lang.Integer";
connectorClient.invoke(mbean, "sendNotifications", params, signatures);

System.out.println(">>> Done.");
System.out.println(">>> Receiving notifications...\n");

// Nothing to do but wait while our handler receives the notifications
Thread.sleep(2000);
```

The connector client exposes the methods for controlling the agent's notification buffer. This caching buffer is not used in push mode, so these methods do not affect pushed notifications. The methods do however set internal variables that will be taken into account if and when pull mode is enabled. Future versions of the product may implement push-mode buffering to provide added functionality.

The advantage of push mode is that it works without any further intervention: notifications eventually reach their remote listeners. Push mode works when the communication layer and the listener's processing capacity are adapted to the notification emission rate, or more specifically to the potential emission rate. Since all notifications are immediately sent to the manager hosts, a burst of notifications will cause a burst of traffic that may or may not be adapted to the communication layer.

If your communication layer is likely to be saturated, either your design should control broadcasters to prevent bursts of notifications, or you should use the pull mode which has this control functionality built-in. The push mode is ideal if you have reliable and fast communication between your agents and your managers. You may also dynamically switch between modes, allowing a management application to fine-tune its communication policy depending on the number of notifications that must be handled.

Pull Mode

In pull mode, notifications are not immediately sent to their remote listeners. Rather, they are stored in the connector server's internal buffer until the connector client requests that they be forwarded. Instead of being sent individually, the notifications are grouped to reduce the load on the communication layer. Pull mode has the following settings that let the manager define the notification forwarding policy:

- A period for automatic pulling

- The size of the agent-side notification buffer (also called the cache)
- The policy for discarding notifications when this buffer is full

For a given connection, there is one cache for all listeners, not one cache per listener. This cache therefore has one buffering policy whose settings are controlled through the methods exposed by the connector client. The cache buffer contains an unpredictable mix of notifications in transit to all manager-side listeners added through a given connector client or through one of its bound proxy MBeans. The buffer operations such as pulling or overflowing apply to this mix of notifications, not to any single listener's notifications.

Periodic Forwarding

Pull mode forwarding is necessarily a compromise between receiving notifications in a timely manner, not saturating the communication layer, and not overflowing the buffer. Notifications are stored temporarily in the agent-side buffer, but the manager-side listeners still need to receive them. Pull mode includes automatic pulling that retrieves all buffered notifications regularly.

The frequency of the pull forwarding is controlled by the pull period expressed in milliseconds. By default, when pull mode is enabled, the manager will automatically begin pulling notifications once per second. Whether or not there are any notifications to receive depends upon events in the agent.

Our manager application sets a half-second pull period and then triggers the notification broadcaster.

CODE EXAMPLE 10–5 Pulling Notifications Automatically

```
System.out.println(">>> Set notification forward mode to PULL.");
connectorClient.setMode(ClientNotificationHandler.PULL_MODE);

// Retrieve buffered notifications from the agent twice per second
System.out.println(">>> Set the forward period to 500 milliseconds.");
connectorClient.setPeriod(500);

System.out.println(">>> Have our MBean broadcast 20 notifications...");
params[0] = new Integer(20);
signatures[0] = "java.lang.Integer";
connectorClient.invoke(mbean, "sendNotifications", params, signatures);
System.out.println(">>> Done.");

// Wait for the handler to process all notifications
System.out.println(">>> Receiving notifications...\n");
Thread.sleep(2000);
```

When notifications are pulled, *all* notifications in the agent-side buffer are forwarded to the manager and the registered listeners. It is not possible to set a limit on the number of notifications which are forwarded, except by limiting the size of the buffer (see “Agent-Side Buffering” on page 155). Even in a controlled example such as ours, the number of notifications in the agent-side buffer at each pull period is completely dependent upon the agent’s execution paths, and therefore unpredictable from the manager-side.

On-Demand Forwarding

You can disable automatic pulling by setting the pull period to zero. In this case, the connector client will not pull any notifications from the agent until instructed to do so. Use the `getNotifications` method of the connector client to pull all notifications when desired. This method will immediately forward *all* notifications in the agent-side buffer. Again, it is not possible to limit the number of notifications that are forwarded, except by limiting the buffer size.

In this example, we disable the automatic pulling and then trigger the notification broadcaster. The notifications will not be received until we request that the connector server pull them. Then, all of the notifications will be received at once.

CODE EXAMPLE 10–6 Pulling Notifications by Request

```
System.out.println(">>> Use pull mode with period set to zero.");
connectorClient.setMode(ClientNotificationHandler.PULL_MODE);
connectorClient.setPeriod(0);

System.out.println(">>> Have our MBean broadcast 30 notifications...");
params[0] = new Integer(30);
signatures[0] = "java.lang.Integer";
connectorClient.invoke(mbean, "sendNotifications", params, signatures);
System.out.println(">>> Done.");

// Call getNotifications to pull all buffered notifications from the agent
System.out.println("\n>>> Press <Enter> to pull the notifications.");
System.in.read();
connectorClient.getNotifications();

// Wait for the handler to process all notifications
Thread.sleep(100);
```

In the rest of our example, we use the on–demand forwarding mechanism to control how many notifications are buffered on the agent-side and thereby test the different caching policies.

Agent-Side Buffering

In pull mode, notifications are stored by the connector server in a buffer until they are pulled by the connector client. Any one of the pull operations, whether on-demand or periodic, empties this buffer, and it fills up again as new notifications are triggered.

By default, this buffer will grow to contain all notifications. The `ClientNotificationHandler` interface defines the static `NO_CACHE_LIMIT` field to represent an unlimited buffer size. If the notifications are allowed to accumulate indefinitely in the cache, this can lead either to an “out of memory” error in the agent application, a saturation of the communication layer, or an overload of the manager’s listeners when the notifications are finally pulled.

To change the size of the agent’s cache, call the connector client’s `setCacheSize` method. The size of the cache is expressed as the number of notifications which can be stored in its buffer. When a cache buffer of limited size is full, new notifications will overflow and be lost. Therefore, you should also choose an overflow mode when using a limited cache size. The two overflow modes are defined by static fields of the `ClientNotificationHandler` interface:

- `DISCARD_OLD` - The oldest notifications will be lost and the buffer will always be renewed with the latest notifications which have been triggered. This is the default value when a limit is first set for the cache size.
- `DISCARD_NEW` - Once the notification buffer is full, any new notifications will be lost until the buffer is emptied by forwarding the messages. The buffer will always contain the first notifications triggered after the previous pull operation.

We demonstrate each of these modes in our sample manager, by first setting the cache size and the overflow mode, then by triggering more notifications than the cache buffer can hold.

CODE EXAMPLE 10–7 Controlling the Agent-Side Buffer

```
System.out.println(">>> Use pull mode with period set to zero, " +
    "buffer size set to 10, and overflow mode set to DISCARD_OLD.");
connectorClient.setMode(ClientNotificationHandler.PULL_MODE);
connectorClient.setPeriod(0);
connectorClient.setCacheSize(10, true); // see "Buffering Specifics"
connectorClient.setOverflowMode(ClientNotificationHandler.DISCARD_OLD);

System.out.println(">>> Have our MBean broadcast 30 notifications...");
params[0] = new Integer(30);
signatures[0] = "java.lang.Integer";
connectorClient.invoke(mbean, "sendNotifications", params, signatures);
System.out.println(">>> Done.");

// Call getNotifications to pull all buffered notifications from the agent
System.out.println("\n>>> Press <Enter> to get notifications.");
System.in.read();
connectorClient.getNotifications();
```

(continued)

```
// Wait for the handler to process the 10 notifications
// These should be the 10 most recent notifications
// (the greatest sequence numbers)
Thread.sleep(100);
System.out.println("\n>>> Press <Enter> to continue.");
System.in.read();

//
 We should see that the 20 other notifications overflowed the agent buffer
System.out.println(">>> Get overflow count = " +
    connectorClient.getOverflowCount());
```

The overflow count gives the total number of notifications that have been discarded because the buffer has overflowed. The number is cumulative from the first manger-side listener registration until all of the manager's listeners have been unregistered. The manager application can modify or reset this value by calling the `setOverflowCount` method.

In our example application, we repeat the actions above, in order to cause the buffer to overflow again, but this time using the `DISCARD_NEW` policy. Again, the buffer size is ten, and there are 30 notifications. In this mode, the first 10 sequence numbers will remain in the cache to be forwarded when the manager pulls them from the agent, and 20 more will have overflowed.

Buffering Specifics

When the buffer is full and notifications need to be discarded, the time reference for applying the overflow mode is the order in which notifications have arrived in the buffer. Neither the time stamps nor the sequence numbers of the notifications are considered, since neither of these are necessarily absolute; even the sequence of notifications from the same broadcaster can be non-deterministic. And in any case, broadcasters are free to set both time stamps and sequence numbers as they see fit, or even to make them `null`.

The second parameter of the `setCacheSize` method is a boolean which determines whether or not the potential overflow of the cache is discarded when reducing the cache size. If the currently buffered notifications do not fit into the new cache size and this parameter is `true`, excess notifications are discarded according to the current overflow mode. The overflow count is also updated accordingly.

In the same situation with the parameter set to `false`, the cache will not be resized. You need to check the return value of the method when you set this parameter to false. If the cache cannot be resized because it would lead to discarded notifications, you need to empty the cache and try resizing the cache size again. To empty the cache, you can either pull the buffered notifications with the getNotifications method or discard them all by calling the connector client's clearCache method.

When the existing notifications fit within the new cache size or when increasing the cache size, the second parameter of setCacheSize has no effect.

Because several managers may connect through the same connector server object, it must handle the notifications for each separately. This implies that each connected manager has its own notification buffer and its own settings for controlling this cache. The overflow count is specific to each manager as well.

Buffering Generalities

Here we have demonstrated each setting of the forwarding mechanism independently by controlling the notification broadcaster. In practice, periodic pulling, agent-side buffering and buffer overflow may all be happening at once. And you can call getNotifications at any time to do an on-demand pull of the notifications in the agent-side buffer. You should adjust the settings to fit the known or predicted behavior of your management architecture, depending upon communication constraints and your acceptable notification loss rate.

The caching policy is completely determined by the manager application. If notification loss is unacceptable, it is the manager's responsibility to configure the mechanism so that they are pulled as often as necessary. Also, the notification mechanism can be updated dynamically. For example, the manager can compute the notification emission rate and update any of the settings (buffer size, pull period, and overflow mode) to minimize the risk of a lost notification.

Running the Notification Forwarding Example

The *examplesDir*/Notification directory contains all of the files for the broadcaster MBean, the BaseAgent application, and our Client application which is itself the listener object.

Compile all files in this directory with the javac command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/Notification/
$ javac -classpath classpath *.java
```

To run the notification forwarding example, we use the `BaseAgent` application which contains an RMI connector server.

▼ Instructions

1. **Launch the agent on another host or in another terminal window with the following command. Be sure that the classes for the `NotificationEmitter` MBean can be found in its *classpath*:**

   ```
   $ java -classpath classpath BaseAgent
   ```

2. **Wait for the agent to be completely initialized, then launch the manager in another window with the following command, where *hostname* is the name of the machine running the agent. If you launched the agent on the same machine, you can omit the *hostname*:**

   ```
   $ java -classpath classpath Client hostname
   ```

   When launched, the manager application first creates the `NotificationEmitter` MBean and then registers itself as a listener.

3. **Press `<Enter>` when the application pauses to step through the various notification forwarding situations that we have seen in this topic.**

4. **Press `<Enter>` one last time in the manager window to exit the application.**

   Leave the agent application running if you wish interact with the example through the HTML adaptor of the `BaseAgent`.

▼ Interacting with the Notification Forwarding Mechanism

1. **Launch the manager in another window with the following command, where *hostname* is the name of the machine running the agent. If you launched the agent on the same machine, you can omit the *hostname*:**

```
$ java -classpath classpath Client hostname
```

2. **Load the following URL in your browser and go to the MBean view of the `NotificationEmitter` MBean:**

   `http://hostname:8082/`

   If you get an error, you may have to switch off proxies in your browser preference settings. Any browser on your local network can also connect to this agent using this URL.

3. **When the manager application pauses for the first time, invoke the `sendNotifications` method from your browser with a small integer as the parameter.**

   You should see the listener handle your notifications in the manager's terminal window. Since the manager is still in push mode, they were forwarded immediately.

4. **Press `<Enter>` in the manger window: the manager is now in pull mode with a pull period of 500 milliseconds. Through the MBean view, send 1000 notifications.**

   If your agent's host is slow enough, or your manager's host fast enough, you may be able to see the manager pause briefly after it has processed all notifications from one period and before the next ones are forwarded.

5. **Press `<Enter>` in the manager window: the agent will now forward notifications by request. Before pressing `<Enter>` again, have the MBean send 15 notifications.**

   You should see the manager pull all of the notifications: the 30 triggered by the manager and the 15 we just triggered. They were all kept in the buffer, waiting for the manager's request to forward them. Remember that the `sendNotifications` operation resets the sequence numbering every time it is invoked.

6. **Press `<Enter>` in the manager's window: the cache size will now be set to 10 notifications and the overflow mode to `DISCARD_OLD`. Before pressing `<Enter>` again, have the MBean send 15 more notifications.**

   Only the last ten of our notifications could fit into the cache buffer, all the rest, including those already triggered by the manager, overflowed and were discarded. Press `<Enter>` to see that they are tallied in the overflow count.

7. **Press** `<Enter>` **in the manager's window: the cache size is still 10 notifications and the overflow mode will be set to** `DISCARD_NEW`**. Before pressing** `<Enter>` **again, have the MBean send only 5 more notifications.**

   The first ten of the manager-triggered notifications are received: all of the more recent notifications, including ours, overflowed the cache buffer and were lost. Press `<Enter>` to see that they are tallied in the overflow count: the 35 from the last step plus 25 more from this step, for a total of 60.

8. **Press** `<Enter>` **in the manager's window one last time to stop the** `Client` **application. Press** `<Enter>` **in the other window to stop the agent application when you are finished running the example.**

CHAPTER **11**

Access Control and Security

Whenever considering a distributed architecture, security issues are often an added factor in the complexity of the design. Not so with the Java Dynamic Management Kit, whose security features are built into the modularity of the components.

Management solutions can evolve from basic password-based protection all the way to secure connections using cryptography simply by switching protocol connectors or by adding filter components. The rest of the architecture is unchanged because it relies on the interface which is common to all connectors.

There are two categories of access-control: connection-level control through a password and request-level control through a context object. Context checkers work as filters between the connector server and the MBean server. The filter logic can be determined dynamically, based on the nature of the request and on a context object provided by the client.

Security in the communication layer is achieved through the cryptography of a Secure Socket Layer (SSL) and the HTTPS connector. Using other components of the Java platform, connectors can effectively make all open communication undecipherable.

The code samples in this topic are taken from the files in the `Context` example directory located in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

Password-Based Authentication

The simplest form of agent security is to accept management requests only if they contain a valid login identity and password. Agents recognize a given a list of login-password pairs, and managers must provide a matching login and password when they try to establish a connection.

In the Java Dynamic Management Kit, only the HTTP–based connectors support password–based authentication. The SNMP protocol adaptor also supports access control, but it is based on a different mechanism (see "Access Control Lists (ACL)" on page 280).

By default, no authentication is enabled in the HTTP-based connectors, and any manager may establish a connection. The password checking behavior is enabled by defining the list of authorized login-password pairs.

You may define this authentication information either:

- Through the constructor which takes an `AuthInfo` array parameter:
  `HttpConnectorServer(int port, AuthInfo[] authInfoList)`
- Through the methods inherited from the `GenericHttpConnectorServer` class:
  `addUserAuthenticationInfo(AuthInfo authinfo)`
  `removeUserAuthenticationInfo(AuthInfo authinfo)`

In both cases, only the agent application has access to these methods, meaning that the agent controls the authentication mechanism. As soon as an `AuthInfo` object is added to the connector server through either method, all incoming requests must provide a recognized name and password. In our example, we read the authentication information from the command line and call the `addUserAuthenticationInfo`.

CODE EXAMPLE 11–1 Implementing Password Authentication in the HTTP Connector Server

```
// Here we show the code for reading the
// id-password pairs from the command line
//
int firstarg = 0;
boolean doAuthentication = (args.length > firstarg);

AuthInfo[] authInfoList;

if (doAuthentication) {
    authInfoList = new AuthInfo[(args.length - firstarg) / 2];
    for (int i = firstarg, j = 0; i < args.length; i += 2, j++)

        authInfoList[j] = new AuthInfo(args[i], args[i + 1]);

} else
```

(continued)

(Continuation)

```
    authInfoList = null;

[...] // instantiate and register an HTTP connector server

// Define the authentication list
//
if (doAuthentication) {
    for (int i = 0; i < authInfoList.length; i++)
        http.addUserAuthenticationInfo(authInfoList[i]);
}
```

On the manager-side, identifiers and passwords are given in the address object, since authentication applies when the connection is established.

CODE EXAMPLE 11–2 Specifying the Login and Password in the HTTP Connector Server

```
// login and password were read from the command line
//
AuthInfo authInfo = null;

if (login != null) {
    authInfo = new AuthInfo( login, password );
}

// agentHost and agentPort are read from the command
// line or take on default values
//
HttpConnectorAddress addr =
    new HttpConnectorAddress(
        agentHost, agentPort, authInfo );

final RemoteMBeanServer connector =
    (RemoteMBeanServer) new HttpConnectorClient();

connector.connect( addr );
```

The connector is identified by the one `AuthInfo` object it uses to instantiate the connector address. If the agent has authentication enabled, both the login and the password much match one of the `AuthInfo` objects in the agent. If the agent does not perform authentication, providing a login and password has no effect because all connections are accepted.

If the authentication fails, the call to the `connect` method will return an exception. Normally, the client's code should catch this exception to handle this error case.

As demonstrated by the code examples, the authentication mechanism is very simple to configure. It prevents unauthorized access with very little overhead.

Note - The HTML adaptor provides a similar authentication mechanism, where the list of accepted identities is given to the server object. In the case of the HTML protocol, the web browser is the management application which must provide a login and password. The behavior is browser-dependent, but the browser will usually ask to user to type this login and password in a dialog box.

Running the Example with Authentication

The *examplesDir*/`Context` directory contains the applications which demonstrate the use of password authentication through the HTTP connector.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/Context/
$ javac -classpath classpath *.java
```

▼ Instructions

1. **Launch the agent in a terminal window, and specify a list of login-password pairs, as in the following command:**

```
$ java -classpath classpath ContextAgent  jack jill  billy bob
```

2. **Wait for the agent to be completely initialized, then launch the manager in another window with the following command:**

```
$ java -classpath classpath ContextClient -ident andy bob
```

The client application will try to establish a connection with the login `andy` and the password `bob`. The authentication mechanism will refuse the connection, an you will see the `com.sun.jdmk.comm.UnauthorizedSecurityException` raised by the connector server.

3. **Launch the manager again, this time with a valid identity:**

```
$ java -classpath classpath ContextClient -ident jack jill
```

The connection is established and you see the output from management operation in both windows.

4. **Leave both applications running for the next example.**

Context Checking

Context checking is a more advanced security mechanism that can perform selective filtering of incoming requests. The context is an arbitrary object provided by the client and used by the server to decide whether or not to allow the request.

Filtering and context checking are performed in between the communicator server and the MBean server. The mechanism relies on two objects called the `MBeanServerForwarder` and the `MBeanServerChecker`.

The Filter Mechanism

The `MBeanServerForwarder` allows the principle of *stackable MBean servers*. An `MBeanServerForwarder` implements the `MBeanServer` interface and one extra method called `setMBeanServer`. Its function is to receive requests and forward them to the designated MBean server.

The `setMBeanServer` method of a communicator server object allows you to specify the MBean server which fulfills its requests. By chaining one or more `MBeanServerForwarder` objects between a communicator server and the actual MBean server, the agent application creates a stack of objects which may process the requests before they reach the MBean server.

The `MBeanServerChecker` is an extension of the forwarder which forces each request to call a *–checker* method. By extending the `MBeanServerChecker` class and providing an implementation of the checker methods, you can define a policy for filtering requests before they reach the MBean server. As shown in the following table, checker methods apply to groups of `MBeanServer` methods.

TABLE 11–1 Filter Method Granularity for Context Checking

Filter Method	MBean Server Operations Filtered
`checkAny`	Every method of the `MBeanServer` interface
`checkCreate`	All forms of the `create` and `registerMBean` methods
`checkDelete`	The `unregisterMBean` method
`checkInstantiate`	All forms of the `instantiate` method
`checkInvoke`	The `invoke` method which handles all operation invocations
`checkNotification`	Both `addNotificationListener` and `removeNotificationListener`
`checkQuery`	Both `queryMBeans` and `queryNames`
`checkRead`	All methods which access but do not change the state of the agent: `getAttribute`, `getAttributes`, `getObjectInstance`, `isRegistered`, `getMBeanCount`, `getDefaultDomain`, `getMBeanInfo`, and `isInstanceOf`
`checkWrite`	The `setAttribute` and `setAttributes` methods

As a request passes through a stack of MBean servers, the checker methods are called to determine if the request is allowed. In order to identify the manager that issued a request, the checker may access the *operation context* of the request.

The operation context, or just context, is an object defined by the manager who seeks access through a context checker. It usually contains some description of the manager's identity. The only restriction on the context object is that it must implement the `OperationContext` interface. The context object is passed from the connector client to the connector server and is then associated with the execution of a request. Conceptually, this object is stored in the user accessible context of the thread which executes the request.

All methods in the `MBeanServerChecker` class may access the context object by calling the protected `getOperationContext` method. The methods of the context checker then implement some policy to filter requests based on the context object, the nature of the request, and the data provided in the request, such as the attribute or operation name.

The following diagram shows the paths of two requests through a stack of MBean server implementations, one of which is stopped by the context checker because it doesn't provide the correct context.

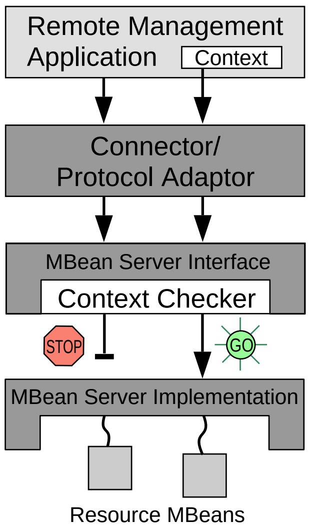

Remote Management Application
Context
Connector/ Protocol Adaptor
MBean Server Interface
Context Checker
STOP
GO
MBean Server Implementation
Resource MBeans

Figure 11–1 Context Checking in Stackable MBean Servers

Only connectors fully support the context mechanism. Their connector clients expose the methods that allow the manager to specify the context object. Existing protocol adaptors have no way to specify a context. Their requests may be filtered and checked, but their context object will always be `null`.

This functionality may still be used to implement a filtering policy, but without a context object, straightforward manager identification is not possible. However, a proprietary protocol adaptor could define some mapping to determine a context object that could be accepted by the filters.

The Context Implementation

An agent wanting to implement context checking first needs to extend the `MBeanServerChecker` class. This class retrieves the context object and decides whether any given operation is allowed.

CODE EXAMPLE 11–3 The Implementation of the Context Checker

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.QueryExp;

import com.sun.jdmk.MBeanServerChecker;
import com.sun.jdmk.OperationContext;

public class ContextChecker extends MBeanServerChecker {
```

(continued)

```
    // Constructor
    public ContextChecker(MBeanServer mbs) {
        super(mbs);
    }

    // Implementation of the abstract methods of the
    // MBeanServerChecker class: for each of the specific
    // checks, we just print out a trace of being called.
    [...]

    protected void checkWrite( String methodName,
                               ObjectName objectName) {
        System.out.println("checkWrite(\"" + methodName +
                           "\", " + objectName + ")");
    }

    protected void checkQuery( String methodName,
                               ObjectName name,
                               QueryExp query) {
        System.out.println("checkQuery(\"" + methodName +
                           "\", " + name + ", " + query + ")");
    }

    [...]

    /**
     * This is where we implement the check that requires every
     * operation to be called with an OperationContext whose
     * toString() method returns the string "nice".
     */
    protected void checkAny( String methodName,
                             ObjectName objectName ) {

        System.out.println("checkAny(\"" + methodName + "\", " +
                           objectName);
        OperationContext context = getOperationContext();
        System.out.println("  OperationContext: " + context);

        if (context == null || !context.toString().equals("nice")) {
            RuntimeException ex =
                new SecurityException("  Bad context: " + context);
            ex.printStackTrace();
            throw ex;
        }
    }

}
```

Then the agent application then needs to instantiate its context checker and stack them in between the communicator servers and the MBean server. Each communicator server would have its own stack, although filters and context checkers may be shared. The agent performs the stacking inside a synchronized block because other threads may try to do stacking simultaneously.

CODE EXAMPLE 11–4 Stacking MBean Server and Context Checkers

```
// Create MBeanServer
//
MBeanServer server = MBeanServerFactory.createMBeanServer();

/* Create context checker.  The argument to the constructor is
 * our MBean server to which all requests will be forwarded
 */
ContextChecker contextChecker = new ContextChecker( server );


[...] // Create HTTP connector server

/* Add the context checker to this HTTP connector server.
 * We point it at the context checker which already points
 * to the actual MBean server.
 * It is good policy to check that we are not sidetracking
 * an existing stack of MBean servers before setting ours.
 */
synchronized (http) {
    if (http.getMBeanServer() != server) {
        System.err.println("After registering connector MBean, " +
           "http.getMBeanServer() != " + "our MBeanServer");
        System.exit(1);
    }
    http.setMBeanServer(contextChecker);
}
```

Finally, the manager operation defines a context object class and then provides a context object instance through its connector client.

CODE EXAMPLE 11–5 Setting the Context in the Connector Client

```
/* In this example, the agent checks the OperationContext of
   each operation by examining its toString() method, so we
   define a simple implementation of OperationContext whose
   toString() is a constant string supplied in the constructor
 */
class StringOperationContext
          implements OperationContext, Cloneable {

    private String s;
```

(continued)

```
    StringOperationContext(String s) {
        this.s = s;
    }

    public String toString() {
        return s;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}

// the contextName must be provided on the command line
OperationContext context =
    new StringOperationContext(contextName);

[...]

// The context is set for all requests issued through
// the connector client; it may be changed at any time
connector.setOperationContext(context);
```

Running the Example with Context Checking

The `ContextClient` and `ContextAgent` applications in he *examplesDir*/`Context` directory also demonstrate the use of stackable MBean servers and context checking through the HTTP connector.

If you have not done so already, compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/Context/
$ javac -classpath classpath *.java
```

▼ Instructions

1. **If the agent and client applications are not already running from the previous example, type the following commands in separate windows:**

```
$ java -classpath classpath ContextAgent
$ java -classpath classpath ContextClient
```

The *classpath* should include the current directory (`.`) for both applications because they rely on classes that were compiled in this directory.

2. **Press** `<Enter>` **in the client application to trigger another set of requests.**

   The agent window displays the output of the `ContextChecker` class. We can see that the `checkAny` method verifies the "`nice`" context of every request and that the other checkers just print out their name, providing a trace of the request.

3. **Stop both applications by typing** `<Control-C>` **in each of the windows. Restart both applications, but specify a different context string for the client:**

```
$ java -classpath classpath ContextAgent
```

```
$ java -classpath classpath ContextClient -context BadToTheBone
```

   This time we see the result of a context that is not recognized. The agent raises a `java.lang.SecurityException` which is propagated to the client who then exits.

4. **Press** `<Control-C>` **in the agent window to stop the ContextAgent application.**

The HTTPS Connector

The HTTPS connector provides data encryption and certificate-based security through a Secure Socket Layer (SSL). An implementation of secure sockets is only available for the Java 2 platform. However, secure sockets are not part of the Java 2 SDK (Software Development Kit), and their libraries must be installed separately.

The Java Secure Socket Extension (JSSE) 1.0 provides a compatible implementation of secure sockets for the Java 2 platform. For optimal performance of the HTTPS connector, it is recommended that you use the Java 2 SDK, Standard Edition, v1.2.2, and the JSSE 1.01.

The web site for the JSSE is `http://java.sun.com/products/jsse`. This site provides links for downloading the software and the documentation. For further information and details regarding the use of the secure sockets, please refer to the JSSE documentation.

The HTTPS connector exposes the same interfaces as all other connectors and has exactly the same behavior. The development of management application which relies on the HTTPS connector is no different from that of any other Java Dynamic Management manager. See “Connector Clients” on page 113 for details about programming with the `RemoteMBeanServer` API.

Where the HTTPS connector differs is that it relies on the security mechanisms built into the Java language and extended by JSSE. In order to use these libraries and communicate securely, you must configure your application environment to meet all security requirement. The cost of security is establishing all of the structures that guarantee the trust between two communicating parties.

This section covers the steps that are required to establish a secure connection between your agent and manager applications. These instructions do not guarantee total security, they just explain the programmatic steps needed to ensure data security between two distant Java applications.

These steps assume that each of your manager and agent applications runs on a separate machine, and that each machine has its own installation of the Java SDK (not a shared network installation).

1. Install All Software

You should install both the Java 2 SDK, Standard Edition, v1.2.2, and the JSSE 1.01 products on all hosts that will use the HTTPS connector client or connector server components.

In this procedure, the directories where you have installed these products are named *JAVAhome* and *JSSEhome*, respectively. These names are used on all hosts where the products are installed, even though their value is specific to each host.

2. Extend Your Java Runtime Libraries

For each of your SDK/JSSE installations, copy the three jar files (`jsse.jar`, `jcert.jar`, and `jnet.jar`) of the JSSE reference implementation into the extensions directory of your Java runtime environment.

For example, on the Solaris platform you would type:

```
$ cp JSSEhome/lib/jsse.jar JAVAhome/jre/lib/ext/
$ cp JSSEhome/lib/jcert.jar JAVAhome/jre/lib/ext/
$ cp JSSEhome/lib/jnet.jar JAVAhome/jre/lib/ext/
```

Or you could add these jar files to your environment's classpath, as follows (for the Korn shell):

```
$ export CLASSPATH=${CLASSPATH}:JSSEhome/lib/jsse.jar:\
JSSEhome/lib/jcert.jar:JSSEhome/lib/jnet.jar
```

3. Designate your Security Provider

The JSSE follows the same "provider" architecture found in the Java Cryptography Architecture (JCA) which is provided in the Java 2 platform as the Java Cryptography Extension (JCE). In order to use JSSE you must install this provider either statically or dynamically. Again, you must do this for all of your SDK/JSSE installations.

To install the provider statically you must edit the security properties file (*JAVAhome/*`lib/security/java.security`). Edit this file as follows, the boldface text is the part you must add:

```
security.provider.1=sun.security.provider.Sun

security.provider.2=com.sun.net.ssl.internal.ssl.Provider
```

The first line of this file depends upon your SDK platform and should not be changed.

To install the provider dynamically in your Java application, you should call the `addProvider` method of the `java.security.Security` class. The line in your source code would look like this:

```
Security.addProvider( new com.sun.net.ssl.internal.ssl.Provider() );
```

4. Generate Public and Private Keys

This step must be repeated on all agent and manager host machines.

Generate a key pair (a public key and associated private key). Wrap the public key into an X.509 v1 self-signed certificate, which is stored as a single-element certificate chain. This certificate chain and the private key are stored in a new keystore entry identified by *alias*.

In the following command, the –`dname` parameters designates the X.500 Distinguished Name for the host where you are generating the certificates. The *commonName* field must be the machine's hostname.

```
$ keytool -genkey -alias alias -keyalg RSA -keysize 1024 -sigalg MD5withRSA
         -dname "CN=commonName, OU=orgUnit, O=org, L=location, S=state, C=country"
         -keypass passPhrase -storetype jks -keystore yourHome/.keystore
```

```
          -storepass passPhrase
```

5. Export a Local Certificate

This step must be repeated on all agent and manager host machines.

Read the certificate that is associated with your *alias* from the keystore, and store it in a *hostCertFile*:

```
$ keytool -export -alias alias -file hostCertFile -storetype jks
          -keystore yourHome/.keystore -storepass passPhrase -rfc
```

When you are done with this step you will have a certificate for each of your host machines.

6. Import all Remote Certificates

This step must be repeated on both the agent and manager host machines, for all pairs of agent-managers in your management architecture.

In this step, agent and manager pairs must exchange their certificates. The manager should import the agent's *hostCertFile* and the agent should import the manager's *hostCertFile*. If a manager has two agents, it will import two certificates, and each agent will import a copy of the manager's certificate.

Import the certificate into the file containing the trusted Certificate Authorities (CA) certificates. This will add our self-signed certificate as a trusted CA certificate to the `cacerts` file so that the server and the client will be able to authenticate each other.

```
$ keytool -import -alias alias -file hostCertFile -noprompt -trustcacerts
          -storetype jks -keystore JAVAhome/jre/lib/security/cacerts
          -storepass changeit
```

This command modifies the *JAVAhome*`/jre/lib/security/cacerts` which will affect all applications running on that installation. If you do not want to modify this file, you could create a file named `jssecacerts` and use it instead. The default location of this file is either *JAVAhome*/`lib/security/jssecacerts` or if that does not exist, then *JAVAhome*/`lib/security/cacerts`.

7. Run Your Java Dynamic Management Agent

Launch your agent applications with the following properties:

```
$ java -Djavax.net.ssl.keyStore=yourHome/.keystore
       -Djavax.net.ssl.keyStoreType=jks
       -Djavax.net.ssl.keyStorePassword=passPhrase
       AgentClass
```

If you are using the notification push mechanism, add the following property definition to the above command line:

```
-Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
```

8. Run Your Management Application

Launch your management applications with the following properties:

```
$ java -Djavax.net.ssl.keyStore=yourHome/.keystore
       -Djavax.net.ssl.keyStoreType=jks
       -Djavax.net.ssl.keyStorePassword=passPhrase
       -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
       ManagerClass
```

PART IV Agent Services

The minimal and base agents presented earlier in this tutorial are manageable but are created empty. In this lesson we will examine agent services, MBeans that interact with your resources to provide management intelligence at the agent level. These services allow your agents to perform local operations on their resources that used to be performed by the remote manager. This frees you from having to implement them in the management application, and it reduces the amount of communication throughout your management solution.

In the Java Dynamic Management architecture, agents perform their own monitoring, avoiding constant manager polling. Agents also handle their own logical relations between objects, providing advanced operations that no longer require a relational database in the manager.

Java Dynamic Management agents are also much smarter, using discovery to be aware of their peers. They also connect to other agents to mirror remote resources, thus providing a single point of entry into a whole hierarchy of agents. Finally, agents are freed from their initial environment settings through dynamic downloading, allowing managers to effectively push new classes to an agent. As new resources and new services are developed, they can be loaded into deployed agents, boosting their capabilities without impacting their availability.

The benefits of agent services is multiplied by their dynamic nature. Agent services can come and go as they are required, either as determined by the needs of agent’s smart resources or on demand from the management application. The agent capability of self-management through the services is completely scalable: small devices might only allow monitors with some logic, whereas a server might embed the algorithms and networking capabilities to oversee its terminals autonomously.

This lesson contains the following topics:

- “The M-Let Class Loader” downloads new classes to the agent from a given URL (Universal Resource Locator). An m-let is a management applet: the HTML-style tag at the target URL that contains information about the classes to download. A manager can store new MBean classes anywhere, update its m-let file to reference them, and instruct an agent to load the classes. The new classes are created as MBeans, registered in the MBean server and ready to be managed.

- "The Relation Service" creates associations between MBeans, allowing for consistency checking against defined roles and relation types. New types can be defined dynamically to create new relations between existing objects. All relations are managed through the service so that it can expose query operations for finding related MBeans. Relations themselves may also be implemented as MBeans, allowing them to expose attributes and operations that act upon the MBeans in the relation.
- "Cascading Agents" allow a manager to access a hierarchy of agents through a single point-of-access. The subagents can be spread across the network, but their resources can be controlled by the manager through a connection to a single master agent. MBeans of a subagent are mirrored in the master agent and respond as expected to all management operations, including their removal. No special proxy classes are needed for the mirror MBean, meaning that every MBean object can be mirrored anywhere without requiring any class loading.
- "The Discovery Service" lets applications discover Java Dynamic Management agents that want to be found. Using active discovery, the client broadcasts a discovery request over the network. Agents which have a discovery responder registered in their MBean server will automatically send a response. The client can then keep a list of reachable agents by using passive discovery to detect when responders are activated and deactivated. Along with the agent's address, the discovery response contains version information from the agent's delegate MBean and the list of available protocol adaptors and connectors.

CHAPTER **12**

The M-Let Class Loader

The “Dynamic” in Java Dynamic Management Kit not only stands for dynamic loading, it also stands for dynamic *down*loading. The agent service that provides this functionality is the m-let class loader. M-let stands for *management applet*, an HTML-style tag that tells the class loader how to retrieve the desired class. Using this information, the m-let loader can retrieve an MBean class from a remote location given as a URL (uniform resource locator) and create it in the agent.

The m-let resides in a separate text file which acts as a loading manifest. The contents of the m-let file let you specify any number of classes to load, possibly a different source for each, arguments for the class constructor, and the object name for the instantiated MBean. Since this mechanism is sometimes too heavy, the m-let loader can also be used to load classes directly and create MBeans in the agent.

The m-let loader is a service implemented as an MBean, so it can be called either directly by the agent or remotely by a management application. It can also be managed remotely, which allows a manager to effectively “push” MBeans to an agent: the manager instantiates the m-let loader in an agent and instructs it to load classes from a predetermined location.

Class loading is significantly different between Java 2 and JDK 1.1.*x*, and this leads to different implementations of the m-let loader. We cover these in separate sections. The example code in the `MLetAgent` and `MLetClient` directories of the main *examplesDir* corresponds to the Java version which you specified during installation (see “Directories and Classpath” in the preface).

Contents:

The M-Let Loader (JDK 1.1)

In an agent application, we might need to load MBeans from remote hosts during the initialization. Or we might have local threads which need to load MBeans in the agent. In this example we demonstrate how to create the m-let loader service and use it to dynamically load new MBean classes.

In an installation of the Java Dynamic Management Kit for the JDK 1.1.*x*, the m-let loader is the `MLetSrv` class in the `javax.management.loading` package. It is an MBean which needs to be registered in the MBean server before we can use it to load classes.

CODE EXAMPLE 12–1 Instantiating the `MLetSrv` Class

```
MBeanServer server = MBeanServerFactory.createMBeanServer();

// Get the domain name from the MBeanServer.
String domain = server.getDefaultDomain();

// Create a new MLetSrv MBean and add it to the MBeanServer.
String mletClass = "javax.management.loading.MLetSrv";
ObjectName mletName = new ObjectName(domain + ":name=" + mletClass);
server.createMBean(mletClass, mletName);
```

If your agent needs to download classes that include native libraries, you will need to handle security issues: see "Security Manager (JDK 1.1)" on page 198. Otherwise, there is no special initialization that needs to be done before loading Java classes.

Loading MBeans from a URL

In order to download an MBean, we must first have its corresponding m-let definition in an HTML file. In our example, we define the following file with three `MLET` tags:

CODE EXAMPLE 12–2 The M-Let File

```
<HTML>
<MLET
  CODE=Square.class
  ARCHIVE=Square.jar
  NAME=MLetExample:name=Square,id=1
>
<ARG TYPE=java.lang.Integer VALUE=10>
```

(Continuation)

```
</MLET>
<MLET
  CODE=EquilateralTriangle.class
  ARCHIVE=EquilateralTriangle.jar
  NAME=MLetExample:name=EquilateralTriangle,id=1
>
<ARG TYPE=java.lang.Integer VALUE=8>
</MLET>
<MLET
  CODE=EquilateralTriangle.class
  ARCHIVE=EquilateralTriangle.jar
  NAME=MLetExample:name=EquilateralTriangle,id=2
>
<ARG TYPE=java.lang.Integer VALUE=15>
</MLET>
</HTML>
```

This file tells the m-let loader to create three MBeans with the given object names, using the given classes in the jar files. The jar files must be located in the same directory as this file, regardless of whether the directory is on a local or remote host. The `MLET` tag may also specify a `CODEBASE`, which is an alternate location for the jar file. The `MLET` tag is fully defined in the JMX specification.

Now we are ready to call the `performLoadURL` method of our m-let loader. In parsing the result vector, we use our knowledge of the class names that we wrote in the m-let file.

CODE EXAMPLE 12–3 Calling the `performLoadURL` Method

```
ObjectName squareMLetClassLoader = null;
ObjectName triangleMLetClassLoader = null;

// The url string is read from the command line
Object mletParams[] = {url};
String mletSignature[] = {"java.lang.String"};
Vector mbeanList = (Vector) server.invoke(
    mletName, "performLoadURL", mletParams, mletSignature);

for (Enumeration enum = mbeanList.elements();
     enum.hasMoreElements(); ) {
    Object element = enum.nextElement();
    if (element instanceof Vector) {
        // Success, we retrieve the new object name
        Vector v = (Vector) element;
        ObjectInstance objectInstance = (ObjectInstance) v.elementAt(0);
        ObjectName classLoaderObjectName = (ObjectName) v.elementAt(1);
        if (objectInstance.getClassName().equals("Square")) {
            // Retrieve MBean that loaded the class Square
```

```
                squareMLetClassLoader = classLoaderObjectName;

            } else if (objectInstance.getClassName().equals(
                           "EquilateralTriangle")) {
                // Retrieve MBean that loaded the class EquilateralTriangle
                triangleMLetClassLoader = classLoaderObjectName;
            }
            echo("\tOBJECT NAME = " + objectInstance.getObjectName());
        } else {
            // Failure, find out why
            echo("\tEXCEPTION = " + ((Throwable)element).getMessage());
        }
    }
```

The result of the call to performLoadURL is a `Vector` object containing as many elements as there are `MLET` tags in the file designated by the URL. Each element is either a vector containing the object instance of the new MBean and the object name of its class loader, or a `Throwable` object containing the exception or error that prevented the MBean from being loaded.

In the result, we obtain the object name of the MBeans that were created from the downloaded classes. The management architecture specified by JMX is designed so that objects are manipulated through the MBean server, not by direct reference. Therefore, downloaded classes are directly registered in the MBean server by the m-let loader, and the caller never receives a direct reference to the new object.

The object names of the class loaders are references to the internal class loader objects used by the m-let service to actually fetch the classes. We save them because they can be used if we ever need to instantiate these classes again. We will see how in the next section.

Shortcut for Loading MBeans

Loading MBeans from a URL requires some preparation and additional files. In some cases, we don't have the ability to create files ahead of time or modify them when we need different classes. In these cases, we would just like to load a class from a jar file and create its MBean.

The `MLetSrv` is not a class loader, we only ask it to load a class from a URL and it instantiates its private class loader for doing this. Even though the internal class loader object used by the m-let loader is a public type, it should not be instantiated to act as a class loader. The m-let loader stores internal information about its private class loaders, and it won't be able to handle one outside of its control.

Instead, use the class loader name that is returned when an MBean is successfully loaded. You can specify this class loader name when creating a class through the MBean server. You will be able to create new MBeans from the same class or from other classes in the associated archive (jar file).

This implies that you must first call the `performLoadURL` with a known URL and a known m-let file. The m-let loader will create one class loader for each code-base specified in the file, and one for the code-base of the file itself. For example, the class loader name returned with the “Square” MBean name is the one used to load its class from the `Square.jar` file in the same directory as the HTML file. We can create other instances of that MBean now just through the MBean server, without needing to call the m-let loader.

The following code sample uses the object name references that were declared and assigned in Code Example 12–3.

CODE EXAMPLE 12–4 Loading Classes Directly

```
// Create a new Square MBean from its class in the Square.jar file
String squareClass = "Square";
ObjectName squareName = new ObjectName(
    "MLetExample:name=" + squareClass + ",id=2");
Object squareParams[] = {new Integer(12)};
String squareSignature[] = {"java.lang.Integer"};
server.createMBean(squareClass, squareName, squareMLetClassLoader,
                   squareParams, squareSignature);

// Create a new EquilateralTriangle MBean from its class in the
// EquilateralTriangle.jar file
String triangleClass = "EquilateralTriangle";
ObjectName triangleName = new ObjectName(
    "MLetExample:name=" + triangleClass + ",id=3");
Object triangleParams[] = {new Integer(20)};
String triangleSignature[] = {"java.lang.Integer"};
server.createMBean(triangleClass, triangleName, triangleMLetClassLoader,
                   triangleParams, triangleSignature);
```

Loading classes directly in this manner implies that the code of the agent or of the MBean must be programmed with the knowledge of the class named in the m-let file, and if needed, the knowledge of other classes in the jar file from which the class was finally loaded.

Running the M-Let Agent Example

To run the m-let agent example for the JDK 1.1.*x*, you must have installed the Java Dynamic Management Kit for 1.1, and set your classpath accordingly. This example is located in the *examplesDir*/`MLetAgent/` directory, see “Directories and Classpath” in the preface for details.

In our example, we have two MBeans representing geometrical shapes. Before running the example, we compile them and create a jar file for each. We also compile the agent application at the same time.

```
$ cd examplesDir/MLetAgent/
$ javac -classpath classpath *.java

$ jar cf Square.jar Square.class SquareMBean.class
$ rm Square.class SquareMBean.class

$ jar cf EquilateralTriangle.jar EquilateralTriangle.class \
EquilateralTriangleMBean.class
$ rm EquilateralTriangle.class EquilateralTriangleMBean.class
```

Since the MBean classes are only found in the jar files now, they cannot be found in our usual classpath, even if it includes the current directory (`.`). However, these jar files are given as the archive in the `MLET` tags of the HTML file, so the m-let loader should find them.

The agent requires you to specify the URL of the m-let file on command line. We have left this file in the examples directory, but you could place it and the jar files on a remote machine. With the Korn shell on the Solaris platform, you would type the following command:

```
$ java -classpath classpath Agent file:${PWD}/GeometricShapes.html
```

In the output of the agent, you can see it create the m-let service MBean, and then load the HTML file which specifies the three MBeans to be loaded. Once these have been loaded, we can see the two MBeans that were loaded directly through the class loader shortcut.

This agent uses the tracing mechanism, and you can select to receive the messages from the m-let loader by specifying the `-DINFO_MLET` property on the command line. The tracing mechanism is covered in the *Java Dynamic Management Kit 4.2 Tools Reference* guide and in the Javadoc API of the `Trace` class.

The agent then launches an HTML adaptor so that we can easily view the new MBeans. In them we can see that the values contained in the `ARG` tags of the m-let file were used to initialize the MBeans. Point your web browser to the following URL and click on the MBeans in the `MLetExample` domain:`http://localhost:8082/`. When you are done, type `<Control-C>` in the window where you launched the agent.

M-Let Loading from a Manager (JDK1.1)

Like the other agent services of the Java Dynamic Management Kit, the m-let loader is an MBean and fully manageable from a remote manager. A manager application might want an agent to load a new MBean to represent a new resource or provide a new management service. In this example, we show a manager which interacts with the m-let loader of an agent to load exactly the same MBeans as in the previous example.

The agent that we will manage only contains an RMI connector and an HTML adaptor when it is launched. We will use the RMI connector to access the agent and perform all of our management operations. You can then view the new MBeans through the HTML adaptor.

The manager is just a simple application which creates its connector client, does its management operations and exits. Here is the code of its `main` method and the constructor that it calls.

CODE EXAMPLE 12–5 The `main` Method of the M-Let Manager

```
public Client() {

    // Enable the trace mechanism
    [...]

    // Connect a new RMI connector client to the agent
    connectorClient = new RmiConnectorClient();

    // Use the default address (localhost)
    RmiConnectorAddress address = new RmiConnectorAddress();
    try {
        connectorClient.connect(address);
    } catch (Exception e) {
        echo("Could not connect to the agent!");
        e.printStackTrace();
        System.exit(1);
    }
}

public static void main(String[] args) {

    // Parse command line arguments.
    [...]

    // Call the constructor to establish the connection
    Client client = new Client();

    // Run the MLet example (see below)
    client.runMLetExample();
```

(continued)

(Continuation)

```
    // Disconnect connector client from the connector server.
    client.connectorClient.disconnect();

    System.exit(0);
}
```

Asking the Agent to Load Classes

Now that the manager is connected to the client, we can “push” classes to it. We do this by first creating an m-let loader service, then having that loader create MBeans from the classes designated by our HTML file. The following code is taken from manager’s `runMLetExample` method. The code is identical to the code of the agent example, except that we now go through the `RemoteMBeanServer` interface of the connector client instead of directly through the MBean server.

CODE EXAMPLE 12–6 Calling the `performLoadURL` Method Remotely

```
// Get the domain name from the Agent
String domain = connectorClient.getDefaultDomain();

// Create a new MLetSrv MBean and add it to the Agent
String mletClass = "javax.management.loading.MLetSrv";
ObjectName mletName = new ObjectName(domain + ":name=" + mletClass);
connectorClient.createMBean(mletClass, mletName);
[...]

// Create and register new Square and EquilateralTriangle MBeans
// by means of an HTML document containing MLET tags
// The url string is read from the command line
ObjectName squareMLetClassLoader = null;
ObjectName triangleMLetClassLoader = null;

Object mletParams[] = {url};
String mletSignature[] = {"java.lang.String"};
Vector mbeanList = (Vector) connectorClient.invoke(
    mletName, "performLoadURL", mletParams, mletSignature);

for (Enumeration enum = mbeanList.elements(); enum.hasMoreElements(); ) {
    Object element = enum.nextElement();
    if (element instanceof Vector) {
        // Success, we retrieve the new object name
        Vector v = (Vector) element;
        ObjectInstance objectInstance = (ObjectInstance) v.elementAt(0);
        ObjectName classLoaderObjectName = (ObjectName) v.elementAt(1);
```

(continued)

(Continuation)

```
        if (objectInstance.getClassName().equals("Square")) {
            // Retrieve the MBean that loaded the Square

            squareMLetClassLoader = classLoaderObjectName;
        } else if (objectInstance.getClassName().equals(
                       "EquilateralTriangle")) {
            // Retrieve the MBean that loaded the EquilateralTriangle
            triangleMLetClassLoader = classLoaderObjectName;
        }
        echo("\tOBJECT NAME = " + objectInstance.getObjectName());
    } else {
        // Failure, find out why
        echo("\tEXCEPTION = " + ((Throwable)element).getMessage());
    }
}
```

As in the agent application, we may need a shortcut for instantiating other MBeans without specifying an m-let file. Again, we can use an existing class loader from a previously loaded class to download the same classes again. We use the `createMBean` method of the connector client which lets us specify a class loader name. The following code is the rest of the manager's `runMLetExample` method, and it is also nearly identical to the agent's code.

CODE EXAMPLE 12–7 Asking the Agent to Load Classes Directly

```
// Create a new Square MBean from its class in the Square.jar file.
String squareClass = "Square";
ObjectName squareName = new ObjectName(
    "MLetExample:name=" + squareClass + ",id=2");
Object squareParams[] = {new Integer(12)};
String squareSignature[] = {"java.lang.Integer"};
connectorClient.createMBean(squareClass, squareName,
    squareMLetClassLoader, squareParams, squareSignature);

// Create a new EquilateralTriangle MBean from its class in the
// EquilateralTriangle.jar file.
String triangleClass = "EquilateralTriangle";
ObjectName triangleName = new ObjectName(
    "MLetExample:name=" + triangleClass + ",id=3");
Object triangleParams[] = {new Integer(20)};
String triangleSignature[] = {"java.lang.Integer"};
connectorClient.createMBean(triangleClass, triangleName,
    triangleMLetClassLoader, triangleParams, triangleSignature);
```

(continued)

Simulating a “push” of the MBeans in this way is plausible, since the management application can specify a URL where it controls the contents of the HTML file and knows which classes are available.

Running the M-Let Manager Example

The MBeans in the agent and manager (client) examples are identical, and we will set up the example in exactly the same manner.

```
$ cd examplesDir/MLetClient/
$ javac -classpath classpath *.java

$ jar cf Square.jar Square.class SquareMBean.class
$ rm Square.class SquareMBean.class

$ jar cf EquilateralTriangle.jar EquilateralTriangle.class \
EquilateralTriangleMBean.class
$ rm EquilateralTriangle.class EquilateralTriangleMBean.class
```

The manager is written to be run on the same host as the agent application. If you wish to run it on a different host, you will need to modify the code for the `Client` class constructor where the agent address is specified (see Code Example 12–5). You could place the jar files and the m-let file on a remote machine and specify its new URL as the parameter to the manager application; we run the example with this file in the current directory.

Before launching the manager, you must launch the agent. Here we give commands for launching the applications from the same terminal window running the Korn shell. On the Windows NT platform, you will have to launch each application in a separate window.

```
$ java -classpath classpath Agent &
$ java -classpath classpath Client file:${PWD}/GeometricShapes.html
```

In the output of the manager, you can see it create the m-let service MBean, and then ask it to load the HTML file. Finally, we can see the two MBeans that were loaded directly through the class loader shortcut. If you connect to the agent in a web browser at the following URL: `http://localhost:8082/` and reload its agent view every time the manager pauses, you can see the MBeans as they are created.

The agent terminates after it disconnects its connector client. When you are done viewing the agent, type the following commands to stop the agent application:

```
$ fg
java [...] Agent <Control-C>
^C$
```

The M-Let Loader (Java 2)

In the version of the Java Dynamic Management Kit for Java 2, the m-let loader is itself a class loader object. It extends the `URLClassLoader` class of the `java.net` package to simplify the downloading service it provides.

The m-let loader service is an instance of the `MLet` class in the `javax.management.loading` package. It is also an MBean that can be accessed remotely. It provides m-let file loading and the shortcut method seen in the version for the JDK 1.1. In addition, it inherits the behavior which lets it be used directly as a class loader, without requiring an m-let file.

We will start by demonstrating the usage of the m-let service as it would be used in an agent or in an MBean. In our example, the agent application creates an MBean server and then the m-let loader.

CODE EXAMPLE 12–8 Instantiating the `MLet` Class

```
// Parse debug properties and command line arguments.
[...]

// Instantiate the MBean server
MBeanServer server = MBeanServerFactory.createMBeanServer();
String domain = server.getDefaultDomain();

// Create a new MLet MBean and add it to the MBeanServer.
String mletClass = "javax.management.loading.MLet";
ObjectName mletName = new ObjectName(domain + ":name=" + mletClass);
```

```
server.createMBean(mletClass, mletName);
```

There is no special initialization that needs to be done before loading classes through an m-let file.

Loading MBeans from a URL

In this example we will only load `EquilateralTriangle` MBeans through the m-let file. We use the same m-let file which is shown in Code Example 12–2, but without the tags for the `Square` MBeans.

The code for downloading the MBeans specified in the m-let file is also similar. In the Java 2 version of the m-let loader, only the name of the method to call and the format of its return value is different. In this code we call the `getMBeansFromURL` method and analyze the result:

CODE EXAMPLE 12–9 Calling the `getMBeansFromURL` Method

```
// the url_2 string is read from the command line
echo("\tURL = " + url_2);
Object mletParams_2[] = {url_2};
String mletSignature_2[] = {"java.lang.String"};
Set mbeanSet = (Set) server.invoke(mletName, "getMBeansFromURL",
     mletParams_2, mletSignature_2);

for (Iterator i = mbeanSet.iterator(); i.hasNext(); ) {
    Object element = i.next();
    if (element instanceof ObjectInstance) {
        // Success, we display the new MBean's name
        echo("\tOBJECT NAME = " + ((ObjectInstance)element).getObjectName());
    } else {
        // Failure, we display why
        echo("\tEXCEPTION = " + ((Throwable)element).getMessage());
    }
}
```

The structure of the returned set is much simpler than in the case of the JDK 1.1 loader. In the JDK 1.1 version, the m-let loader handles separate class loader objects, one for each code-base it has accessed. In the Java 2 version, the m-let loader is the class loader, and it handles just a list of code-bases that it has accessed directly. You can view this list by calling the getURLs method of the m-let loader MBean.

This behavior means that the getMBeansFromURL method does not need to return the object names of class loaders it has used. Instead it just returns either the object instance of the downloaded and registered MBean or a Throwable object in case or an error or an exception. These are returned in a Set object containing as many elements as there are MLET tags in the target m-let file.

Shortcut for Loading MBeans

This behavior also simplifies any repeated loading of the classes after they have been loaded from an m-let file. Because the m-let loader has already used the code-base of the MBean, it is available to be used again. All you need to do is specify the object name of the m-let loader as the class loader when creating the MBean.

You can also load other MBeans in the same code-base, once the code-base has been accessed by a call to the getMBeansFromURL method. In our example we will just download another MBean of the EquilateralTriangle class.

CODE EXAMPLE 12–10 Reloading Classes in the M-Let Class Loader

```
// Create another EquilateralTriangle MBean from its class
// in the EquilateralTriangle.jar file.
String triangleClass = "EquilateralTriangle";
ObjectName triangleName = new ObjectName(
    "MLetExample:name=" + triangleClass + ",id=3");
Object triangleParams[] = {new Integer(20)};
String triangleSignature[] = {"java.lang.Integer"};

server.createMBean(triangleClass, triangleName, mletName,
                   triangleParams, triangleSignature);
```

Again, loading classes from known code-bases or reloading a class directly from its jar file implies that the agent or MBean programmer has some knowledge of the code-bases and jar file contents at runtime.

Loading MBeans Directly

Since the m-let loader object is a class loader, you can use it to load classes directly, without needing to define an m-let file. This is the main advantage of the Java 2 version of the m-let loader service.

Before you can load an MBean directly, you need to add the URL of its code-base to the m-let loader's internal list. Then we just use the m-let loader's object name as the class loader name when creating the MBean. Here is the code to do this in the agent example:

CODE EXAMPLE 12–11 Using the M-Let MBean as a Class Loader

```
// Add a new URL to the MLet class loader
// The url_1 string is read from the command line
Object mletParams_1[] = {url_1};
String mletSignature_1[] = {"java.lang.String"};
server.invoke(mletName, "addURL", mletParams_1, mletSignature_1);

// Create a Square MBean from its class in the Square.jar file.
String squareClass = "Square";
ObjectName squareName = new ObjectName(
    "MLetExample:name=" + squareClass);
Object squareParams[] = {new Integer(10)};
String squareSignature[] = {"java.lang.Integer"};
server.createMBean(squareClass, squareName, mletName,
                   squareParams, squareSignature);
```

You only need to add the URL to the m-let loader the first time you want to download a class. Once it is added, we can load it as many times as necessary by calling `createMBean` directly.

Since this loading mechanism doesn't use the `MLET` tag, the programmer must insure that either the downloaded class provides its own object name or, as in the example above, the agent provides one.

The fact that the m-let loader is also a class loader into which you can load multiple URLs brings up the issue of name spaces. If there exists two classes with the same name within the code-bases defined by the set of all URLs, the m-let loader will load one of them non-deterministically. In order to specify one of them precisely, you shouldn't add the URL of the second code-base to the m-let loader. Instead, you will have to create a second m-let loader MBean to which you can add the URL for the second version of the class. In this case, you will have one m-let MBean that can load one version of the class and another m-let MBean that can load the other.

Running the M-Let Agent Example

To run the m-let agent example for Java 2, you must have installed the Java Dynamic Management Kit for 1.2, and set your classpath accordingly. This example is located in the *examplesDir*/`MLetAgent/` directory, see "Directories and Classpath" in the preface for details.

In our example, we have two MBeans representing geometrical shapes. Before running the example, we compile them and create a jar file for each. We also compile the agent application at the same time.

```
$ cd examplesDir/MLetAgent/
$ javac -classpath classpath *.java

$ jar cf Square.jar Square.class SquareMBean.class
$ rm Square.class SquareMBean.class

$ jar cf EquilateralTriangle.jar EquilateralTriangle.class \
EquilateralTriangleMBean.class
$ rm EquilateralTriangle.class EquilateralTriangleMBean.class
```

The agent command line requires you to specify first the URL of a jar file for directly loading the `Square` class, then the URL of the m-let file. We have left these files in the examples directory, but you could place them on a remote machine. With the Korn shell on the Solaris platform, you would type the following command:

```
$ java -classpath classpath Agent \
file:${PWD}/Square.jar file:${PWD}/GeometricShapes.html
```

In the output of the agent, you can see it create the m-let loader MBean, and then download classes to create MBeans. It starts with the direct loading of the `Square` class, and then loads from the HTML file which specifies two `EquilateralTriangle` MBeans to be loaded. Once these have been loaded, we can see the third one that is loaded through the class loader shortcut.

This agent uses the tracing mechanism, and you can select to receive the messages from the m-let loader by specifying the `-DINFO_MLET` property on the command line. The tracing mechanism is covered in the *Java Dynamic Management Kit 4.2 Tools Reference* guide and in the Javadoc API of the `Trace` class.

The agent then launches an HTML adaptor so that we can easily view the new MBeans. In them we can see that the values contained in the `ARG` tags of the m-let file were used to initialize the MBeans. Point your web browser to the following URL and click on the MBeans in the `MLetExample` domain: `http://localhost:8082/` . When you are done, type `<Control-C>` in the window where you launched the agent.

M-Let Loading from a Manager (Java 2)

Since the `MLet` class is an MBean, the m-let loader service is fully manageable from a remote manager. This lets a manager create an m-let loader in an agent, add URLs to its list of code-bases, and create MBeans whose classes must be downloaded first.

Using the Java 2 m-let class loader, we can again implement a "push" mechanism originating from a management application. In fact, it is even easier due to the direct class loading that the `MLet` MBean allows.

In the example, our manager will create an m-let loader in an agent and have it load new MBean classes. Launched with only the an RMI connector and an HTML adaptor, we can see at the end that the agent contains all of the new MBeans loaded from jar files, along with the m-let loader MBean. The initialization of manager is very simple:

CODE EXAMPLE 12–12 The `main` Method of the Manager Application

```
public Client() {

    // Enable the trace mechanism
    [...]

    // Connect a new RMI connector client to the agent
    connectorClient = new RmiConnectorClient();

    // Use the default address (localhost)
    RmiConnectorAddress address = new RmiConnectorAddress();
    try {
        connectorClient.connect(address);
    } catch (Exception e) {
        echo("Could not connect to the agent!");
        e.printStackTrace();
        System.exit(1);
    }
}

public static void main(String[] args) {

    // Parse command line arguments.
    [...]

    // Call the constructor to establish the connection
    Client client = new Client();

    // Run the MLet example (see below)
    client.runMLetExample();

    // Disconnect connector client from the connector server.
```

(continued)

(Continuation)

```
    client.connectorClient.disconnect();

    System.exit(0);
}
```

Asking the Agent to Load Classes

Now that the manager is connected to the client, we can ask it to load classes. First we have it create an m-let loader MBean that we can use to download classes. Then we demonstrate the various ways of loading classes:

- Through an m-let file, which has the advantage of loading many MBeans at once
- Directly, from a code-base that was used in an m-let file
- Directly, after specifying a URL for the code-base

The following code is taken from manager's `runMLetExample` method. The code is identical to the code of the agent example, except that we now go through the `RemoteMBeanServer` interface of the connector client instead of directly through the MBean server.

CODE EXAMPLE 12–13 Calling the `getMBeansFromURL` Method Remotely

```
// Get the domain name from the MBeanServer.
String domain = connectorClient.getDefaultDomain();

// Create a new MLet MBean and add it to the MBeanServer
String mletClass = "javax.management.loading.MLet";
ObjectName mletName = new ObjectName(domain + ":name=" + mletClass);
connectorClient.createMBean(mletClass, mletName);

[...]

// Create new EquilateralTriangle MBeans through MLET tags
// The url_2 string is read from the command line
Object mletParams_2[] = {url_2};
String mletSignature_2[] = {"java.lang.String"};
Set mbeanSet = (Set) connectorClient.invoke(
    mletName, "getMBeansFromURL", mletParams_2, mletSignature_2);

for (Iterator i = mbeanSet.iterator(); i.hasNext(); ) {
    Object element = i.next();
    if (element instanceof ObjectInstance) {
```

(continued)

(Continuation)

```
        // Success
        echo("OBJECT NAME = " + ((ObjectInstance)element).getObjectName());
    } else {
        // Failure
        echo("EXCEPTION = " + ((Throwable)element).getMessage());
    }
}
```

Now that the class loader has used the code-base of the jar file, we can create more of the MBeans from the same jar file. We invoke the createMBean method of the server with the object name of the class loader.

CODE EXAMPLE 12–14 Reloading MBeans from an Existing Code-Base

```
// Create another EquilateralTriangle MBean from the same jar file
// used in the MLET file
String triangleClass = "EquilateralTriangle";
ObjectName triangleName = new ObjectName(
    "MLetExample:name=" + triangleClass + ",id=3");
Object triangleParams[] = {new Integer(20)};
String triangleSignature[] = {"java.lang.Integer"};
connectorClient.createMBean(triangleClass, triangleName, mletName,
                            triangleParams, triangleSignature);
```

Finally, if we have a different code-base not associated with an m-let file, we can give its URL directly to the loader. This allows us to ask the agent to create almost any MBean, imitating a class "push" mechanism.

CODE EXAMPLE 12–15 Implementing a "Push" Operation

```
// Add a new URL to the MLet MBean to look for classes
// The url_1 string is read from the command line
Object mletParams_1[] = {url_1};
String mletSignature_1[] = {"java.lang.String"};
connectorClient.invoke(mletName, "addURL",
                       mletParams_1, mletSignature_1);

// Create a new Square MBean from its class in the Square.jar file
String squareClass = "Square";
```

(continued)

```
ObjectName squareName = new ObjectName(
    "MLetExample:name=" + squareClass);
Object squareParams[] = {new Integer(10)};
String squareSignature[] = {"java.lang.Integer"};
connectorClient.createMBean(squareClass, squareName, mletName,
                            squareParams, squareSignature);
```

In this way, the manager can make code available on the network, and it can direct its agents to load the classes to create new MBeans ready for management. This mechanism can be used to distribute new resources, provide new services,or update applications, all under the control of the manager.

Running the M-Let Manager Example

The MBeans in the agent and manager (client) examples are identical, and we will set up the example in exactly the same manner.

```
$ cd examplesDir/MLetClient/
$ javac -classpath classpath *.java

$ jar cf Square.jar Square.class SquareMBean.class
$ rm Square.class SquareMBean.class

$ jar cf EquilateralTriangle.jar EquilateralTriangle.class \
EquilateralTriangleMBean.class
$ rm EquilateralTriangle.class EquilateralTriangleMBean.class
```

The manager is written to be run on the same host as the agent application. If you wish to run it on a different host, you will need to modify the code for the `Client` class constructor where the agent address is specified (see Code Example 12–12). You could place the jar files and the m-let file on a remote machine and specify its new URL as the parameter to the manager application; we run the example with this file in the current directory.

Before launching the manager, you must launch the agent. Here we give commands for launching the applications from the same terminal window running the Korn shell. On the Windows NT platform, you will have to launch each application in a separate window.

```
$ java -classpath classpath Agent &
$ java -classpath classpath Client \
file:${PWD}/Square.jar file:${PWD}/GeometricShapes.html
```

In the output of the manager, you can see it create the m-let service MBean, and then load all of the MBeans from different sources. If you connect to the agent in a web browser at the following URL: `http://localhost:8082/` and reload its agent view every time the manager pauses, you can see the MBeans as they are created.

The agent terminates after it disconnects it connector client. When you are done viewing the agent, type the following commands to stop the agent application:

```
$ fg
java [...] Agent <Control-C>
^C$
```

Secure Class Loading

Because class loading exposes an agent to external classes, the Java Dynamic Management Kit offers security within the m-let service. Security mechanisms differs between the JDK 1.1 and Java 2 Java runtime environments.

Security Manager (JDK 1.1)

The default behavior of the Java virtual machine forbids you from downloading native libraries across the network, since the code they contain cannot be controlled by the usual Java security mechanism. If the classes you wish to download are bundled with native libraries, you will need to instantiate a custom security manager.

The Java Dynamic Management Kit provides a simple implementation of a security manager, called `com.sun.jdmk.AgentSecurityManager`, which accepts all incoming libraries. This security manager extends the `java.lang.SecurityManager` interface, and you must install it as the default security manager for your agent application.

Before your agent instantiates the `MLetSrv` class it must set the security manager in the Java virtual machine. This is done with the following call:

```
import com.sun.jdmk.AgentSecurityManager;
```

```
System.setSecurityManager(new AgentSecurityManager());
```

Code Signing (JDK 1.1)

Code signing is a security measure which you can use to identify the originator of a downloaded class. The m-let service will enforce code signatures if it is instantiated in secure mode. One of the constructors of the `MLetSrv` class takes a boolean parameter which specifies the security mode. For obvious security reasons, the security mode cannot be modified once the m-let service is instantiated.

When the m-let service is running in secure mode, it will only load classes and native libraries which are signed by a trusted party. A trusted party is identified by a key: this key was used to sign the code and a copy of the key is given to all parties that wish to download the signed class. Therefore, you must identify trusted keys in your agent before attempting to download their signed classes.

Note - Downloading native libraries always requires a custom security manager, regardless of whether they are trusted or not. See the description of the "Security Manager (JDK 1.1)" on page 198.

In the JDK 1.1 environment, `.jar` files are signed using the `javakey` utility. You also use the `javakey` utility on your agent's host to identify trusted keys. The command line parameters of this tool allow you to define your security policy based on trusted identities and keys. Please refer to the JDK documentation of the `javakey` utility for details.

When the secure mode of the m-let service is enabled, unsigned classes and libraries will never be loaded.

When the secure mode is not enabled, all classes and native libraries may be downloaded, regardless of whether they are signed and not trusted, or not signed.

Code Signing (Java 2)

The code signing mechanism and trust policies were modified for the Java 2 platform.

In the `MLet` class for the Java 2 platform, security is no longer determined when you instantiate the m-let service. Rather, security is enabled or disabled for your entire agent application, including any class loaders used by the m-let service. Class loaders on the Java 2 platform also rely on code signing, but the mechanism is different.

To enable security on the Java 2 platform, launch your agent applications with the `java.lang.SecurityManager` property on the command line. Then, when the m-let service loads a class through one of its class loaders, the class loader will check the origin and signature of the class against the list of trusted origins and signatures.

The tools involved in signing a class file are the `jar`, `keytool`, and `jarsigner` utilities. On the host where the agent application would like to download a class, you define a set of permissions for signatures and URL origins. Then, you need to use the `policytool` utility to generate a `java.policy` file containing the trusted signatures. Please refer to the JDK documentation for the description of these utilities.

When the agent application is launched with a security manager, it will check this policy file to insure that the origin and signature of a downloaded class match a trusted origin and a trusted signature. If they do not match, the code is not trusted and cannot be loaded.

When the agent application is launched without the security manager, all classes and native libraries may be downloaded and instantiated, regardless of their origin and signature, or lack thereof.

CHAPTER **13**

The Relation Service

The relation service defines and maintain logical relations between MBeans in an agent. It acts as central repository of relation types and relation instances, and it ensures the consistency of all relations it contains. Local and remote applications that access the agent can define new types of relations and then declare a new instance of a relation between MBeans.

You may only create relations that respect a known relation type, and the service allows you to create new relation types dynamically. Then you can access the relations and retrieve the MBeans in specific roles. The relation service listens for MBeans being deregistered and removes relation instances that no longer fulfill their relation type. It sends notifications of its own to signal when relation events occur.

The Java Dynamic Management Kit also exposes the interfaces classes for defining your own relation objects. By creating relation instances as objects, you can implement them as MBeans which can expose operations on the relation they represent.

Like the other services, the relation service is instrumented as an MBean, allowing remote access and management.

The code samples in this topic are taken from the files in the `Relation` directory located in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

- “Defining Relations” on page 202 demonstrates how to create relations through the relation service.
- “Operations of the Relation Service” on page 207 covers the operations for manipulating relations and their consequences.
- “Objects Representing Relations” on page 210 shows how to define relations outside of the relation service, yet under its control.
- “Running the Relation Service Example” on page 215 demonstrates the functionality of this service.

Defining Relations

A relation is composed of named *roles*, each of which defines the cardinality and class of MBeans that will be put into association with the other roles. A set of one or more roles defines a *relation type*. The relation type is a template for all relation instances that wish to associate MBeans representing its roles. We use the term *relation* to mean a specific instance of a relation that associates existing MBeans according to the roles in its defining relation type.

For example, we can say that `Books` and `Owner` are roles. `Books` represents any number of owned books of a given MBean class, and `Owner` is a single book owner of another MBean class. We might define a relation type containing these two roles and call it `Personal Library`: it represents the concept of book ownership.

The following diagram represents this sample relation, as compared to the UML modeling of its corresponding association.

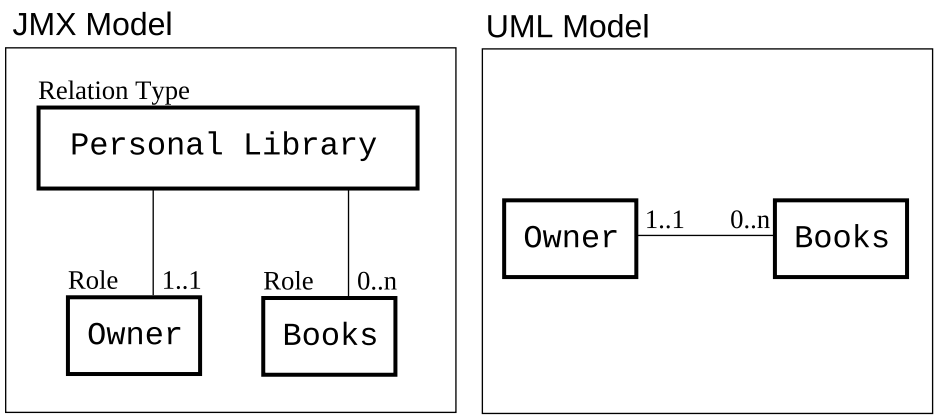


Figure 13–1 Comparison of the Relation Models

There is a slight difference between the two models. The UML association implies that each one of the `Books` can only have one owner. Our relation type only models a set of roles, guaranteeing that a relation instance has one `Owner` MBean and any number of MBeans in the role of `Books`.

Note - The relation service does not do inter-relation consistency checks, they are the responsibility of the designer if they are needed. In our example, the designer would need to ensure that the same book MBean does not participate in two different `Personal Library` relations, while allowing it for an owner MBean.

In the rest of this topic, we will see how to handle the roles, relation type and relation instances through the relation service. We will use the names and relations presented in the programming example. First of all, we instantiate the relation service as we would any other MBean and register it in our agent's MBean server.

CODE EXAMPLE 13–1

```
// Create the RelationService MBean
String relServClassName =
    "javax.management.relation.RelationService";
ObjectName relServObjName = new ObjectName(
    "DefaultDomain:type=javax.management.relation.RelationService1");

// We use a constructor which takes a boolean parameter
// to set the ImmediatePurge mode for relation consistency
Object[] params = new Object[1];
params[0] = new Boolean(true);
String[] signature = new String[1];
signature[0] = "boolean";

server.createMBean(theRelServClassName, theRelServObjName,
                   params, signature);
```

The relation service exposes an attribute called `Active` that is `false` until its MBean is registered with the MBean server. All of the operations that handle relation or role MBeans, either directly or indirectly, will throw an exception when the service is not active.

Defining Role Information

Before we can create relation instances, we need a relation type, and before we can define a relation type, we need to represent the information about its roles. This information includes:

- A name string (required)
- The name of the MBean class that fulfills this role (required)
- Read-write permissions (the default is both readable and writeable)
- The multiplicity, expressed as a single range (the default is 1..1)
- A description string (the default is a null string)

The name can be any string that is manipulated by the Java `String` class. It will be used to retrieve the corresponding MBeans when accessing a relation instance. The multiplicity is limited to the range between the minimum and maximum number of MBeans required for this role to fulfilled. The read-write permissions apply to all of the MBeans in the role, since the value of a role is read or written as a complete list.

The role information is represented by the `RoleInfo` class. In our example, we define two different roles:

CODE EXAMPLE 13–2 Instantiating `RoleInfo` Objects

```
// Define two roles in an array
// - container: SimpleStandard class/read-write access/multiplicity: 1..1
// - contained: SimpleStandard class/read-write access/multiplicity: 0..n

RoleInfo[] roleInfoArray = new RoleInfo[2];
String role1Name = "container";
roleInfoArray[0] =
    new RoleInfo( role1Name, "SimpleStandard",
                  true, true,
                  1, 1,
                  null);

String role2Name = "contained";
roleInfoArray[1] =
    new RoleInfo( role2Name, "SimpleStandard",
                  true, true,
                  0, -1,
                  null);
```

We build an array of `RoleInfo` objects that is intended to define a relation type, so it needs to define a valid set of roles. All role names must be unique within the array, and none of the array's elements may be null. Also, the minimum and maximum cardinalities must define a range of at least one integer.

Defining Relation Types

We define a relation type in the relation service by associating a name for the relation type with a non-empty array of `RoleInfo` objects. These are the parameters to the service's `createRelationType` method that we call through the MBean server.

CODE EXAMPLE 13–3 Defining a Relation Type

```
try {
    String relTypeName = "myRelationType";

    Object[] params = new Object[2];
    params[0] = relTypeName;
    params[1] = roleInfoArray;
    String[] signature = new String[2];
    signature[0] = "java.lang.String";
    // get the string representing the "RoleInfo[]" object
    signature[1] = (roleInfoArray.getClass()).getName();

    server.invoke( relServObjName, "createRelationType",
```

(continued)

(Continuation)

```
                    params, signature);

} catch (Exception e) {
    echo("\tCould not create the relation type " + relTypeName);
    printException(e);
}
```

The relation type name given by the caller must be unique among all relation type names already created in the relation service. Relations will refer to this name to define their type, and the service will verify that the roles of the relation match the role information in this type.

The relation service provides methods for managing the list of relation types it stores. The `removeRelationType` removes the type’s definition from the relation service and also removes all relation instances of this type. This mechanism is further covered in “Maintaining Consistency” on page 208.

Other methods give the list of all currently defined relation types or the role information associated with a given type. Role information is always obtained through the name of the relation type where it is defined. Here we show the subroutine that our example uses to print out all role and type information.

CODE EXAMPLE 13–4 Retrieving Relation Types and Role Information

```
try {
    echo("\n-> Retrieve all relation types");
    Object[] params1 = new Object[0];
    String[] signature1 = new String[0];
    ArrayList relTypeNameList = (ArrayList)
    (server.invoke( relServObjName, "getAllRelationTypeNames",
                    params1, signature1));

    for (Iterator relTypeNameIter = relTypeNameList.iterator();
         relTypeNameIter.hasNext(); ) {
        String currRelTypeName = (String)(relTypeNameIter.next());
        echo("\n-> Print role info for relation type " +
             currRelTypeName);
        Object[] params2 = new Object[1];
        params2[0] = currRelTypeName;
        String[] signature2 = new String[1];
        signature2[0] = "java.lang.String";
        ArrayList roleInfoList = (ArrayList)
            (server.invoke( relServObjName,"getRoleInfos",
                            params2, signature2));
        printList(roleInfoList);
```

(continued)

(Continuation)

```
    }
} catch (Exception e) {
    echo("\tCould not browse the relation types");
    printException(e);
}
```

Creating Relations

Now that we have defined a relation type, we can use it as a template for creating a relation. A relation is a set of roles which fulfills all of the role information of the relation type. The `Role` object contains a role name and value which is the list of MBeans that fulfills the role. The `RoleList` object contains the set of roles used when setting a relation or getting its role values.

In order to create a relation we must provide a set of roles whose values will initialize the relation correctly. In our example we use an existing `SimpleStandard` MBean in each role that we have defined. Their object names are added to the value list for each role. Then the each `Role` object is added to the role list.

CODE EXAMPLE 13–5 Initializing Role Objects and Creating a Relation

```
[...] // define object names and create SimpleStandard MBeans

// Instantiate the roles using the object names of the MBeans
ArrayList role1Value = new ArrayList();
role1Value.add(mbeanObjectName1);
Role role1 = new Role(role1Name, role1Value);

ArrayList role2Value = new ArrayList();
role2Value.add(mbeanObjectName2);
Role role2 = new Role(role2Name, role2Value);

RoleList roleList1 = new RoleList();
roleList1.add(role1);
roleList1.add(role2);

String relId1 = relTypeName + "_instance";

try {
    Object[] params = new Object[3];
    params[0] = relId1;
    params[1] = relTypeName;
    params[2] = roleList1;
    String[] signature = new String[3];
    signature[0] = "java.lang.String";
```

(continued)

(Continuation)

```
    signature[1] = "java.lang.String";
    signature[2] = "javax.management.relation.RoleList";
    server.invoke(theRelServObjName, "createRelation",
                  params, signature);
} catch(Exception e) {
    echo("\tCould not create the relation " + RelId1);
    printException(e);
}
```

The `createRelation` method will raise an exception if the provided roles do not fulfill the specified relation type. You may omit `Role` objects for roles that allow a cardinality of 0; their values will be initialized with an empty list of object names. The relation service will check all provided object names to ensure they are registered with the MBean server. Also, the relation identifier name is a string which must be unique among all relations in the service.

The corresponding `removeRelation` method is also exposed for management operations. It is also called internally to keep all relations coherent, as described in "Maintaining Consistency" on page 208. In both cases, removing a relation means that you can no longer access it through the relation service, and the `isRelation` operation will return false when given its relation identifier. Also, its participating MBeans will no longer be associated through the roles in which they belonged. The MBeans continue to exist unaltered otherwise and may continue to participate in other relations.

Operations of the Relation Service

In addition to the creation and removal methods for relation types and instances, the relation service provides operations for finding related MBeans, determining the role of a given MBean, and accessing the MBeans of a given role or relation.

Query Operations

Once relations have been defined, the relation service allows you to do searches based on the association of objects that the relations represent. The following operation perform queries on the relations:

- `findAssociatedMBeans` - Returns a list of all object names referenced in any relation where a given object name appears; each of these object names is mapped to the list of relation identifiers where the two MBeans are related, since the pair can be related through different relation instances

  Two optional parameters let you specify a relation type and role name. When either or both of these are specified, the only relations to be considered are those of the given type and/or where the given object name appears in the named role.
- `findReferencingRelations` - Takes an object name and returns the list of relation identifiers where it is referenced; each identifier is mapped to the list of roles in which the corresponding MBean appears in that relation; again, you may specify the relation type and/or role name in which the given MBean must appear
- `getReferencedMBeans` - Returns a list of all MBeans currently in a given relation; their object names are mapped to the role names where they are referenced, since the same MBean may appear in more than one role of the same relation
- `findRelationsOfType` - Returns the list of identifiers of all relations that were created or added with a given relation type
- `getRelationTypeName` - This method is the inverse of the previous, returning the relation type name of the relation with a given identifier

Accessing Roles

Once you have a relation identifier, you will probably want to access its role values. The relation service provides getters and setters for roles, as well as bulk operations to get or set several or all roles in a relation. Remember that the value of a role is a list of MBeans, and a role is identified by a name string.

Input parameters to setter operations are the same `Role` and `RoleList` classes that are used to create relations. Bulk getters and setters return a `RoleResult` object which contains separate lists for successful and failed individual role operations.

Inside a role result, the list of roles and values that were successfully accessed are given in a `RoleList` instance. The information about roles that couldn't be read or written is returned in a `RoleUnresolvedList`. This list contains `RoleUnresolved` objects that name the role which couldn't be accessed and an error code explaining the reason, such as an invalid role name or the wrong cardinality for a role. The error codes are defined as static final fields in the `RoleStatus` class.

Maintaining Consistency

All relation service operations that set the role of a relation always verify that the role value fulfills its definition in the relation type. An incorrect MBean type, the wrong cardinality of the role, or an unknown role name will prevent that role from being modified, guaranteeing that the relation always fulfills it relation type.

As we shall see in "Objects Representing Relations" on page 210, relations and relation types can also be objects that are external to the relation service. In this case, they are responsible for maintaining their own role-consistency. The relation service MBean exposes methods to assist these object is verifying that their roles conform to their defined relation type. In all cases of an external implementation of a relation, the object designer is responsible for ensuring its coherence.

Removing a relation type can cause existing relations to no longer have a defining type. The policy of the `removeRelationType` operation is to assume that the caller is aware of existing relations of the given type. Instead of forbidding the operation, this method removes the relations that were defined by the given type. It is the designer's responsibility to first call the `findRelationsOfType` operation to determine if any existing relations will be affected.

MBeans participating in a relation may be removed from the MBean server by some other management operation, thereby modifying the cardinality of a role where they were referenced. In order to maintain the consistency in this case, the relation service must remove all relations in which a role no longer has the cardinality defined in it relation type. The process of determining invalid relations and removing them is called a purge.

The relation service listens for deregistration notifications of the MBean server delegate, and will need to purge its MBeans whenever one is received. It must determine if the removed MBean was involved in any relations, and if so, whether its removal violates the cardinality of each role where it appears. The relation service exposes the boolean `PurgeFlag` attribute which the programmer must set to determines whether purges are done automatically or not.

When the purge flag is `true`, the relation service will purge its data immediately after every deregistration notification. However, the purge operation can be resource intensive for large sets of relation data. In that case the managing application may set the purge flag to `false` and only call the `purgeRelations` operation to purge the data when needed.

For example, if there are many MBean deregistrations and few relation operations, it may make sense to only purge the relation service manually before each operation. Or the automatic purge flag may be temporarily set to false while executing time-critical operations that need to remove MBeans, but that won't access the relation service.

There are two possible consequences of an unpurged relation service. Roles in a relation may reference object names which no longer have an associated MBean. Or worse, the object name may have been reassigned to another MBean, leading to a totally incoherent state. The designer of the management solution is responsible for setting the purge policy so that operations will always access consistent relation values.

Relation Service Notifications

The relation service is a notification broadcaster that notifies listeners of events affecting relations. It sends `RelationNotification` objects in the following cases:

- A new relation is **created**: the notification contains its identifier and its relation type
- An existing relation is **updated**: in addition to the relation identifier and type, the notification contains the list of names, the list of new values, and the list of old values for all roles that have been modified
- A relation is **removed**: the notification contains its identifier and its relation type

There are three equivalent notification types for events affecting relations defined as external MBeans (see "Objects Representing Relations" on page 210). In addition to the role and relation information, these notifications contain the object name of this MBean.

Objects Representing Relations

When creating relation types and instances, their representations are handled internally and only accessed through the interface of the relation service. However, the service also allows you to create external objects which represent relations, and then add them under the service's control to access them through its operations.

One advantage of representing relation types and instances as classes that they can perform their initialization in the class constructor. Applications can then instantiate the classes and add them to the relation service directly, without needing to code for their creation. The relation type classes can be downloaded to agent applications for use throughout the management architecture.

Another advantage is that relations are represented as MBeans and must be registered in the MBean server. This means that management applications can get role information directly from the relation MBean instead of going through the relation service.

The main advantage of an external relation class is that it can be extended to add properties and methods to a relation. They can be accessible through attributes and operations in the relation MBean so that they are also exposed for management. These extensions can represent more complex relations and allow more flexible management architectures.

With the power of manipulating relations comes the responsibility of maintaining the relation model. A relation MBean can expose an operation for adding an object name to a role, but its implementation must first ensure that the new role value will conform to the relation's type. Then, it must also instruct the relation service to send a role update notification. The relation service MBean exposes methods for maintaining consistency and performing required operations. It is the programmer's responsibility to call the relation service when necessary in order to maintain the consistency of its relation data.

The `RelationTypeSupport` Class

A class must implement the `RelationType` interface in order to be considered a representation of a relation type. The methods of this interface are used to access the the role information that makes up a relation type. Since relation types are immutable within the relation service, there are no methods for modifying the role information.

The `RelationTypeSupport` class is the implementation of this interface that is used internally by the relation service to represent a relation type. By extending this class, you can quickly write new relation type classes with all the required functionality. The class has a method for adding roles to the information that is exposed; this method can be called by the class constructor to initialize all roles. Our simple example does just this, and there is little other functionality that can be added to a relation type object.

CODE EXAMPLE 13–6 Extending the `RelationTypeSupport` Class

```
import javax.management.relation.*;

public class SimpleRelationType extends RelationTypeSupport {

    // Constructor
    public SimpleRelationType(String theRelTypeName) {

        super(theRelTypeName);

        // Defines the information for two roles
        // - primary: SimpleStandard class/read-write/cardinality=2
        // - secondary: SimpleStandard class/read-only/cardinality=2
        try {
            RoleInfo primaryRoleInfo =
                new RoleInfo("primary", "SimpleStandard",
                             true, true,
                             2, 2,
                             "Primary description");
            addRoleInfo(primaryRoleInfo);

            RoleInfo secondaryRoleInfo =
                new RoleInfo("secondary", "SimpleStandard",
                             true, false,
                             2, 2,
                             "Secondary description");
            addRoleInfo(secondaryRoleInfo);
        } catch (Exception exc) {
            throw new RuntimeException(exc.getMessage());
```

(continued)

```
        }
    }
}
```

We now use our class to instantiate an object representing a relation type. We then call the relation service's `addRelationType` operation to make this type available in the relation service. Thereafter, it is manipulated through the service's operations and there is no way to distinguish it from other relation types that have been defined.

CODE EXAMPLE 13–7 Adding an Externally Defined Relation Type

```
String usrRelTypeName = "SimpleRelationType";
SimpleRelationType usrRelType =
    new SimpleRelationType("SimpleRelationType");
try {
    Object[] params = new Object[1];
    params[0] = usrRelType;
    String[] signature = new String[1];
    signature[0] = "javax.management.relation.RelationType";
    server.invoke( relServObjName, "addRelationType",
                   params, signature);
} catch(Exception e) {
    echo("\tCannot add user relation type");
    printException(e);
}
```

The role information defined by a relation type should never change once the type has been added to the relation service. This is why the `RelationTypeSupport` class is not an MBean: it would make no sense to manage it remotely. All of the information about its roles is available remotely through the relation service MBean.

The `RelationSupport` Class

The external class representation of a relation instance must implement the `Relation` interface so that it can be handled by the relation service. The `RelationSupport` class is the implementation provided which is also used internally by the service.

The methods of the relation interface expose all of the information needed to operate on a relation instance: the getters and setters for roles and the defining relation type.

Because the relation support class must represent any possible relation type, it has a constructor which takes a relation type and a role list, and it uses a generic mechanism internally to represent any roles.

You could implement a simpler relation class that implements a specific relation type, in which case it would know and initialize its own roles. The class could also interact with the relation service to create its specific relation type before adding itself as a relation.

However, the simplest way to define a relation object is to extend the `RelationSupport` class and provide any additional functionality you require. In doing so, you can rely on the relation support class' own methods for getting and setting roles, thereby taking advantage of their built-in consistency mechanisms.

CODE EXAMPLE 13–8 Extending the `RelationSupport` Class

```
import javax.management.ObjectName;
import javax.management.relation.*;

public class SimpleRelation extends RelationSupport
    implements SimpleRelationMBean {

    // Constructor
    public SimpleRelation(String theRelId,
                          ObjectName theRelServiceName,
                          String theRelTypeName,
                          RoleList theRoleList)
        throws InvalidRoleValueException,
               IllegalArgumentException {

        super(theRelId, theRelServiceName, theRelTypeName, theRoleList);
    }

    [...] // Implementation of the SimpleRelationMBean interface
}
```

Our MBean's `SimpleRelationMBean` interface itself extends the `RelationSupportMBean` in order to expose its operations for management. In order to represent a relation, the class must be an MBean registered in the MBean server. This allows the relation service to rely on deregistration notifications in order to know if the object name is no longer valid.

When instantiating our `SimpleRelation` MBean, we use the relation type defined in Code Example 13–3. We also reuse the role list from Code Example 13–5 which contains a single MBean in each of two roles. Before adding the relation to the relation service, we must create it as an MBean in the MBean server. We then call the `addRelation` operation of the relation service with our relation's object name.

CODE EXAMPLE 13–9 Creating an External Relation MBean

```
// Using relTypeName="myRelationType"
// and roleList1={container,contained}

String relMBeanClassName = "SimpleRelation";
String relId2 = relTypeName + "_instance";
ObjectName relMBeanObjName = new ObjectName(
    "DefaultDomain:type=SimpleRelation2";

try {
    Object[] params1 = new Object[4];
    params1[0] = relId2;
    params1[1] = relServObjName;
    params1[2] = relTypeName;
    params1[3] = roleList1;
    String[] signature1 = new String[4];
    signature1[0] = "java.lang.String";
    signature1[1] = "javax.management.ObjectName";
    signature1[2] = "java.lang.String";
    signature1[3] = "javax.management.relation.RoleList";
    server.createMBean(relMBeanClassName, relMBeanObjName,
                       params1, signature1);
} catch(Exception e) {
    echo("\t Could not create relation MBean for relation " + relId2);
    printException(e);
}

// Add our new MBean as a relation to the relation service
try {
    Object[] params2 = new Object[1];
    params2[0] = relMBeanObjName;
    String[] signature2 = new String[1];
    signature2[0] = "javax.management.ObjectName";
    server.invoke(relServObjName, "addRelation",
                  params2, signature2);
} catch(Exception e) {
    echo("\t Could not add relation MBean " + relMBeanObjName);
    printException(e);
}
```

After a relation instance is added to the relation service, it can be accessed normally like other relations. Management operations can either operate on the MBean directly or access it through its identifier in the relation service. Two methods of the service are specific to external relation MBeans:

- `isRelationMBean` - Takes a relation identifier and returns the object name of the relation MBean, if it is defined
- `isRelation` - Takes an object name and returns its relation identifier, if it is a relation MBean that has been added to the relation service

In our example, our MBean does not expose any functionality that interacts with the relation service. It relies fully on the support class and only adds the implementation of its own simple MBean interface. In a more realistic example, the MBean would expose attributes or operations related to its role values.

For example, `Personal Library` could be a unary relation type containing the `Books` role. We could then design an `Owner` MBean to be a relation of this type. In addition to exposing attributes about the owner, the MBean would give the number of books in the library, return an alphabetized list of book titles, and provide an operation for selling a book. All of these would need to access the role list, and the sale operation would need to modify the role to remove a book MBean.

All of these operations would need to keep the consistency of the relation. To do this, the relation service exposes several methods that relation MBeans must call:

- `checkRoleReading` - Verifies the read access for the role of a given relation type, before it is read
- `checkRoleWriting` - Verifies the write access, multiplicity and MBean class of a role value before it can be written
- `updateRoleMap` - Provides the old and new vales of a role so that the relation service can update its internal lists of referenced MBeans
- `SendRoleUpdateNotification` - Instructs the relation service to send a notification containing the given old and new values of a role; since the relation service must be the broadcaster for all relation notifications, relation MBeans must call this method after modifying a role value

Running the Relation Service Example

The *examplesDir*/`Relation` directory contains all of the files for the agent application and the associated MBeans.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/Relation/
$ javac -classpath classpath *.java
```

To run the relation service example, launch its application with the following command. Be sure that the classes for the `SimpleRelationType` and `SimpleRelation` MBean can be found in its *classpath*.

```
$ java -classpath classpath RelationAgent
```

When launched, the application first creates the `RelationService` MBean and then begins its long demonstration of operations on roles, types and relations. The output of each step is separated by a horizontal line to detect its starting point in the scrolling display.

Press `<Enter>` to step through the example when the application pauses, or `<Control-C>` at any time to exit.

CHAPTER **14**

Cascading Agents

The cascading service allows you to access the MBeans of a subagent directly through the MBean server of the master agent. The service is implemented in the `CascadingAgent` MBean which connects to a remote subagent and makes all of the subagent's MBeans visible in the master agent. An agent can have several subagents, and subagents may themselves cascade other agents, forming a hierarchy of cascading agents.

By connecting to the root of an agent hierarchy, managers can have a single access point to many resources and services. All MBeans in the hierarchy are manageable through the top-most master agent, and a manager doesn't need to worry about their physical location. Like the other services, the cascading agent is implemented as an MBean that can be managed dynamically. This allows the manager to control the structure of the agent hierarchy, adding and removing subagents as necessary.

In particular, the cascading service MBean can work with any protocol connector, including any custom implementation. Those supplied by the Java Dynamic Management Kit give you the choice of using RMI, HTTP, or HTTPS. The cascading service also lets you specify a filter for selecting precisely the MBeans of the subagent that will be mirrored in the master agent. This mechanism lets you limit the number of MBeans which are mirrored in the top-most agent of a large cascading hierarchy.

The code samples in this topic are taken from the files in the `Cascading` directory located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Contents:

The `CascadingAgent` MBean

You should create one `CascadingAgent` MBean for every subagent you wish to manage through the master agent. Each connects to an agent and mirrors all of that agent's registered MBeans in the master agent's MBean server. No other classes are required or need to be generated in order to represent the MBeans.

The agent whose MBean server contains an active cascading service is called a *master agent* in relation to the other agent which is mirrored. The agent to which the cascading service is connected is called the *subagent* in relation to its master agent. We say that it creates *mirror MBeans* to represent the subagent's MBeans. See "The Mirror MBeans in the Master Agent" on page 220 for a description of these objects.

A master agent can have any number of subagents, each controlled individually by a different `CascadingAgent` MBean. A subagent may itself contain cascading agents and mirror MBeans, all of which are mirrored again in the master agent. This effectively allows cascading hierarchies of arbitrary depth and width.

Two master agents may also connect to the same subagent. This is similar to the situation where two managers connect to the same agent and may access the same MBean. If the implementation of a management solution permits such a condition, it is the designer's responsibility to handle any synchronization issues in the MBean.

The connection between two agents resembles the connection between a manager and an agent. The cascading service MBean relies on a connector client, and the subagent must have the corresponding connector server. The subagent's connector server must already be instantiated, registered with its MBean server, and ready to receive connections.

In our example application, we use the RMI connector client which we will connect to the RMI connector server of the subagent on port 1099 of the localhost. In fact, this is the same as the default values when instantiating a cascading agent MBean, but we also wish to specify a pattern for selecting the MBeans to mirror. By default, all MBeans of the subagent are mirrored in the master agent; we provide an object name pattern to only select those in the subagent's `CascadedDomain`.

CODE EXAMPLE 14–1 Connecting to a Subagent

```
ObjectName mbeanObjectName = null;
String domain = server.getDefaultDomain();
mbeanObjectName = new ObjectName(domain + ":type=CascadingAgent");
[...]

RmiConnectorAddress address = new RmiConnectorAddress(
    java.net.InetAddress.getLocalHost().getHostName(),
    1099,
    "name=RmiConnectorServer");
```

(continued)

```
CascadingAgent remAgent = new CascadingAgent(
    address,
    "com.sun.jdmk.comm.RmiConnectorClient",
    new ObjectName("CascadedDomain:*"),
    null);
ObjectInstance remAgentInstance =
    server.registerMBean(remAgent, mbeanObjectName);

[...] // Output omitted
// Now we explicitly start the cascading agent
// as it is not started automatically
//
echo("\nStarting the cascading agent...");
[...]
server.invoke(mbeanObjectName, "start", null, null);
sleep(1000);
echo("\tIs ACTIVE = " + server.getAttribute(mbeanObjectName, "Active"));
echo("done");
```

Before the subagent's MBeans are mirrored, the `CascadingAgent` MBean must be registered in the master agent's MBean server, and its "mirroring" must be started. When you invoke the `start` operation of the cascading service MBean, it will connect it to its designated subagent and create one mirror MBean to represent each MBean in the subagent. When its `Active` attribute becomes true, the cascading mechanism is ready to use.

The `CascadingAgent` MBean exposes two writeable attributes:

- `Address` is a `ConnectorAddress` object whose subclass defines the protocol used for the connection and whose contents gives the address or machine name of the subagent, along with a port number
- `ClientConnectorClassName` is a string which gives the class name of the connector client to be used in the connection; it must be compatible with the protocol defined by the class of the `Address` attribute

Neither of these attributes may be modified when the cascading service is active. You must first call the MBean's `stop` operation: this will remove all of the mirror MBeans for the given subagent and disconnect from the subagent's connector server. You may then modify the address or the class of the connector client. The new values will be used when you start the mirroring again: this lets you change subagents or even change protocols.

When the cascading service is stopped or its MBean is removed from the master agent, all of its mirror MBeans are deregistered. The MBean server delegate in the master agent will send a deregistration notification for each mirror MBean as it is removed.

The Mirror MBeans in the Master Agent

Once the cascading service is active, you interact directly with the mirror MBeans representing the subagent’s MBeans. You may access and manage a mirror MBean *as if* you are connected to the subagent and accessing or managing the original MBean. The mirror MBeans are actual MBean objects registered in the master agent’s MBean server with the same object name.

All management operations that could be performed on the original MBean can be performed identically on its mirror MBean. You may modify attributes, invoke operations and add or remove listeners, all with exactly the same result as if the manager were connected to the subagent when performing the action.

The behavior of a mirror MBean is to transmit the action to the subagent’s MBean and return with an answer or result. The actual computation is performed by the original MBean running in its own agent.

In our example, we know that there is a timer MBean that was created in the subagent. Once the cascading service is active for our subagent, we operate the timer through it local mirror MBean. We can never have the direct reference to a mirror MBean, so we always invoke operations through the master agent’s MBean server.

CODE EXAMPLE 14–2 Managing Mirrored MBeans

```
// Here we know the object name of the MBean we want to access, the
// object name of its mirrored MBean in the master agent will be identical.
ObjectName timerName = new ObjectName("CascadedDomain:type=timer");

echo("\n>>>  Ask the Timer MBean to send a notification every 5 seconds ");
java.util.Date currentDate = new java.util.Date();
Object params[] = {
    "Timer",
    "Message",
    new Integer(5),
    new java.util.Date (currentDate.getTime() + new Long (2).longValue()),
    new Long(1000) };

String signatures[]={ "java.lang.String",
                      "java.lang.String",
                      "java.lang.Object",
                      "java.util.Date",
                      "long"};

server.invoke(timerName, "addNotification", params, signatures);
```

(continued)

```
server.invoke(timerName, "start", null, null);

echo("\n>>>  Add ourselves as a listener to the Timer MBean");
server.addNotificationListener(timerName, this, null, null);

echo("\nPress <Enter> to remove the listener from the Timer MBean ");
waitForEnterPressed();
server.removeNotificationListener(timerName, this);
```

For the managing application, the mirror MBean in the master agent *is* the MBean. Unregistering a mirror MBean in the master agent will unregister the mirrored MBean in the subagent. If you want to control the number of mirror objects without removing the originals, you must use filters and/or queries of the subagent's MBeans in the constructor of the cascading service MBean.

The mirror and its mechanism make the cascading totally transparent: a manager has no direct way of knowing whether an object is a mirror or not. Neither does it have any direct information about the topology of the cascading hierarchy rooted at the agent which it accesses. If this information is necessary, the MBeans should expose some kind of identification through their attributes, operations, or object names.

The Class of a Mirror MBean

Mirror MBeans are implemented as dynamic MBeans; they are instances of the `CascadeGenericProxy` class. The cascading service gives them the MBeanInfo object that they will expose and establishes their connection with the original MBean. The MBean information contains the class name of the original MBean, not their own class name. Exposing this borrowed class name guarantees that the cascading service is completely transparent.

The symmetry of the Java Dynamic Management architecture means that this cascading mechanism is scalable to any number of levels. The mirror object of a mirror object is again an instance of the `CascadeGenericProxy` class, and it borrows the same object name and class name. Any operation on the top-most mirror will be propagated to its subagent, where the intermediate mirror will send it its own subagent, and so forth. The cost of cascading is the cost of accessing the subagent: the depth of your cascading hierarchy should be adapted to your management solution.

Because the cascading service MBean instantiates and controls all mirror MBeans, the `CascadeGenericProxy` class should never be instantiated through a management operation, nor by the code of the agent application. We have described it here only to provide an example application of dynamic MBeans.

Cascading Issues

In this section, we explain some of the design issues that are determined by the implementation of the cascading service.

Dynamic Mirroring

Any changes in the subagent's MBeans are automatically applied to the set of mirror MBeans, to ensure that both master agent and subagent remain consistent.

When an MBean is unregistered from the subagent, the cascading service MBean removes its mirror MBean from the master agent. When a new MBean is registered in the subagent, the cascading service instantiates its mirror MBean, sets up its connection to the new MBean, and registers the mirror with the master agent's MBean server.

Both of these mechanisms scale to cascading hierarchies: adding or removing an MBean in the master agent will trigger a notification that any cascading service connected to the master agent will receive. This will start a chain reaction up to the top of the hierarchy. Removing an MBean from the middle of a hierarchy also triggers a similar reaction down to the original MBean which is finally removed.

Dynamic unregistration only applies to subagent MBeans that are actually mirrored. Dynamic registration is also subject to filtering, as described in the next section.

MBean Filtering

When the cascading service MBean is instantiated, you may pass it an object name pattern and a query expression. These will be applied to the list of MBeans in the subagent to determine those that will be mirrored in the master agent. The filtering will be in effect for the life of the cascading service connected to this MBean.

Filtering the mirrored MBeans reduces the number of MBeans in the master agent. It also provides a way of identifying mirror MBeans in the master agent, as in our example where cascading is limited to MBeans in the `CascadedDomain`.

Both the object name pattern and query expression will be used to filter any new MBean which is registered in the subagent. If the new MBean meets the filter criteria, it will become visible and mirrored in the master agent. Since the query expression applies to attribute values of the MBean, you must be careful to initialize the new MBean before registering it so that its values are significant when the filter is applied by the cascading service.

Naming in Cascading Agents

Mirror MBeans are registered in the master agent with the same object name as the mirrored MBean in the sub-agent. If the registration fails in the master agent's MBean server, no error is raised and no action is taken: the corresponding MBean will simply not be mirrored in the master agent.

The most likely cause for registration to fail is that the object name already exists in the master agent. An MBean cannot be registered if its chosen object name already exists in the MBean server.

If your management solution has potential naming conflicts, you will need a design that is guaranteed to assign unique object names throughout the cascade hierarchy. You can set the default domain name in your subagents or use the `MBeanServerId` attribute of the delegate MBean to give MBeans a unique object name.

Running the Cascading Example

The *examplesDir*`/Cascading` directory contains all of the files for the two agent applications, along with a simple MBean.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/Cascading/
$ javac -classpath classpath *.java
```

To run the cascading example, launch the subagent in another terminal window with the following command. Be sure that the classes for the `SimpleStandard` MBean can be found in its *classpath*.

```
$ java -classpath classpath SubAgent
```

Wait for the agent to be completely initialized, then launch the master agent with the following command:

```
$ java -classpath classpath MasterAgent
```

When launched, the master agent application first creates the `CascadingAgent` MBean and then sets up its connection to the subagent. The master agent then performs operations on the mirrored MBeans of the subagent. Press `<Enter>` to step through the example when the application pauses.

You may also interact with the example through the HTML adaptor of the master agent and subagent. If you are still receiving timer notification on the master agent, press `<Enter>` once more to remove the listener, but leave both agent applications running.

▼ Interacting with a Cascade Hierarchy

1. **Open two browser windows side by side and load the following URLs:**

The Subagent	The Master Agent
`http://localhost:8082/`	`http://localhost:8084/`

In the subagent, you should see the timer MBean in the `CascadedDomain` and a `SimpleStandard` MBean in the `DefaultDomain`.

The master agent is recognizable by the cascading service MBean in the `DefaultDomain`. Otherwise it has an identical timer MBean registered in the `CascadedDomain`: this is the mirror for the timer in the subagent. The `SimpleStandard` MBean is not mirrored because our cascading service instance filters with the following object name pattern:

```
CascadedDomain:*
```

2. **Create four MBeans of the `SimpleStandard` class in following order:**

On the Master Agent:	`CascadedDomain:name=SimpleStandard,number=1`
On the Subagent:	`CascadedDomain:name=SimpleStandard,number=1`
	`CascadedDomain:name=SimpleStandard,number=2`
	`CascadedDomain:name=SimpleStandard,number=3`

When you are finished, reload the agent view on the master agent: you should see that the mirror MBeans for the last two have been created automatically. Look at the MBean view of either of these mirror MBeans on the master agent: their class name appears as `SimpleStandard`.

3. **In the master agent, set a new value for the** `State` **string attribute of all 3 of its** `SimpleStandard` **MBeans.**

   When you look at the corresponding MBeans in the subagent, you see that `number=2` and `number=3` were updated by their mirror MBean. However, `number=1` has not changed on the subagent: because it was created first in the master agent, it is not mirrored and exists separately on each agent.

4. **In the subagent, invoke the** `reset` **operation of all 3 of its** `SimpleStandard` **MBeans.**

   When you inspect the MBeans in the master agent, the values for `number=2` and `number=3` were reset. Remember that the HTML adaptor must get the values of attributes for displaying them, so they were correctly retrieved from the mirrored MBeans that we reset.

5. **In the master agent, unregister MBeans** `number=1` **and** `number=2`**, then update the agent view of the subagent.**

   In the subagent, you should still see the local version of `number=1`, but `number=2` has been removed at the same time as its mirror MBean.

   We are in a state where `number=1` is a valid MBean for mirroring but it isn't currently mirrored. This incoherence results from the fact that we did not have unique object names throughout the cascading hierarchy. Only new MBeans are mirrored dynamically, following the notification that signals their creation. We would have to stop and restart the master agent's cascading service MBean to mirror `number=1`.

6. **In the subagent, unregister MBeans** `number=1` **and** `number=3`**, then update the agent view on the master agent.**

   The mirror MBean for `number=3` was automatically removed by the cascading service, so none of the MBeans we added now remain.

7. **Invoke the** `stop` **operation of the** `CascadingAgent` **MBean in the master agent.**

   The last mirror MBean for the timer is removed from the master agent. The two agents are no longer connected.

8. **If you are finished with the agents, press** `<Enter>` **in both of their terminal windows to exit the applications.**

CHAPTER **15**

The Discovery Service

The discovery service enables you to discover Java Dynamic Management agents in a network. This service relies on a discovery client object which sends out multicast requests to find agents. In order to be discovered, an agent must have a registered discovery responder in its MBean server. Applications may also use a discovery monitor which detects when discovery responders are launched or stopped.

The combination of this functionality allows interested applications to establish a list of active agents and keep it current. In addition to knowing about the existence of an agent, the discovery service provides the version information from an MBean server's delegate and the list of communication MBeans that are currently registered. The discovery service uses the term *communicators* to designate a set of MBeans consisting of protocol adaptors and connector server objects.

Often, the discovery client and the discovery monitor are located in a manager application that wishes to know about the available agents. However, agent applications are free to use the discovery service since they may require such information for cascading (see "Cascading Agents") or for any other reason. To simplify using the discovery service in an agent, all of its components are implemented as MBeans.

The code samples in this topic are taken from the file in the `Discovery` directory located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Contents:

Active Discovery

In active discovery, the discovery client initiates searches for agents on the network. It involves a discovery client that sends the discovery request and a discovery responder in each agent that responds. Each instance of the responder supplies the return information about the agent in which it is registered. The return information is represented in the discovery client by a vector of discovery response objects.

The application containing the discovery client can initiate a search at any time. For example, it might do a search when it is first launched and periodically search again for information about the communicators which may have changed. For each search, the discovery client broadcasts a request and waits for the return information from any responders.

In the following sections, we describe each of these objects in further detail.

The Discovery Client

The `DiscoveryClient` class provides methods to discover agents. The active discovery operation sends a discovery request to a multicast group and waits for responses. These messages are proprietary and are not exposed to the user. Discovery clients can only discover agents listening on the same multicast group and port, so your design must coordinate this information between the discovery client and responders.

You can instantiate and perform searches from multiple discovery clients in a single application. Each discovery client can be configured to use different multicast groups or ports, allowing you to discover different groups of agents.

In our example, the discovery client is in an agent application: we register it as an MBean and interact with it through the MBean server.

CODE EXAMPLE 15–1 Instantiating and Initializing a Discovery Client

```
// build the DiscoveryClient MBean ObjectName
//
ObjectName discoveryClientMBeanObjectName =
    new ObjectName(domain + "name=myDiscoveryClient") ;

// Create, register and start the DiscoveryClient MBean
//
try {
    ObjectInstance discoveryClientObjectInstance =
        myMBeanServer.createMBean(
            "com.sun.jdmk.discovery.DiscoveryClient",
```

(continued)

```
            discoveryClientMBeanObjectName) ;
    myMBeanServer.invoke (discoveryClientMBeanObjectName,
        "start", null, null) ;

} catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
}
```

The default multicast group is 224.224.224.224 and the default port is 9000. These may be set to other values through the multicastGroup and multicastPort attributes, but only when the state of the discovery client is OFFLINE. Before initiating searches, you must call the discovery client's start method. This will create its multicast socket and join the multicast group used for broadcasting its discovery request.

The scope of the discovery request depends on the time-to-live used by the multicast socket. Time-to-live is defined by the Java class java.net.MulticastSocket to be a number between 1 and 255. By default, the time-to-live is 1, which corresponds to the machine's local area network. You can modify this value at any time by setting the discovery client's TimeToLive attribute.

By default, a discovery client waits for responses for one second after it has sent a discovery request. This period can be customized by setting a value in milliseconds for its TimeOut attribute. When setting this attribute, you should take into account estimated time for a round-trip of a network packet using the given time-to-live.

Performing a Discovery Operation

An application triggers a search operation by invoking the findMBeanServers or findCommunicators methods on an active DiscoveryClient object. Using the current settings, it will send the multicast request and block for the timeout period. At the end of the timeout period, these methods return the responses that were received.

Both methods return a vector of DiscoveryResponse objects. This class exposes methods for retrieving information about the MBean server and the registered communicator MBeans in the agent. The MBean server informations is the same as that exposed by that agent's MBean server delegate. The communicators are identified by ConnectorAddress objects and indexed by object name in a hash table.

Both search methods return the information about the agent's MBean server. The hash table of communicator MBeans is always empty for discovery responses returned by the `findMBeanServers` method. Otherwise, you can extract object names and protocol information from the hash table. One way of distinguishing the communicator MBeans is to rely on the default names provided by the `ServiceName` class.

Note - All discovery messages sent between components of the discovery service are compatible between applications running different versions of the Java SDK or different versions of the Java Dynamic Management Kit (4.*x* only). However, these different configurations are not compatible for subsequent management operations through connectors. You may use the `getImplementationVersion` method of the `DiscoveryResponse` object to determine both the Java SDK and product version numbers.

In our example, we request all information about the agents and use a simple subroutine to print out all information in the discovery responses.

CODE EXAMPLE 15–2 Performing a Discovery Operation

```
// Discover all JDMK agents with a registered discovery responder
//
Vector discoveryResponses = (Vector) myMBeanServer.invoke (
    discoveryClientMBeanObjectName,"findCommunicators", null, null) ;

echo("We have found " + discoveryResponses.size() + " JDMK agent(s): ");
for (Enumeration e = discoveryResponses.elements();
     e.hasMoreElements();) {
    DiscoveryResponse discoveryResponse =
        (DiscoveryResponse)e.nextElement() ;
    printDiscoveryResponse(discoveryResponse) ;
}

[...]

private void printDiscoveryResponse(DiscoveryResponse discoveryResponse) {

    // display information about the agent's MBean server
    //
    echo("\t MBeanServerId = " + discoveryResponse.getMBeanServerId())  ;
    echo("\t\t host name        = " + discoveryResponse.getHost())  ;
    [...]

    // display information about communicator objects, if any
    //
    if (discoveryResponse.getObjectList() != null) {
     for( Enumeration e= discoveryResponse.getObjectList().keys();
             e.hasMoreElements(); ) {
            ObjectName o = (ObjectName) e.nextElement();
            echo("\t\t Communicator name        = " + o ) ;
     }
```

(continued)

```
    }
}
```

On the agent side, the discovery responder automatically replies to discovery requests. Any active, registered responder in the same multicast group that is reached within the given time-to-live of the request will respond. It will automatically gather the requested information about its MBean server and send the response. The settings of the responder do not affect its automatic reply to discovery requests. In “The Discovery Responder” on page 234 we will cover how its settings control passive discovery.

In active discovery, the discovery client controls all parameters of a search it initiates, including the response mode of the discovery responder. The discovery client determines whether responses are sent back on a different socket (unicast) or sent to the same multicast group. The default is unicast: if you want to use the multicast response mode, set the `PointToPointResponse` attribute to `false` before initiating the discovery.

Unicast Response Mode

When the `PointToPointResponse` boolean attribute is `true`, the discovery client specifies unicast mode in its discovery requests. The responder will create a datagram socket for sending the response only to the discovery client. As shown in the following diagram, each responder will send its response directly back to the discovery client. The datagram socket used by each responder is bound to its local host address; this cannot be customized.

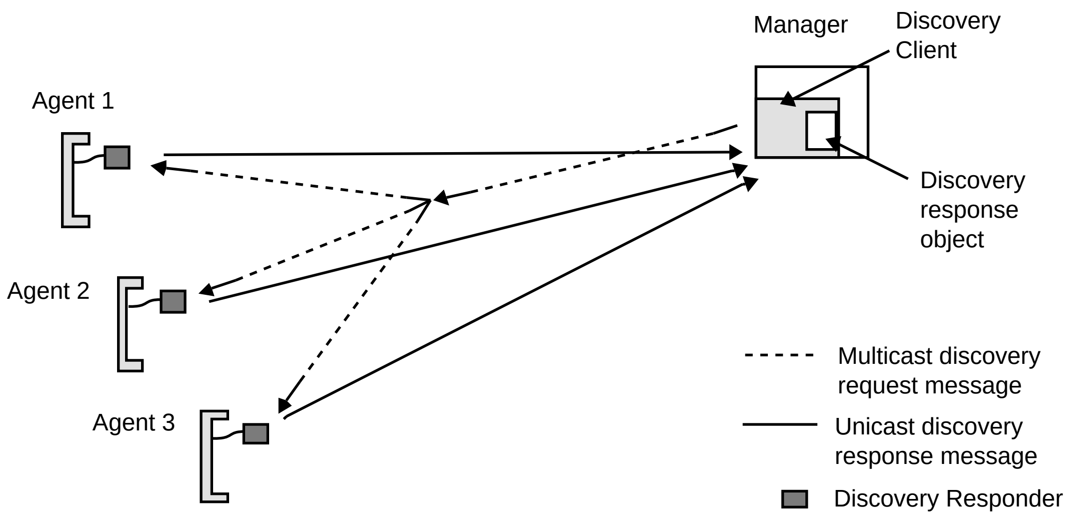

Manager
Discovery Client
Agent 1
Discovery response object
Agent 2
Multicast discovery request message
Agent 3
Unicast discovery response message
Discovery Responder

Figure 15–1 Unicast Response Mode

Multicast Response Mode

When the `PointToPointResponse` boolean attribute is `false`, the discovery client specifies multicast mode in its requests. The discovery responder will use the existing multicast socket to send response, broadcasting it to the same multicast group as the request. As shown in the following diagram, every member of the multicast group will receive the message, but only the discovery client can make use of its contents. Multicast mode avoids having to open another socket for the response, but all of the responses will create traffic in each application's socket.

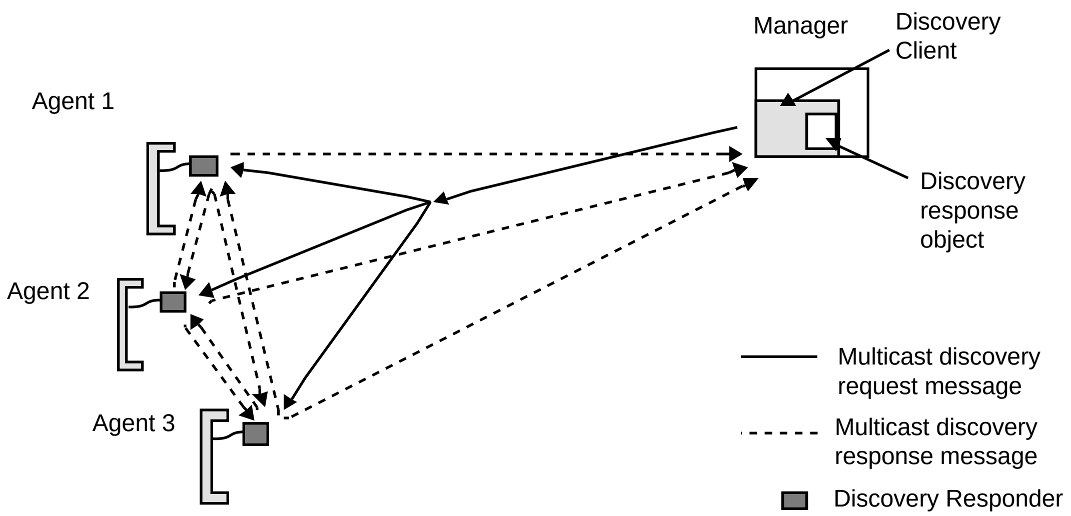

Manager
Discovery Client
Agent 1
Discovery response object
Agent 2
Agent 3
Multicast discovery request message
Multicast discovery response message
Discovery Responder

Figure 15–2 Multicast Response Mode

Passive Discovery

In passive discovery, the entity seeking knowledge about agents listens for their discovery responders being activated or deactivated. When discovery responders are started or stopped, they send out a proprietary message that contains all discovery response information. The `DiscoveryMonitor` object waits to receive any of these messages from the multicast group.

A discovery monitor is often associated with a discovery client. By relying on the information from both, you can keep an up-to-date list of all agents in a given multicast group.

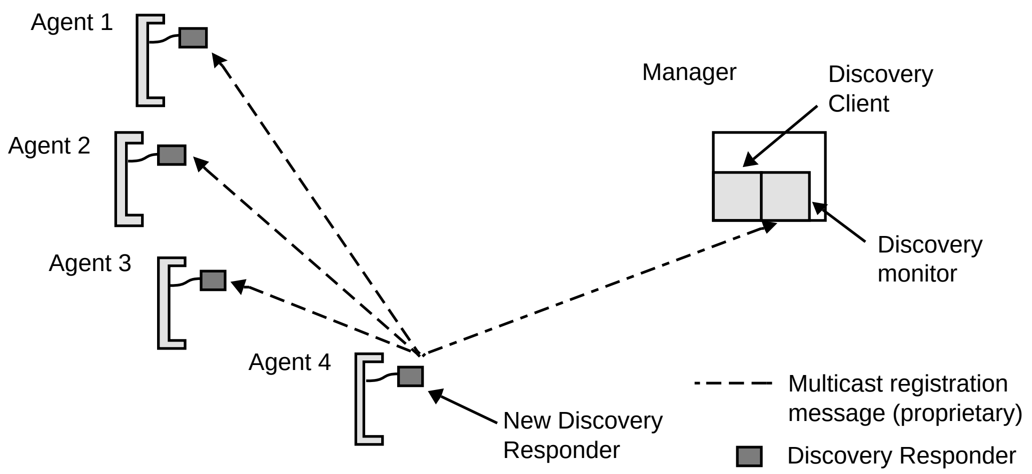


Figure 15–3 Passive Discovery of Discovery Responders

Therefore, configuring and starting the discovery responder is an important step to the overall discovery strategy of your applications.

The Discovery Responder

The agents that wish to be discovered must have an active `DiscoveryResponder` registered in their MBean server. The responder plays a role in both active and passive discovery:

- When the `start` operation of a discovery responder is invoked, it sends out a multicast message indicating that it has been activated.
- When the discovery responder is active, it automatically responds to discovery requests.
- When the responder's MBean is unregistered or its `stop` operation is invoked, it sends out a multicast message to indicate that it will be deactivated.

Both types of messages are proprietary and their contents are not exposed to the user. These messages contain information about the MBean server, its delegate's information and a list of communicator MBeans, unless not requested by the discovery client.

In our example we create the discovery responder in the MBean server and then activate it.

CODE EXAMPLE 15–3 Initializing a Discovery Responder

```
// Set the domain name for the demo
//
```

(continued)

(Continuation)

```
String domain = "DiscoveryDemo:" ;

// build the DiscoveryResponder MBean ObjectName
//
ObjectName discoveryResponderMBeanObjectName =
    new ObjectName(domain + "name=myDiscoveryResponder");

// Create and register the DiscoveryResponder MBean
//
try {
    ObjectInstance discoveryResponderObjectInstance =
        myMBeanServer.createMBean(
            "com.sun.jdmk.discovery.DiscoveryResponder",
            discoveryResponderMBeanObjectName) ;
    // we don't start the responder until our monitor is listening

} catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
}

[...]

try {
    myMBeanServer.invoke (discoveryResponderMBeanObjectName,
        "start", null, null) ;
} catch(Exception e) {
    echo("\tDiscoveryResponder MBean was already started.") ;
}
```

The discovery responder has attributes for exposing a multicast group and a multicast port. These attributes define a multicast socket that the responder will use to receive discovery requests. It will also send activation and deactivation messages to this multicast group. When sending automatic responses to discovery requests, the time-to-live is provided by the discovery client. The responder's time-to-live attribute is *only* used when sending activation and deactivation messages.

We use the default settings of the discovery responder which are the multicast group 224.224.224.224 on port 9000 with time-to-live of 1. In order to modify these values, you need to set the corresponding attributes *before* starting the discovery responder MBean. You may also specify them in the class constructor. If the responder is active, you will need to stop it before trying to set any of these attributes. In that way, it will send a deactivation message using the old values and then an activation message with the new values.

Discovery Monitor

The discovery monitor is a notification broadcaster: when it receives an activation or deactivation message from a discovery responder, it sends a discovery responder notification to its listeners. Once its parameters are configured and the monitor is activated, the discovery is completely passive. You can add or remove listeners at any time.

The `DiscoveryMonitor` MBean has multicast group and multicast port attributes that determine the multicast socket where it will receive responder messages. Like the other components of the discovery service, the default multicast group is 224.224.224.224 and the default port is 9000. You may specify other values for the group and port either in the constructor or through attribute setters when the monitor is off-line.

The discovery monitor in our example is registered as an MBean. We then add a listener through the MBean server as we would for any other notification broadcaster.

CODE EXAMPLE 15–4 Instantiating and Starting a Discovery Monitor

```
// build the DiscoveryMonitor MBean ObjectName
//
ObjectName discoveryMonitorMBeanObjectName =
    new ObjectName(domain + "name=myDiscoveryMonitor");

// Create, register and start the DiscoveryMonitor MBean
//
try {
    ObjectInstance discoveryMonitorObjectInstance =
        myMBeanServer.createMBean(
            "com.sun.jdmk.discovery.DiscoveryMonitor",
            discoveryMonitorMBeanObjectName) ;
    myMBeanServer.invoke (discoveryMonitorMBeanObjectName,
        "start", null, null);

} catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
}

// Add ourselves as a listener to the DiscoveryMonitor MBean
//
try {
    myMBeanServer.addNotificationListener(
        discoveryMonitorMBeanObjectName, this, null, null ) ;

} catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
}
```

The discovery monitor must be activated with the `start` operation before it will receive responder messages and send notifications. It will be stopped automatically if it is unregistered from its MBean server. If it is not used as an MBean, you should invoke its `stop` method before your application exits.

Discovery Responder Notifications

When it receives a responder's activation or deactivation message, the discovery monitor sends notification objects of the `DiscoveryResponderNotification` class. This notification contains the new state of the discovery responder (`ONLINE` or `OFFLINE`) and a `DiscoveryResponse` object with information from the agent where the responder is located.

The listener could use this information to update a list of agents in the network. In our example, the listener is the agent application itself, and the handler method only prints out the information in the notification.

CODE EXAMPLE 15–5 The Discovery Responder Notification Handler

```
public void handleNotification(
    javax.management.Notification notification, java.lang.Object handback) {

    // We know we will only receive this subclass, so we can do the cast
    DiscoveryResponderNotification discoveryNotif =
        (DiscoveryResponderNotification) notification;
    echo("\n>>> RECEIVED Discovery Notification FROM JDMK agent on host \"" +
        discoveryNotif.getEventInfo().getHost() + "\"");

    if ( discoveryNotif.getState().intValue() == DiscoveryMonitor.ONLINE ) {
        echo("\t DiscoveryResponder state = ONLINE");
    } else {
        echo("\t DiscoveryResponder state = OFFLINE");
    }
    DiscoveryResponse info =
        (DiscoveryResponse) discoveryNotif.getEventInfo();

    // internal method for displaying the discovery response information
    printDiscoveryResponse(info);
}
```

Running the Discovery Example

The *examplesDir*/Discovery directory contains the source file for the agent application that demonstrates the discovery service. This agent creates the discovery client, monitor and responder in the same MBean server, so it notifies itself when the responder is online and discovers its own presence. By launching several of these applications, you can see other agents being discovered.

Compile the file in this directory with the javac command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/Discovery/
$ javac -classpath classpath *.java
```

▼ Interacting with the Discovery Example

1. **To run the discovery example, launch the agent application with the following command:**

   ```
   $ java -classpath classpath DiscoveryAgent
   ```

   The agent application registers an HTML adaptor on the default port (8082). Then press <Enter> to initialize the discovery components. The discovery service is now ready, except for the discovery responder which hasn't been started yet.

2. **Press <Enter> a second time to activate the discovery responder.**

   The discovery monitor which has already been started receives the responder's activation message and triggers the notification listener. Our handler method prints out the information from the discovery response notification, indicating that the responder is ONLINE.

   Press <Enter> again to invoke the findCommunicators method of the discovery client and display its results. The content of the discovery response is identical to the previous information, in particular, it contains the object name of the HTML adaptor in our application.

   Leave the application running in this state.

3. **Now launch another DiscoveryAgent application in another terminal window or on another host. You must specify a non-default port number for the HTML adaptor if running on the same host, for example:**

```
$ java -classpath classpath DiscoveryAgent 8084
```

Both applications will be using the multicast group `224.224.224.224` and multicast port `9000`, so they will detect each other's presence. Go through the steps of this second application: you should see the first application receive the activation message of the second responder. Then, the active discovery should detect both agents. If your agents are running on the same host, you can tell them apart by their unique `MBeanServerId` values in the response information.

4. **Before stopping either of the discovery responders, you can interact with them through their HTML adaptors. Connect to the first agent by loading the following URL in a browser:**

   ```
   http://localhost:8082/
   ```

   You can see the MBeans for all of the discovery components. From the MBean view of the discovery responder, call its stop operation, modify its `TimeToLive` attribute, and restart it. The discovery monitor in the other agent should detect the two events and signal them to our listener.

   You can also initiate an active search through the discovery client's MBean: invoke either of its find methods. However, the HTML adaptor cannot display the contents of the resulting `DiscoveryResponse` object.

5. **When you are finished, stop the discovery responders on both agents and then the applications themselves by pressing `<Enter>` twice in each terminal window. The second agent to be stopped should see the discovery responder of the first being deactivated.**

PART V SNMP Interoperability

From the start, the Java Dynamic Management Kit was designed to be compatible with existing management standards. The Simple Network Management Protocol (SNMP) is the most widely used of these, and the Java Dynamic Management Kit provides the tools to integrate Java technology-based solutions into the SNMP world.

Using the SNMP interoperability with Java Dynamic Management solutions, you can develop agents that can be accessed through SNMP and through other protocols. You can also develop managers in the Java programming language which access both SNMP agents and Java Dynamic Management agents.

The agent and manager tools of the Java Dynamic Management Kit are completely independent. SNMP agents developed with this toolkit can be accessed by any SNMP manager, and the SNMP manager API lets you connect to any SNMP agent. The sample applications in this lesson use the toolkit on both agent and manager sides, but this is only one possible configuration.

This lesson contains the following topics:

- “Creating an SNMP Agent” demonstrates how the SNMP protocol adaptor makes a Java Dynamic Management agent also act as an SNMP agent. The MBeans generated by the `mibgen` tool represent SNMP MIBs which can be accessed by any SNMP manager connecting to the SNMP adaptor. This lets you implement your MIB through Java code and take advantage of the agent services. The example applications also demonstrate how traps are sent through the SNMP protocol adaptor.
- “Developing an SNMP Manager” shows you how to use the SNMP manager API to develop an SNMP manager in the Java programming language. An SNMP manager handles Java objects representing peers, parameters, sessions, and requests to access SNMP agents and perform management operations. Two examples demonstrate synchronous and asynchronous manager applications, and a third example shows how managers can communicate through inform requests.
- “Security Mechanisms in the SNMP Toolkit” groups all of the information about creating secure SNMP agents and managers. Access control lists (ACL) provide security on the agent side by restricting access to specific hosts based on their

community identification. Custom packet encoding between managers and agents is also possible, letting you develop any level of communication security you need.

- “Implementing an SNMP Proxy” gives an example of an SNMP proxy that you can use in an agent that uses the SNMP adaptor. An SNMP proxy is an object that handles remote MIBs in a sub-agent. The proxy acts as a single point-of-entry to let a manager access a whole hierarchy of sub-agents.

CHAPTER 16

Creating an SNMP Agent

Using the Java Dynamic Management Kit, you can create an agent application that is both an SNMP agent and a normal JMX agent. SNMP MIBs can be represented as MBeans and the SNMP protocol adaptor exposes them to SNMP managers. Only MBeans derived from MIBs may be managed through the SNMP protocol adaptor, but other managers may view and access them through other protocols.

The `mibgen` tool provided generates the code for MBeans that represent a MIB. Groups and table entries are represented as MBean instances which expose the SNMP variables through their attributes. The `mibgen` tool creates skeleton getters and setters which only read or write internal variables. In order to expose the behavior of your host machine or device, you need to implement the code that will access the host- or device-specific functionality.

The SNMP protocol adaptor interacts with your customized MIB MBeans to implement a compatible SNMPv1 and SNMPv2c agent. It also provides the mechanisms for sending traps and implementing both community-based access control and message-level data security (see “Security Mechanisms in the SNMP Toolkit”).

The program listings in this tutorial show only functional code: comments and output statements have been modified or removed for space considerations. However, all management functionality has been retained for the various demonstrations. The complete source code is available in the `Snmp/Agent` example directory located in the main *examplesDir* (see “Directories and Classpath” in the preface).

Contents:

- "Stand-Alone SNMP Agents" on page 257 demonstrates an alternative way of implementing an SNMP agent.

MIB Development Process

Here we describe the process for making MIBs manageable through the SNMP protocol adaptor of the Java Dynamic Management Kit. In our example, we demonstrate this process on a subset of the MIB-II defined by RFC 1213.

Once you have defined the MIB you want to manage in your SNMP agent you need to generate its MBean representation using the `mibgen` tool. This tool generates MBeans that represent the whole MIB, each of its groups and nested groups, and each of its table entries. This command-line tool and its output are fully described in the *Java Dynamic Management Kit 4.2 Tools Reference* guide.

The `mibgen` tool only generates the MBean structure to represent the MIB, it is up to the developer to implement the MIB functionality inside the generated classes. Our example will only give a simple implementation of the MIB-II for demonstration purposes. However, this will show you the way to extend the generated classes in order to provide your own implementation.

Generating MIB MBeans

To run the mibgen tool for our example, go to the *examplesDir*`/Snmp/Agent` directory and enter the following command:

```
$ mibgen -d . mib_II_subset.txt
```

This will generate the following files in the current directory:

- The MBean (by inheritance) for the whole MIB: `RFC1213_MIB.java`
- The MBean and its helper class for the `Snmp` group: `Snmp.java`, `SnmpMBean.java`, `SnmpMeta.java`
- The MBean and its helper class for the `System` group: `System.java`, `SystemMBean.java`, `SystemMeta.java`
- The MBean and its helper class for the `Interfaces` group: `Interfaces.java`, `InterfacesMBean.java`, `InterfacesMeta.java`
- The class representing the Interfaces table, and the MBean representing entries in the table: `TableIfTable.java`, `IfEntry.java`, `IfEntryMBean.java`, `IfEntryMeta.java`

- Classes representing enumerated types used in these groups: `EnumSnmpEnableAuthenTraps.java`, `EnumIfOperStatus.java`, `EnumIfAdminStatus.java`, `EnumIfType.java`
- The OID table for SNMP managers wanting to access this MIB: `RFC1213_MIBOidTable.java`

The MBean with the name of the MIB is a central administrative class for managing the other MBeans that represent the MIB groups and table entries. All of the other MBeans contain the SNMP variables as attributes of their management interface. The `mibgen` tool generates standard MBeans for the MIB, so attributes are implemented with individual getter and setter methods.

These MBeans are just skeletons, meaning that the attribute implementations only return a default value. You must implement the getters and setters of the attributes to read and write data with the correct semantic meaning of the SNMP variable.

Since SNMP does not support actions in MIBs, the only operations in these MBeans are *checkers* associated with the SNMP "Set" request in writeable variables. Again, these are skeleton methods which you must implement to do the checks that you require before the corresponding "Set" operation. You may add operations and expose them in the `MBean` interface, but the SNMP manager will not be able to access them. However, other managers will be able to call these operations if they are connected through another protocol.

Implementing the MIB

Our example only implements a fraction of the attributes, those that are used in this tutorial. The others are simply initialized with a plausible value. Using `DEFVAL` statements in our MIB, we could force `mibgen` to generate MBeans with user-defined default values for attributes. As this is not done in our example, `mibgen` provides a plausible default value according to the variable type.

Our implementations of MIB behavior are contained in the classes with the `Impl` suffix. These implementation classes extend those that are generated by `mibgen` so that we can regenerate them without overwriting our customizations.

Here is a summary of the implementation shown in the agent example:

- `InterfacesImpl.java` - adds a notification listener to the `IfTable` object, then creates two table entries with plausible values and adds them to the table; this class is associated with:
    - `TableEntryListenerImpl.java` - the listener for table notifications when entries are added or removed: it prints out the values of a new table entry and prints a message when an entry is removed
    - `IfEntryImpl.java` - implements a table entry and provides an internal method for switching the `OperStatus` variable that triggers a trap (see Code

Example 16–2); this method is not exposed in the `MBean` interface, so it is only available to the code of this agent application

- `SnmpImpl.java` - initializes and implements variables of the Snmp group; many of these are state variables of the SNMP agent, so we call the getter methods of the SNMP adaptor object to return the information
- `SystemImpl.java` - initializes the `System` group variables with realistic values

The `SnmpImpl.java` and `SystemImpl.java` files provide code that you may reuse when you need to implement these common SNMP groups.

The only class that we need to replace is `RFC1213_MIB`. Our implementation is located in `patchfiles/RFC1213_MIB.java` so that we can overwrite the one generated by `mibgen`. The main function of this MBean is to register the other MBeans of the MIB during its pre-registration phase. Our customization consists of specifying our `*Impl` classes when instantiating the group and table entry MBeans.

Compiling the MBeans and Agents

We replace the generated file with our implementation before compiling all of the classes in the *examplesDir*/`Snmp/Agent` directory. The classpath must contain the current directory (`.`):

```
$ cp -i patchfiles/RFC1213_MIB.java .
cp: overwrite ./RFC1213_MIB.java (yes/no)? y
$ javac -classpath classpath -d . *.java
```

We are now ready to look at the implementation of an SNMP agent and run the example applications.

The SNMP Protocol Adaptor

Once your MIBs are implemented as MBeans, your agent application needs an SNMP protocol adaptor in order to function as an SNMP agent. Since the SNMP adaptor is also an MBean, it can be created and started dynamically in your agent by a connected manager, or through the HTML adaptor. In our simple `Agent` example, we will launch it through the code of the agent application.

CODE EXAMPLE 16–1 The SNMP `Agent` Application

```
public class Agent {

    static SnmpAdaptorServer snmpAdaptor = null;

    public static void main(String args[]) {

        MBeanServer server;
        ObjectName htmlObjName;
        ObjectName snmpObjName;
        ObjectName mibObjName;
        ObjectName trapGeneratorObjName;
        int htmlPort = 8082;
        int snmpPort = 8085; // non-standard

        [...]
        try {
            server = MBeanServerFactory.createMBeanServer();
            String domain = server.getDefaultDomain();

            // Create and start the HTML adaptor.
            //
            htmlObjName = new ObjectName( domain +
                ":class=HtmlAdaptorServer,protocol=html,port=" + htmlPort);
            HtmlAdaptorServer htmlAdaptor = new HtmlAdaptorServer(htmlPort);
            server.registerMBean(htmlAdaptor, htmlObjName);
            htmlAdaptor.start();

            // Create and start the SNMP adaptor.
            //
            snmpObjName = new ObjectName(domain +
                ":class=SnmpAdaptorServer,protocol=snmp,port=" + snmpPort);
            snmpAdaptor = new SnmpAdaptorServer(snmpPort);
            server.registerMBean(snmpAdaptor, snmpObjName);
            snmpAdaptor.start();

            // The rest of the code is specific to our SNMP agent

            // Send a coldStart SNMP Trap (use port = snmpPort+1)
            // Trap communities are defined in the ACL file
            //
            snmpAdaptor.setTrapPort(new Integer(snmpPort+1));
            snmpAdaptor.sendV1Trap(0, 0, null);

            // Create the MIB-II (RFC 1213) and add it to the MBean server.
            //
            mibObjName = new ObjectName("snmp:class=RFC1213_MIB");
            RFC1213_MIB mib2 = new RFC1213_MIB();
            // The MBean will register all group and table entry MBeans
            // during its pre-registration
            server.registerMBean(mib2, mibObjName);

            // Bind the SNMP adaptor to the MIB
            mib2.setSnmpAdaptorName(snmpObjName);

            [...]
```

(continued)

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Needed to get a reference on the SNMP adaptor object
    static public SnmpAdaptorServer getSnmpAdaptor() {
        return snmpAdaptor;
    }
}
```

Launching the SNMP Adaptor

We launch the SNMP adaptor in the same way that we launch the HTML adaptor. First we create a meaningful object name for its MBean, then we instantiate the class with a constructor allowing us to specify a non-default port, we register the MBean with the MBean server, and we start the adaptor to make it active.

By default, the SNMP protocol adaptor uses the standard SNMP port 161. Since other applications may be using this port on a machine, our simple agent uses port 8085. When we connect to this agent, our SNMP manager will need to specify this non-standard port number.

Note - On certain platforms, applications also require super-user privileges to assign the default SNMP port 161. If your SNMP adaptor uses this port, its agent application will have to be launched with super-user privileges.

Creating MIB MBeans

Our agent application creates and manages one MIB, our subset of MIB-II. To do so, it instantiates the corresponding `RFC1213_MIB` MBean that has been generated by the `mibgen` tool (see "MIB Development Process" on page 244). We give it a meaningful object name and then we register it in the MBean server.

The registration process lets the MBean instantiate and register other MBeans that represent the groups of the MIB and the entries of its tables. The set of all these MBeans at the end of registration makes up the MBean representation of the MIB. If an SNMP manager later adds entries to a table, the MIB implementation will register the new entry MBean into the MBean server as well.

If you do not wish to expose a MIB through the MBean server, you do not have to register it. However, you still need to create all of its other MBean objects so that the SNMP adaptor can access all of its groups and table entries. The generated code provides the `init` method in the main MBean of the MIB. Calling this method will create all necessary MBean objects without registering them in the MBean server.

Binding the MIB MBeans

The SNMP adaptor does not interact with MBeans in the same way as the other connectors and adaptors. Because the SNMP data model relies on MIBs, only MBeans representing MIBs can be managed through SNMP. The SNMP adaptor does not interact with MBeans of a MIB through the MBean server, they must be explicitly bound to the instance of the SNMP adaptor.

After a MIB is instantiated, you must set the `SnmpAdaptorName` attribute of its main MBean to bind it to the SNMP adaptor. You can either call its `setSnmpAdaptorName` method directly or, if the MIB's MBean was registered in the MBean server, another management application may set the attribute through the MBean's exposed interface.

In the binding process, the SNMP adaptor obtains the root OID of the MIB. The adaptor uses this OID to determine which variables are implemented in the MIB's corresponding MBeans. In order for the SNMP adaptor to resolve a request on a given OID, the root OID of all bound MIBs must not overlap. This implies that no root OID may be equal to another or be a substring of another.

Even though the SNMP adaptor may be registered in the MBean server, the adaptor only makes MIBs visible to SNMP managers. Other MBeans in the agent cannot be accessed or even represented in the SNMP protocol. The SNMP manager is limited by its protocol: it cannot take full advantage of a Java Dynamic Management agent through the basic MIBs, and it does not have access to any other MBeans. In an advanced management solution, you could write a special MIB and implement it so that operations on its variables actually interact with the MBean server. This is left as an exercise for the reader.

Accessing a MIB MBean

Once the MBean representing a MIB has been instantiated and bound to the SNMP adaptor, it is accessible through the SNMP adaptor. SNMP managers can send requests to operate on the contents of the MIB. The SNMP adaptor interprets the SNMP management requests, performs the operation on the corresponding MBean and returns the SNMP response to the manager. The SNMP protocol adaptor is compatible with SNMPv1 and SNMPv2c.

The advantage of having an SNMP agent "inside" a Java Dynamic Management agent is that you can use the other communications protocols to interact with MIBs and manage the SNMP adaptor. Since both the registered MIBs and the adaptor are MBeans, they are exposed for management. In our simple agent, the MIB was registered, and you can view its MBeans in a web browser through the HTML protocol adaptor.

If our agent included other connectors, management applications could connect to the agent and also manage the MIB and the SNMP adaptor. A non-SNMP manager could instantiate new MIB objects, bind them to the SNMP adaptor and operate on the exposed attributes and operations of the MIB.

Non-SNMP managers may operate on the variables of a MIB, getting and setting values, regardless of any SNMP manager that might also be accessing them through the SNMP adaptor. When dealing with a table, however, they may not create new table entry MBeans without adding them to the table. For example, in the `InterfacesImpl.java` class, we called the `addEntry` method of the `IfTable` object before registering the entry MBeans with the MBean server. This ensures that the new entries will be visible when an SNMP manager accesses the table.

In order for a non-SNMP manager to create a table entry, you must customize the table's group MBean to expose this functionality. Briefly, you would need to write a new method that instantiates and initializes the entry's MBean, adds the MBean to the table object, and registers the entry MBean in the MBean server. Advanced customization such as this is not covered in our example. In general, the designer of the agent and management applications is responsible for all coherency issues when accessing MIBs concurrently through different protocols and when adding table entries.

Managing the SNMP Adaptor

Non-SNMP managers can also control the SNMP agent through the MBean of the SNMP adaptor. Like the other communications MBeans, the port and other attributes can be modified when the SNMP adaptor is stopped. You can also get information about its state, and stop or restart it to control when it is online. These administrative attributes and operations are defined in the `CommunicatorServerMBean` interface.

The SNMP adaptor server also implements the `SnmpAdaptorServerMBean` interface to define its operating information. The SNMP protocol defines certain variables that SNMP agents must expose about their current state. For example, the SNMP adaptor provides methods for `getSnmpInPkts` and `getSnmpOutBadValues`. Non-SNMP managers can read these variables as attributes of the SNMP adaptor MBean.

The SNMP adaptor also exposes other operating information that is unavailable to SNMP managers. For example, the `ActiveClientCount` and `ServedClientCount` read-only attributes report on SNMP manager connections to this agent. The read-write `BufferSize` attribute lets you change the size of the message buffer, but only when the adaptor is not online. The adaptor MBean also exposes operations for sending traps or implementing your own security (see "Message-Level Security" on page 284).

Running the SNMP Agent Example

After building the example as described in “MIB Development Process” on page 244, launch the simple agent with the following command:

```
$ java -classpath classpath Agent nbTraps
```

For this run, set *nbTraps* to zero. You should see some initialization messages, including our notification listener giving information about the two table entries which are created. Access this agent’s HTML adaptor by pointing a web browser to the following URL: `http://localhost:8082/`. Through the HTML adaptor, you can see the MBeans representing the MIB:

- The `class=RFC1213_MIB` MBean in the `snmp` domain is the MBean representing the MIB; it contains a name and information about the SNMP adaptor to which the MIB is bound
- The `RFC1213_MIB` domain contains the MBeans for each group; both `name=Snmp` and `name=System` contain variables with values provided by our customizations
- The `ifTable` domain contains the entries of the Interfaces table
- The `trapGenerator` domain contains the class that sends traps periodically, as part of our sample MIB implementation

In any of these MBeans, you could write new values into the text fields of exposed attributes and click the “Apply” button. This will set the corresponding SNMP variable, and thereafter, SNMP managers will see the new value. This is an example of managing a MIB through a protocol other than SNMP.

For any SNMP agent application, you can turn on trace messages for the SNMP adaptor by specifying the `-DINFO_ADAPTOR_SNMP` property on the command line. The tracing mechanism is covered in the *Java Dynamic Management Kit 4.2 Tools Reference* guide and in the Javadoc API of the `Trace` class.

Type “Control-C” when you are finished viewing the agent.

Sending Traps

Agents can send unsolicited event reports to management application by using traps. The SNMP protocol adaptor can send both v1 and v2 traps, the difference being in the format of the corresponding PDU. Traps in the SNMP specification are not acknowledged by the management application, so agents do not know if traps are received.

Inform requests are acknowledged event reports, they are sent by entities acting in a manager role, according to RFC 1905. In the Java Dynamic Management Kit, both the SNMP adaptor and the classes of the SNMP manager API may send inform requests.

Manager-to-manager inform requests are described in“The Inform Request Example” on page 272. Agent-to-manger inform requests are demonstrated by the applications in the `Snmp/Inform/` example directory located in the main *examplesDir*

In this example, we demonstrate how our simple SNMP agent application can send traps. The customized class `IfEntryImpl` in the example directory extends the `IfEntry` class generated by `mibgen` in order to provide a method that switches the `IfOperStatus` variable and sends a trap. This is an example of customization of the generated code: an agent-side entity will switch the operation status, the MIB variable will be updated and a trap will be sent to SNMP managers.

CODE EXAMPLE 16–2 Sending a Trap in the `IfEntryImpl` Class

```
public void switchifOperStatus() {
    // implements the switch and then calls sendTrap indirectly
    [...]
}

// Method called after the variable has been switched
// Should be called with generic==2 (up) or 3 (down or testing)
public void sendTrap(int generic) {

    SnmpAdaptorServer snmpAdaptor = null;

    // Retrieve the reference of the SNMP protocol adaptor through
    // the static method of the Agent or StandAloneSnmpAgent class
    snmpAdaptor = Agent.getSnmpAdaptor();
    if (snmpAdaptor == null) {
        snmpAdaptor = StandAloneSnmpAgent.getSnmpAdaptor();
    }
    if (snmpAdaptor == null) {
        return;
    }

    // Create the variable bindings to send in the trap
    Vector varBindList = new Vector();

    SnmpOid oid1 = new SnmpOid("1.3.6.1.2.1.2.2.1.1." + IfIndex);
    SnmpInt value1 = new SnmpInt(IfIndex);
    SnmpVarBind varBind1 = new SnmpVarBind(oid1, (SnmpValue) value1);

    varBindList.addElement(varBind1);

    try {
        snmpAdaptor.sendV1Trap(generic, 0, varBindList);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

As the `sendTrap` method runs in a different thread, it needs to get a reference to the SNMP adaptor instance. Here we call the static methods of our two possible agent implementations. This code is specific to these agents and is only an example of how to retrieve this information.

In order to simulate a live operation status, we invent the `LinkTrapGenerator` class which will switch the status periodically. It is an MBean which contains a thread which loops endlessly. The interval period between traps and the number of the table entry can be modified through the MBean's attributes.

CODE EXAMPLE 16–3 The Thread of the Link Trap Generator

```
public void run() {
    int remainingTraps = nbTraps;
    while ((nbTraps == -1) || (remainingTraps > 0)) {
        try {
            sleep(interval);
        } catch (Exception e) {
            e.printStackTrace();
        }
        triggerTrap();
        remainingTraps--;
    }
}

public void triggerTrap() {
    // get the entry whose status we will switch
    IfEntryImpl ifEntryImpl = InterfacesImpl.find(ifIndex);
    if (ifEntryImpl == null) {
        errors++;
        return;
    }
    ifEntryImpl.switchifOperStatus();
    successes++;
}
```

To run the trap generator, the example application instantiates and registers a `LinkTrapGenerator` MBean. During its registration, this MBean starts the thread, sending a trap every two seconds by default.

CODE EXAMPLE 16–4 Starting the Trap Generator Example

```
// Create a LinkTrapGenerator (specify the ifIndex in the object name)
//
String trapGeneratorClass = "LinkTrapGenerator";
int ifIndex = 1;
trapGeneratorObjName = new ObjectName("trapGenerator" +
    ":class=LinkTrapGenerator,ifIndex=" + ifIndex);
LinkTrapGenerator trapGenerator = new LinkTrapGenerator(nbTraps);
```

(continued)

```
server.registerMBean(trapGenerator, trapGeneratorObjName);

[...] // Press <Enter> to start sending traps

trapGenerator.start();
```

Specifying the Trap Destination

There are several methods in the SNMP protocol adaptor for sending traps to remote managers. They differ in their method signatures, depending upon whether or not you need to specify the destination host. When no host is specified, the SNMP protocol adaptor relies on the trap group definition in access control lists (ACL), as described below.

In all cases, traps are sent to the port specified by the current value of the `TrapPort` attribute on the `SnmpAdaptorServer` MBean. In our simple agent, we set the trap port to 8086, but this can be changed at any time by a custom MIB implementation or a management application.

Using an ACL Trap Group

This is the method that was used in Code Example 16–2 to send traps, along with its v2 equivalent (see the Javadoc API for a description of the parameters):

- `sendV1Trap( int generic, int specific, java.util.Vector varBindList )`
- `sendV2Trap( SnmpOid trapOid, java.util.Vector varBindList )`

Using these methods, you must first define the trap group in an access control list. See "Access Control Lists (ACL)" on page 280 for a formal definition of the trap group and instructions for defining the ACL file when starting the agent. By default, these lists are file-based, but you may implement other mechanisms, as described in "Custom Access Control" on page 284.

In this example we provide the following template file:

CODE EXAMPLE 16–5 Trap Group of the `jdmk.acl` File

```
acl = {
  …
}

trap = {
  {
  trap-community = public
  hosts = yourmanager
  }
}
```

The trap group lists all of the hosts to which the SNMP protocol adaptor will send every trap. A community definition associates a community name with a list of hosts specified either by their hostname or by their IP address. All hosts in a community definition will receive the trap in a PDU identified by the community name.

Note - Since access control and trap recipients share the same file, you must fully define the access control when you want to send traps using the ACL mechanism.

Given this definition, traps will be sent to a host called *yourmanager*, and the community string of the trap PDU would contain the value `public`. By adding community definitions to this file, you can specify all hosts which will receive traps along with the community string for each host or group of hosts.

If the ACL file is not defined, or if the trap group is empty, the default behavior of these methods is to send a trap only to the localhost.

Specifying the Hostname Directly

The other two methods of the SNMP protocol adaptor, one for each trap version, let you send a trap to a specified recipient:

- `sendV1Trap( java.net.InetAddress address, java.lang.String cs, … )`
- `sendV2Trap( java.net.InetAddress address, java.lang.String cs, … )`

In both cases, these methods take an address and a community string, in addition to the version-specific trap information. The `address` is an `InetAddress` object which is usually instantiated by its static methods `getLocalHost` or `getByName`. The latter method returns a valid `InetAddress` object when given a string representing a hostname or IP address.

The `cs` parameter is the community string, a name that the agent and manager exchange to help identify one another. The string given will be used as the community when sending the trap PDU.

Either one of these methods sends a trap to a single manager using a single community string. The ACL trap group mechanism is better suited to sending traps to multiple managers, though it requires the setup of a trap group. Note that even if a trap group is in use, the two methods above only send one trap to the specified host address.

Traps in the Agent Example

Before launching the SNMP agent again, edit the `jdmk.acl` file to replace the occurrences of *yourmanager* with the name of a host running an SNMP manager. You then have two options for launching the simple agent:

- By first copying the ACL file to the configuration directory where it is automatically detected:

```
$ cp jdmk.acl installDir/SUNWjdmk/jdmk4.2/JDKversion/etc/conf
$ java -classpath classpath Agent nbTraps
```

- Or by specifying the ACL file as a property when launching the agent

```
$ java -Djdmk.acl.file=examplesDir/Snmp/Agent/jdmk.acl \
       -classpath classpath Agent nbTraps
```

In these commands, *nbTraps* is the number of traps that the agent will send. Set it to a small integer to avoid too much output. If you omit this parameter, traps will be sent continuously.

If you don't have an SNMP manager or a second host, don't copy the ACL file or specify it as a property. In the absence of the trap-community definitions, the traps will be addressed to the trap port on the local host. And even if no manager is running, we can still see the agent sending the traps. See "SNMP Trap Handler" on page 266 in the SNMP manager topic for details about receiving traps.

▼ Interacting with the Trap Generator

1. **Access this agent's HTML adaptor by pointing a web browser to the following URL:** `http://localhost:8082/`**. Click on the** `class=LinkTrapGenerator,ifIndex=1` **MBean in the** `trapGenerator` **domain.**

   Through the HTML adaptor, you can see the MBean representing the trap generator object. You can modify its attributes to change the table entry that it operates on and to change the interval between traps.

2. **Change the trap interval to** `10000` **so that traps are sent every 10 seconds.**

3. **Go back to the agent view and click on the** `ifEntry.ifIndex=1` **MBean in the** `ifTable` **domain. Set the reload period to 10, and click the "Reload" button.**

   You should see the effect of the trap generator which is to switch the value of the `IfOperStatus` variable. It is our implementation of the table entry which sends a trap when this status is changed.

4. **Go back to the agent view and click on the** `name=Snmp` **MBean in the** `RFC1213_MIB` **domain. Scroll down to see the** `SnmpOutPkts` **and** `SnmpOutTraps` **variables.**

   These variables should be the only non zero values, if no manager has connected to the SNMP agent. The `Snmp` group shows information about the SNMP adaptor, and we can see how many traps have been sent since the agent was launched.

5. **Type** `<Control-C>` **when you are finished interacting with the agent.**

The `LinkTrapGenerator` MBean is not manageable through the SNMP adaptor because it is not part of any MIB. It is an example of another MBean providing some control of the SNMP agent, and this control can be exercised by other managers connecting through other protocols. This shows that designing an SNMP agent application involves both the implementation of the MIB functionality and, if desired, the implementation of other dynamic controls afforded by the JMX architecture and the services of the Java Dynamic Management Kit.

Stand-Alone SNMP Agents

The design of the SNMP protocol adaptor and of the MBeans generated by `mibgen` give you the option of creating an SNMP agent that is not a Java Dynamic Management agent.

This stand-alone agent has no MBean server and thus no possibility of being managed other than through the SNMP protocol. The application must instantiate all MIBs that the SNMP agent will need, as it will be impossible to create them through another manager. The advantage of a stand-alone agent is the reduced size of the application, in terms of memory usage.

CODE EXAMPLE 16–6 The `StandAloneSnmpAgent` Example

```
import com.sun.jdmk.Trace;
import com.sun.jdmk.comm.SnmpAdaptorServer;

public class StandAloneSnmpAgent {

    static SnmpAdaptorServer snmpAdaptor = null;

    public static void main(String args[]) {

        // Parse command line and enable tracing
        [...]

        try {
            // The agent is started on a non standard SNMP port: 8085
            int port = 8085;
            snmpAdaptor = new SnmpAdaptorServer(port);

            // Start the adaptor
            snmpAdaptor.start();

            // Send a coldStart SNMP Trap
            snmpAdaptor.sendV1Trap(0, 0, null);

            // Create the MIB you want in the agent (ours is MIB-II subset)
            RFC1213_MIB mib2 = new RFC1213_MIB();

            // Initialize the MIB so it creates the associated MBeans
            mib2.init();

            // Bind the MIB to the SNMP adaptor
            mib2.setSnmpAdaptor(snmpAdaptor);

            // Optional: create a LinkTrapGenerator
            int ifIndex = 1;
            LinkTrapGenerator trapGen = new LinkTrapGenerator(ifIndex);
            trapGen.start();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Needed to get a reference on the SNMP adaptor object
    static public SnmpAdaptorServer getSnmpAdaptor() {
        return snmpAdaptor;
    }
```

(continued)

(Continuation)

```
}
```

As this example demonstrates, the stand-alone agent uses exactly the same MIB MBeans, with the same customization, as our other agents. However, instead of registering them in the MBean server, they are only instantiated. And whereas the registration process creates all subordinate MBeans of the MIB, now we must call its `init` method explicitly.

The `init` method performs the same function as the `preRegister` method, only it does not register the MBean with the MBean server. Each of the group MBeans then has two constructors, one with and one without a reference to the MBean server. When table entries are added dynamically, the corresponding object only registers the new entry's MBean if the MBean server reference is non-null; that is, only if the MBean is not instantiated in a stand-alone agent.

The `mibgen` tool automatically generates both the pre-registration methods and the `init` methods in the MIB MBeans. Therefore, no special action is necessary to use them in either a regular agent or a stand-alone agent. If you use a stand-alone agent for memory considerations, you can remove the registration process from the generated MBean and only customize the "init" process.

In our example, we have applied the customizations to both processes so that the MIB can be used by either agent. In the following code, customizations are noted with `MODIF_` comments:

CODE EXAMPLE 16–7 Customizations in the Generated `RFC1213_MIB.java` File

```
public class RFC1213_MIB extends SnmpMib implements Serializable {

    // Default constructor. Initialize the Mib tree
    public RFC1213_MIB() {
        mibName = "RFC1213_MIB";
    }

    // Initialization of the MIB with no registration in the MBean server
    public void init() throws IllegalAccessException {
        // Allow only one initialization of the MIB
        if (isInitialized == true) {
            return ;
        }

        // Initialization of the "Snmp" group
```

(continued)

(Continuation)

```
        {
            SnmpMeta meta = new SnmpMeta((SnmpMib)this);
// MODIF_BEGIN
            //meta.setInstance(new Snmp((SnmpMib)this));
            meta.setInstance(new SnmpImpl((SnmpMib)this));
// MODIF_END
            root.registerNode("1.3.6.1.2.1.11", (SnmpMibNode)meta);
        }

        // Initialization of the other groups
        [...]

        isInitialized = true;
    }

    // Initialization of the MIB with AUTOMATIC REGISTRATION
    // in the MBean server
    public ObjectName preRegister(MBeanServer server, ObjectName name)
        throws Exception {

        // Allow only one initialization of the MIB
        if (isInitialized == true) {
            throw new InstanceAlreadyExistsException();
        }

        // Initialize MBeanServer information
        this.server = server;

        // Initialization of the "Snmp" group
        {
            SnmpMeta meta = new SnmpMeta((SnmpMib)this);
// MODIF_BEGIN
            //Snmp instance = new Snmp((SnmpMib)this, server);
            Snmp instance = new SnmpImpl((SnmpMib)this, server);
// MODIF_END
            meta.setInstance(instance);
            root.registerNode("1.3.6.1.2.1.11", (SnmpMibNode)meta);
            server.registerMBean(
                instance, new ObjectName(mibName + ":name=Snmp"));
        }

        // Initialization of the other groups
        [...]

        isInitialized = true;
        return name;
    }

    private boolean isInitialized = false;

}
```

(continued)

After the MIB is initialized, it only needs to be bound to the SNMP adaptor, as in the other agents; except that in the stand-alone case, we use the `setSnmpAdaptor` method which takes a direct reference to the SNMP adaptor instead of an object name. That is all you need to do when programing a stand-alone SNMP agent.

Running the Stand-Alone Agent Example

Launch the stand-alone agent with the following command:

```
$ java -classpath classpath StandAloneSnmpAgent nbTraps
```

If you haven't copied the `jdmk.acl` file to the configuration directory, add the following property to your command line:
`-Djdmk.acl.file=`*`examplesDir`*`/Snmp/Agent/jdmk.acl`

You should see the same initialization messages as with the simple agent. Then, you should see the agent sending out a trap every two seconds. If you have an SNMP manager application, you can send requests to the agent and receive the traps. See "Developing an SNMP Manager" for example applications you can use.

The only limitation of a stand-alone agent is that you cannot access or manage the SNMP adaptor and MIB MBeans in the dynamic management sense. However, the SNMP adaptor still relies on the ACL file for access control and traps, unless you have customized the ACL mechanism, and you can implement other security schemes as described in "Message-Level Security" on page 284.

Type `<Control-C>` when you are finished running the stand-alone agent.

CHAPTER **17**

Developing an SNMP Manager

The Java Management extensions specify the SNMP manager API for implementing an SNMP manager application in the Java programming language. This API is covered in the JMX specification and in the Javadoc API provided with the Java Dynamic Management Kit (see "Related Books" in the preface for more information). Here we explain the example applications that use this API.

The SNMP manager API can be used to access any SNMP agent, not just those developed with the Java Dynamic Management Kit. It is compatible with both SNMP v1 and v2, and it includes mechanisms for handling traps. It lets you program both synchronous managers which block while waiting for responses and multi-threaded asynchronous managers that don't. Managers may also communicate with other managers using inform requests and responses.

The complete source code for these applications is available in the `Snmp/Manager` and `Snmp/Inform` example directories located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Contents:

- "The Synchronous Manager Example" on page 264 shows the simplest way to program an SNMP manager in the Java programming language.
- "The Asynchronous Manager Example" on page 268 demonstrates the advantages of a manager that doesn't block when sending request.
- "The Inform Request Example" on page 272 shows how two managers can exchange management information.

The Synchronous Manager Example

The synchronous SNMP manager is the simplest to program: the manager sends a request to an agent (peer) and waits for the answer. During the wait, the manager is blocked until either a response is received or the timeout period expires.

The SNMP manager API allows two ways of referring to variables when issuing requests:

- By OID (for example, “1.3.6.1.2.1.11.29”)
- Or by name (“SnmpOutTraps” in this case)

Referring directly to OIDs requires no setup but makes code less flexible. The advantages of using variable names are simplified coding and the independence of manager code when custom MIBs are modified. The SNMP manager API supports variable names by storing a description of the MIB it will access in the static `SnmpOid` object.

In order to refer to variable names, the manager needs to initialize this description with an OID table object. The OID table is instantiated from the `SnmpOidTableSupport` class generated by the `mibgen` tool when “compiling” the MIB. Since this support class is regenerated whenever the MIB is recompiled, the new MIB definition will be automatically loaded into the manager when it is launched (see the code example below).

The SNMP manager API specifies the `SnmpPeer` object for describing an agent, and the `SnmpParameters` object for describing its read-write communities and its protocol version (SNMPv1 or SNMPv2). The `SnmpSession` is an object for sending requests and we can associate a default peer to it. The session instance has an `SnmpOptions` field which we can use to set multiplexing and error fixing behavior.

Note - The objects specified by the SNMP manager API are not MBeans and cannot be registered in an MBean server to create a manager that could be controlled remotely. However, you could write an MBean that uses these classes to retrieve and expose information from SNMP agents.

A manager can contain any number of peers, one for each agent it wishes to access, and any number of sessions, one for each type of behavior it wishes to implement. Once the peers and the sessions are initialized, the manager can build lists of variables and send session requests to operate on them. The session returns a request object, and the manager calls its `waitForCompletion` method with the desired timeout delay.

Finally, the manager analyzes the result of the request, first to see if there were any errors, then to extract the data returned by the request.

Here is the code of the `main` method of the `SyncManager` application. It applies all of the above steps to execute a very simple management operation.

CODE EXAMPLE 17–1 The SyncManager Example

```
// read the command line parameters
String host = argv[0];
String port = argv[1];

// Specify the OidTable containing all the MIB II knowledge
// Use the OidTable generated by mibgen when compiling MIB II
//
SnmpOidTableSupport oidTable = new RFC1213_MIBOidTable();
SnmpOid.setSnmpOidTable(oidTable);

SnmpPeer agent = new SnmpPeer(host, Integer.parseInt(port));

// When creating the parameter object, you can specify the
// read and write community to be used when querying the agent.
SnmpParameters params = new SnmpParameters("public", "private");
agent.setSnmpParam(params);

SnmpSession session = new SnmpSession("SyncManager session");

// When invoking a service provided by the SnmpSession, it
// will use the default peer if none is specified explicitly
session.setDefaultPeer(agent);

// Create a listener and dispatcher for SNMP traps:
// SnmpEventReportDispatcher will run as a thread and
// listens for traps in UDP port = agent port + 1
SnmpEventReportDispatcher trapAgent =
    new SnmpEventReportDispatcher(Integer.parseInt(port)+1);
// TrapListenerImpl will receive a callback
// when a valid trap PDU is received.
trapAgent.addTrapListener(new TrapListenerImpl());
new Thread(trapAgent).start();

// Build the list of variables you want to query.
// For debugging, you can associate a name to your list.
SnmpVarbindList list= new SnmpVarbindList(
    "SyncManager varbind list");

// We want to read the "sysDescr" variable.
list.addVariable("sysDescr.0");

// Make the SNMP get request and wait for the result.
SnmpRequest request = session.snmpGet(null, list);
boolean completed = request.waitForCompletion(10000);

// Check for a timeout of the request.
if (completed == false) {
    java.lang.System.out.println(
        "Request timed out. Check reachability of agent");
    java.lang.System.exit(0);
}

// Check if the response contains an error.
int errorStatus = request.getErrorStatus();
if (errorStatus != SnmpDefinitions.snmpRspNoError) {
```

(continued)

```
    java.lang.System.out.println("Error status = " +
        SnmpRequest.snmpErrorToString(errorStatus));
    java.lang.System.out.println("Error index = " +
        request.getErrorIndex());
    java.lang.System.exit(0);
}

// Now we can extract the content of the result.
SnmpVarbindList result = request.getResponseVbList();
java.lang.System.out.println("Result: \n" + result);

[...] // Wait for user to type enter. Traps will be handled.

// End the session properly and we're done
session.destroySession();
java.lang.System.exit(0);
```

In this SNMP manager application, we demonstrate how to implement and enable a trap listener for the traps sent by the agent. First we need to instantiate an `SnmpEventReportDispatcher` object. Then we add our listener implementation through its `addTrapListener` method, and finally we start its thread. Trap listeners can be implemented in any manager using the SNMP manager API, not only synchronous managers.

SNMP Trap Handler

A trap handler for the SNMP manager is an object that implements the `SnmpTrapListener` interface in the `javax.management.snmp.manager` package. When this object is bound as a listener of an `SnmpEventReportDispatcher` object, its methods will be called to handle trap PDUs.

A trap listener is not a notification listener because the dispatcher is not a notification broadcaster. The listener has callback methods that the work in the same manner, but they are given objects that represent traps, not instances of the `Notification` class.

The interface defines two methods, one for processing SNMPv1 traps and the other for SNMPv2 traps. Trap PDU packets have already been decoded by the dispatcher, and these methods handle an object representation of the trap: `SnmpPduTrap` objects for v1 and `SnmpPduRequest` objects for v2. In our implementation, we are only interested in v1 traps, and we just print out the trap information fields.

CODE EXAMPLE 17–2 The `SnmpTrapListener` Implementation

```
public class TrapListenerImpl implements SnmpTrapListener {

    public void processSnmpTrapV1(SnmpPduTrap trap) {
        java.lang.System.out.println(
            "NOTE: TrapListenerImpl received trap :");
        java.lang.System.out.println(
            "\tGeneric " + trap.genericTrap);
        java.lang.System.out.println(
            "\tSpecific " + trap.specificTrap);
        java.lang.System.out.println(
            "\tTimeStamp " + trap.timeStamp);
        java.lang.System.out.println(
            "\tAgent address " + trap.agentAddr.stringValue());
    }

    public void processSnmpTrapV2(SnmpPduRequest trap) {
        java.lang.System.out.println("NOTE: Trap V2 ignored");
    }
}
```

Running the `SyncManager` Example

In the *examplesDir*/`Snmp/Manager` directory, we first need to generate the OID table description of MIB-II that our manager will access. Then we compile the example classes. To set up your environment, see “Directories and Classpath” in the preface.

```
$ mibgen -mo mib_II.txt
[output omitted]
$ javac -classpath classpath -d . *.java
```

Make sure that no other agent is running on port 8085, and launch the simple SNMP agent in *examplesDir*/`Snmp/Agent`. See “MIB Development Process” on page 244 if you have not already built and run this example.

Here we give commands for launching the applications from different Unix terminal windows running the Korn shell. In the first window, enter the following commands:

```
$ cd examplesDir/Snmp/Agent
$ java -classpath classpath Agent nbTraps
```

If you will also be running the asynchronous manager example with this agent, omit the *nbTraps* parameter. The agent will then send traps continuously and they can be seen in both managers. Otherwise, specify the number of traps to be sent to the manager. Wait until the manager is started to send the traps.

Now we can launch the manager application in another window to connect to this agent. If you wish to run the manager on a different host, replace `localhost` with the name of the machine where you launched the agent.

```
$ cd examplesDir/Snmp/Manager
$ java -classpath classpath SyncManager localhost 8085
SyncManager::main: Send get request to SNMP agent on localhost at port 8085
Result:
[Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
Value : SunOS sparc 5.7]
```

Here we see the output of the SNMP request, it is the value of the `sysDescr` variable on the agent.

Now press `<Enter>` in the agent's window: you should see the manager receiving the traps it sends. Leave the agent running if you are going on to the next example, otherwise remember to stop it by typing `<Control-C>`.

The Asynchronous Manager Example

The asynchronous SNMP manager lets you handle more requests in the same amount of time because the manager is not blocked waiting for responses. Instead, it creates a request handler object which runs as a separate thread and processes several responses concurrently. Otherwise, the initialization of peers, parameters, sessions, options, and dispatcher is identical to that of a synchronous manager.

CODE EXAMPLE 17–3 The `AsyncManager` Example

```
// read the command line parameters
String host = argv[0];
String port = argv[1];

// Use the OidTable generated by mibgen when compiling MIB-II.
SnmpOidTableSupport oidTable = new RFC1213_MIBOidTable();

// Sample use of the OidTable.
SnmpOidRecord record = oidTable.resolveVarName("udpLocalPort");
java.lang.System.out.println(
    "AsyncManager::main: variable = " + record.getName() +
        " oid = " + record.getOid() + " type = " + record.getType());

// Initialize the SNMP Manager API.
SnmpOid.setSnmpOidTable(oidTable);

// Create an SnmpPeer object for representing the agent
SnmpPeer agent = new SnmpPeer(host, Integer.parseInt(port));

// Create parameters for communicating with the agent
SnmpParameters params = new SnmpParameters("public", "private");
agent.setSnmpParam(params);

// Build the session and assign its default peer
SnmpSession session = new SnmpSession("AsyncManager session");
session.setDefaultPeer(agent);

// Same dispatcher and trap listener as in SyncManager example
SnmpEventReportDispatcher trapAgent =
    new SnmpEventReportDispatcher(Integer.parseInt(port)+1);
trapAgent.addTrapListener(new TrapListenerImpl());
new Thread(trapAgent).start();

// Build the list of variables to query
SnmpVarbindList list = new SnmpVarbindList("AsyncManager varbind list");
list.addVariable("sysDescr.0");

// Create a simple implementation of an SnmpHandler.
AsyncRspHandler handler = new AsyncRspHandler();

// Make the SNMP walk request with our handler
SnmpRequest request = session.snmpWalkUntil(
    handler, list, new  SnmpOid("sysServices"));

// Here you could do whatever processing you need.
// In the context of the example, we are just going to wait
// 4 seconds while the response handler displays the result.
Thread.sleep(4000);

[...] // Wait for user to type enter. Traps will be handled.

// End the session properly and we're done.
//
```

(continued)

(Continuation)

```
session.destroySession();
java.lang.System.exit(0);
```

The trap mechanism in this application is identical to the one presented in the `SyncManager` example. The event report dispatcher will receive traps and call the corresponding method of our `SnmpTrapListener` class.

In this example, the manager performs an `snmpWalkUntil` request which will give a response for each variable that it gets. The response handler will be called to process each of these responses.

The Response Handler

A response handler for an asynchronous manager is an implementation of the `SnmpHandler` interface. When a handler object is associated with a request, its methods are called when the agent returns an answer or fails to return an answer. In these methods, you implement whatever actions you wish for processing the responses to a request. Typically, these methods will extract the result of each request or the reason for its failure.

The timeout used by the request handler is the one specified by the `SnmpPeer` object representing the agent. The handler is also called to process any errors caused by the request in the session. This ensures that the manager application is never interrupted after issuing a request.

CODE EXAMPLE 17–4 The `SnmpHandler` Implementation

```
public class AsyncRspHandler implements SnmpHandler {

    // Empty constructor
    public AsyncRspHandler() {
    }

    // Called when the agent responds to a request
    public void processSnmpPollData( SnmpRequest request,
        int errStatus, int errIndex, SnmpVarbindList vblist) {

        java.lang.System.out.println(
            "Processing response: " + request.toString());
        java.lang.System.out.println(
```

(continued)

```
            "errStatus = " + SnmpRequest.snmpErrorToString(errStatus) +
            " errIndex = " + errIndex);

        // Check if a result is available.
        if (request.getRequestStatus() ==
            SnmpRequest.stResultsAvailable) {

            // Extract the result for display
            SnmpVarbindList result = request.getResponseVbList();
            java.lang.System.out.println(
                "Result = " + result.vbListToString());
        }
    }

    // Called when the agent fails to respond to a request
    public void processSnmpPollTimeout(SnmpRequest request) {

        java.lang.System.out.println(
            "Request timed out: " + request.toString());

        if (request.getRequestStatus() ==
            SnmpRequest.stResultsAvailable) {

            // The result is empty and will display an error message
            SnmpVarbindList result = request.getResponseVbList();
            java.lang.System.out.println(
                "Result = " + result.vbListToString());
        }
    }

    // Called when there is an error in the session
    public void processSnmpInternalError(SnmpRequest request,
        String errmsg) {

        java.lang.System.out.println(
            "Session error: " + request.toString());
        java.lang.System.out.println("Error is: " + errmsg);
    }
}
```

Running the `AsyncManager` Example

If you have not done so already, launch the simple SNMP agent in *examplesDir/*`Snmp/Agent`, after making sure that no other agent is running on port 8085. This manager also uses the OID table description (the `SnmpOidTableSupport` class) that we generated from the MIB for the synchronous manager. If you have not already done so, see “Running the `SyncManager` Example” on page 267 for instructions on how to do this.

If you do not have an SNMP agent still running, make sure that no other agent is running on port 8085 and launch one with the following command:

```
$ cd examplesDir/Snmp/Agent
$ java -classpath classpath Agent nbTraps
```

Specify the number of traps to be sent to the manager in the *nbTraps* parameter. Wait until the manager is started to send the traps.

In another terminal window, launch the manager application to connect to this agent. If you wish to run the manager on a different host, replace `localhost` with the name of the machine where you launched the agent.

```
$ cd examplesDir/Snmp/Manager
$ java -classpath classpath AsyncManager localhost 8085
```

You should then see the output of the `SnmpWalkUntil` request: the response handler method is called for each variable that is returned.

Press `<Enter>` in the agent's window to send traps and see the trap reports as they are received in the manager. When you are finished with the agent, don't forget to stop it by typing `<Control-C>` in its terminal window.

The Inform Request Example

The inform request is specified in SNMP v2 as a mechanism for sending a report and receiving a response. This functionality is implemented in the JMX SNMP manager API for transmitting management information from one SNMP manager to another.

Since SNMP managers both send and receive inform requests, the SNMP manager API includes the mechanisms for doing both. Roughly, inform requests are sent in the same way as other requests, and they are received in the same way as traps. Both of these mechanisms are explained in the following sections.

This simple example has two manager applications, one of which sends an inform request, and the other which listens for and replies to this request. No SNMP agents are involved in this exchange.

Sending an Inform Request

Like the other types of requests, the manager sends an inform request through a session. The only difference is that the peer object associated with the request should be an SNMP manager able to receive and reply to `InformRequest` PDUs.

You may associate a peer with a session by making it the default peer object. This is how we do it in this example. This means that if we don't specify a peer when sending requests, they are automatically addressed to our manager peer. Since sessions often have agent peers as a default, you can specify the manager peer as a parameter to the `snmpInform` method of the session object.

CODE EXAMPLE 17–5 Sending an Inform Request in `SimpleManager1`

```
// When calling the program, you must specify the hostname
// of the SNMP manager you want to send the inform to.
//
String host = argv[0];

// Initialize the port number to send inform PDUs on port 8085.
//
int port = 8085;

try {
    // Create an SnmpPeer object that represents the entity to
    // communicate with.
    //
    SnmpPeer peer = new SnmpPeer(host, port);

    // Create parameters to associate to the peer.
    // When creating the parameter object, you can specify the
    // read and write community to be used when sending an
    // inform request.
    //
    SnmpParameters params = new SnmpParameters(
        "public", "private", "public");

    // The newly created parameter must be associated to the peer.
    //
    peer.setSnmpParam(params);

    // Build the session. A session creates, controls and manages
    // one or more requests.
    //
    SnmpSession session = new SnmpSession("SimpleManager1 session");
    session.setDefaultPeer(peer);

    // Make the SNMP inform request and wait for the result.
    //
    SnmpRequest request = session.snmpInform(
        null, new SnmpOid("1.2.3.4"), null);
    java.lang.System.out.println(
        "NOTE: Inform request sent to SNMP manager on " +
        host + " at port " + port);
```

(continued)

```
    boolean completed = request.waitForCompletion(10000);

    // Check for a timeout of the request.
    //
    if (completed == false) {
        java.lang.System.out.println(
            "\nSimpleManager1::main: Request timed out. " +
            "Check reachability of agent");

        // Print request.
        //
        java.lang.System.out.println("Request: " + request.toString());
        java.lang.System.exit(0);
    }

    // Now we have a response. Check if the response contains an error.
    //
    int errorStatus = request.getErrorStatus();
    if (errorStatus != SnmpDefinitions.snmpRspNoError) {
        java.lang.System.out.println("Error status = " +
            SnmpRequest.snmpErrorToString(errorStatus));
        java.lang.System.out.println("Error index = " +
            request.getErrorIndex());
        java.lang.System.exit(0);
    }

    // Now we shall display the content of the result.
    //
    SnmpVarbindList result = request.getResponseVbList();
    java.lang.System.out.println("\nNOTE: Response received:\n" + result);

    // Stop the session properly before exiting
    session.destroySession();
    java.lang.System.exit(0);

} catch(Exception e) {
    java.lang.System.err.println(
        "SimpleManager1::main: Exception occurred:" + e );
    e.printStackTrace();
}
```

Before sending the request, the `snmpInform` method automatically adds two variables to the head of the `varbind` list which is passed in as the last parameter. These are `sysUpTime.0` and `snmpTrapOid.0`, in the order they appear in the list. These variables are mandated by RFC 1905 and added systematically so that the caller doesn't need to add them.

Like all other requests in a session, inform requests can be handled either synchronously or asynchronously in the sender. In our example, we process the inform request synchronously: the manager blocks the session while waiting for the completion of the request. In an asynchronous manager, you would need to implement a response handler as explained in "The Response Handler" on page 270, and then use it to process responses, as shown in Code Example 17–3.

Receiving Inform Requests

Managers receive inform requests as they do traps: they are unsolicited events that must be received by a dispatcher object. Unlike traps, an inform request requires a response PDU which, according to the SNMP specification, must contain the same variable bindings. Therefore, immediately after an inform request is successfully received and decoded, the `SnmpEventReportDispatcher` class automatically constructs and sends the inform response back to the originating host.

The manager application then retrieves the data in the inform request through a listener on the dispatcher. Inform request listeners are registered with the dispatcher object in the same way as trap listeners. The receiving manager in our example is very simple, since its only function is to create the dispatcher and the listener for inform requests.

CODE EXAMPLE 17–6 Receiving Inform Requests in `SimpleManager2`

```
//
 Initialize the port number to listen for incoming inform PDUs on port 8085.
//
int port = 8085;

try {

    //
 Create a dispatcher for SNMP event reports (SnmpEventReportDispatcher).
    // SnmpEventReportDispatcher is run as a thread and listens for informs
    // on the specified port.
    // Add our InformListenerImpl class as an SnmpInformListener.
    // InformListenerImpl will receive a callback when a valid trap
    // PDU is received.
    //
    SnmpEventReportDispatcher informDispatcher =
        new SnmpEventReportDispatcher(port);
    informDispatcher.addInformListener(new InformListenerImpl());
    new Thread(informDispatcher).start();

    // Note that you can use the same SnmpEventReportDispatcher object
    // for both incoming traps and informs.
    // Just add your trap listener to the same dispatcher, for example:
    //    informDispatcher.addTrapListener(new TrapListenerImpl());

    // Here we are just going to wait for inform PDUs.
    //
```

(continued)

(Continuation)

```
    java.lang.System.out.println("\nNOTE: Event report listener initialized");
    java.lang.System.out.println(
      "      and listening for incoming inform PDUs on port " + port + "...");

} catch(Exception e) {
    java.lang.System.err.println(
        "SimpleManager2::main: Exception occurred:" + e );
    e.printStackTrace();
}
```

The remaining step is to program the behavior we want upon receiving an inform request. To do this, we must write the `InformListenerImpl` class that we registered as an inform request listener in the previous code sample. This class implements the `SnmpInformListener` interface and its `processSnmpInform` method handles the incoming inform request.

Because the dispatcher automatically sends the inform response back to the originating host, the `SnmpInformListener` implementation does not need to do this. Usually this method will extract the variable bindings and take whatever action is necessary upon receiving an inform request. In our example, we simply print out the source and the contents of the inform request.

CODE EXAMPLE 17–7 The `InformListenerImpl` Class

```
import java.io.IOException;
import javax.management.snmp.SnmpPduRequest;
import javax.management.snmp.manager.SnmpInformListener;

/**
 * This class implements the SnmpInformListener interface.
 * The callback method processSnmpInform is called when a
 * valid inform PDU is received.
 */

public class InformListenerImpl implements SnmpInformListener {

    public void processSnmpInform(SnmpPduRequest inform) {

        // Display the received PDU.
        //
        java.lang.System.out.println("\nNOTE: Inform request received:\n");
        java.lang.System.out.println("\tType      = " +
            inform.pduTypeToString(inform.type));
        java.lang.System.out.println("\tVersion   = " + inform.version);
```

(continued)

```
        java.lang.System.out.println("\tRequestId = " + inform.requestId);
        java.lang.System.out.println("\tAddress   = " + inform.address);
        java.lang.System.out.println("\tPort      = " + inform.port);
        java.lang.System.out.println("\tCommunity = " +
            new String(inform.community));
        java.lang.System.out.println("\tVB list   = ");

        for (int i = 0; i < inform.varBindList.length; i++) {
            java.lang.System.out.println("\t\t" + inform.varBindList[i]);
        }

        // Our listener stop the manager after receiving its first
        // inform request
        java.lang.System.out.println(
            "\nNOTE: SNMP simple manager 2 stopped...");
        java.lang.System.exit(0);
    }
}
```

Running the Inform Request Example

The *examplesDir*/`Snmp/Inform` directory contains all of the files for the two manager applications, along with the `InformListenerImpl` class.

Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:

```
$ cd examplesDir/Snmp/Inform/
$ javac -classpath classpath *.java
```

To run the example, launch the inform request receiver with the following command. You may launch the application in another terminal window or on another machine.

```
$ java -classpath classpath SimpleManager2
```

Wait for this manager to be initialized, then launch the other manager with the following command. The *hostname* is the name of the machine where you launched the receiving manager, or **`localhost`**.

```
$ java -classpath classpath SimpleManager1 hostname
```

When the sender is ready, press `<Enter>` to send the inform request. You should see the contents of the request displayed by the receiving manager. Immediately afterwords, the sender receives the inform response containing the same variable bindings and displays them. Both manager applications then exit automatically.

CHAPTER **18**

Security Mechanisms in the SNMP Toolkit

Both the SNMP protocol adaptor and the SNMP manager API provide mechanisms for ensuring the security of management operations. Agents act as information servers, and access control is used to protect this information from unauthorized access. This topic covers the different ways that the SNMP protocol adaptor can limit access to the data in the agent.

Security in a manager involves positively identifying the source of management responses, to ensure that the expected agent answered the request. This amounts to securing the communication against falsification of data and usurpation of identity, which must both be performed at the communication level. Both the agent and the manager handle the communication packets and both provide hooks for implementing your own security scheme.

The complete source code for these examples is available in the `Snmp/Agent` directory located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Contents:

Access Control Lists (ACL)

For the SNMP adaptor, the Java Dynamic Management Kit provides access control based on the IP address and community of the manager's host machine. Information on the access rights for communities and host machines is stored in access control lists (ACL). The default implementation provided with the product uses an ACL file, but you may provide your own implementation as described in "Custom Access Control" on page 284.

The ACL mechanism can also be used to define the communities and managers to which the agent will send traps. When you rely on the ACL trap group, the agent will send traps to all hosts listed in the ACL file. See "Specifying the Trap Destination" on page 254 for the different ways that an agent application may sends traps.

The following code example gives the contents of the *examplesDir*`/Snmp/Agent/jdmk.acl` file used when running the SNMP example applications. When using it in the security examples, you should replace *yourmanager* with the complete IP address or hostname of the machine running your SNMP manager application.

CODE EXAMPLE 18–1 A Sample ACL File

```
acl = {
 {
 communities = public
 access = read-only
 managers = yourmanager
 }
 {
 communities = private
 access = read-write
 managers = yourmanager
 }
}

trap = {
  {
  trap-community = public
  hosts = yourmanager
  }
}
```

ACL File Format

An ACL file contains an `acl` group defining community and manager access rights and a `trap` group defining the community and hosts for sending traps.

Format of the `acl` Group

The `acl` group contains one or more access configurations.

```
acl = {
   access1
   access2
      ...
   accessN
}
```

Each access configuration has the following format:

```
{
   communities = communityList
   access = accessRights
   managers = hostList
}
```

The *communityList* is a list of SNMP community names to which this access control applies. The community names in this list are separated by commas.

The *accessRights* specifies the rights to be granted to all managers connecting from the machines specified in the *hostList*. There are two possible values: either `read-write` or `read-only`.

The *hostList* item gives the host machines of the managers to be granted the access rights. The *hostList* is a comma-separated list of hosts, each of which can be expressed as any one of the following:

- A host name
- An IP address
- A subnet mask

Note - To distinguish between IP addresses and subnet masks in an ACL file, each integer in a subnet mask is separated by an exclamation mark (`!`) instead of a dot (`.`).

The set of all access configurations defines the access policy of the SNMP agent. A manager whose host is specified in a *hostList* and which identifies itself in one of the communities of the same configuration will be granted the permissions defined by the corresponding *accessRights*. A manager's host may appear in several access configurations provided it is associated with a different community list. This will define different access communities with different rights from the same manager.

A manager whose host-community identification pair does not appear in any of the access configurations will be denied all access. This means that PDUs from this manager will be dropped without being processed.

Format of the `trap` Group

The `trap` group specifies the hosts to which the agent will send traps if the ACL mechanism is used. This group contains one or more trap community definitions.

```
trap = {
   community1
   community2
   ...
   communityN
}
```

Each defines the association between a set of hosts and the SNMP community string in the traps to be sent to them. Each trap definition has the following format:

```
{
   trap-community = trapCommunityName
   hosts = trapHostList
}
```

The *trapCommunityName* item specifies a single SNMP community string. It will be included in the traps sent to the hosts specified in the `hosts` item.

The *trapHostList* item specifies a comma-separated list of hosts. Each host must be identified by its name or complete IP address.

When the SNMP protocol adaptor is instructed to send a trap using the ACL mechanism, it will send a trap to every host listed in the trap community definitions. If a host is present in more than one list, it will receive more than one trap, each one identified by its corresponding trap community.

Enabling Access Control

The default ACL mechanism provided with the Java Dynamic Management Kit relies on an ACL file to define the access rights and trap recipients. To enable access control with this mechanism, you must first write an ACL file to reflect the access and trap policy of your SNMP agent. Then, there are two ways to enable file-based access control, one way to modify the file in use and one way to disable access control.

The simplest way of enabling access control and traps is to ensure that an ACL file exists when the SNMP protocol adaptor MBean is instantiated. In order to be automatically detected, the ACL file must be named `jdmk.acl` and must be located in the configuration directory of the Java Dynamic Management Kit installation. On Unix systems with a standard installation of the product, the configuration directory is owned by root and requires super-user privileges in order to write or modify the ACL file.

Operating Environment	**Configuration Directory**
Solaris	*installDir*`/SUNWjdmk/jdmk4.2/`*JDKversion*`/etc/conf/`
Windows NT	*installDir*`\SUNWjdmk\jdmk4.2\`*JDKversion*`\etc\conf\`

In order for the application to locate the configuration directory, the classpath of the Java virtual machine running the agent must include the full path of the `jdmkrt.jar` file.

The other way of enabling file-based access control is to specify a different file through the `jdmk.acl.file` system property. The filename associated with the property will override any ACL file in the configuration directory. This property may be set programmatically, but it is usually done on the command line when launching your agent. For example, if the full pathname of your ACL file is *MyAclFile*, use this command to launch the agent with SNMP access control enabled:

```
$ java -classpath classpath -Djdmk.acl.file=MyAclFile MyAgent
```

If an ACL file exists, the access rights it defines apply to all management applications that access the agent through its SNMP adaptor. This includes managers on the agent’s local machine: the ACL groups must explicitly give permissions to `localhost` or the host’s machine name or IP address for such managers. If the ACL file does not exist when the SNMP adaptor is instantiated, either in the configuration directory or defined as a property, all SNMP requests will be processed, and traps will be sent only to the localhost.

The ACL file-based mechanism relies on the `JdmkAcl` class to provide the access control functionality. This is the class that is initialized with the contents of the ACL file. This class provides the `rereadTheFile` method to reset the access control and trap lists with the contents of the ACL file. This method will reload the same file that was used originally, regardless of any new property definitions. After you have updated the ACL file, call the following methods to update the access control lists:

```
// assuming mySnmpAdaptor is declared as an SnmpAdaptorServer object
JdmkAcl theAcl = (JdmkAcl)(mySnmpAdaptor.getIPAcl());
theAcl.rereadTheFile();
```

The `JdmkAcl` class that is used by default might not be suitable for all environments. For example, it relies on the `java.security.acl` package which is not available in the PersonalJava™ runtime environment. Therefore, one constructor of the `SnmpAdaptorServer` class lets you override this default, forcing the adaptor not to use access control, regardless of any existing ACL file. If you specify `false` for the `useAcl` parameter of this constructor, the SNMP adaptor won’t even search for an ACL file. In this case, no access control is performed, as if there were no ACL file: all SNMP requests will be processed, and traps will be sent only to the localhost. For security considerations, the use of access control cannot be toggled once the SNMP adaptor has been instantiated.

Custom Access Control

The `JdmkAcl` class which relies on an ACL file is the default access control mechanism in the SNMP adaptor. For greater adaptability, the `SnmpAdaptorServer` class has constructors that let you specify your own access control mechanism. For example, if your agent runs on a device with no file system, you could implement a mechanism which doesn't rely on the `jdmk.acl` file.

In order to instantiate an SNMP adaptor with your own access control, use one of the constructors which takes an `acl` parameter of the type `IPAcl`. Note that if this parameter's value is `null`, or if you use a constructor that doesn't specify an `acl` parameter, the SNMP adaptor will use the `JdmkAcl` class by default. If you want to instantiate an SNMP adaptor without access control, call the constructor with the `useAcl` parameter set to `false`.

Your access control mechanism must be a class that implements the `IPAcl` interface. This interface specifies the methods that the SNMP adaptor uses to check permissions when processing a request. If you instantiate the SNMP adaptor with your access control class, the adaptor will call your implementation of the access control methods. Again, for security reasons, the `IPAcl` implementation in use cannot be changed once the SNMP adaptor has been instantiated.

The `JdmkAcl` class implements the default access mechanism that uses the `jdmk.acl` file. It is also an implementation of the `IPAcl` interface, and it provides a few other methods, such as `rereadTheFile`, to control the ACL mechanism.

Message-Level Security

The `SecureAgent` example shows another level of security at the SNMP message level. Whereas the access control mechanism handles access rights to all MIBs for communities of manager hosts, message-level security lets you control how PDUs (Protocol Data Units) representing requests are encoded and decoded by the SNMP protocol adaptor.

The data in an SNMP message is stored in its raw form as a byte array. When receiving a message, the SNMP protocol adaptor must decode the data to obtain the corresponding request, and before sending a request, the adaptor must encode it as a message. By default, the basic encoding rules (BER) are used to translate back and forth between message and decoded PDU. The SNMP protocol adaptor provides a hook to let you implement your own encoding and decoding mechanism.

Message-level security relies on the following classes in the `javax.management.snmp` package:

- `SnmpPduPacket` class
- `SnmpMessage` class
- `SnmpPduFactory` interface

An `SnmpPduPacket` object represents the fully decoded description of an SNMP request. In particular, it includes the operation type (get, set, ...), the list of variables to which the operation applies, the request identifier, and the protocol version.

The `SnmpMessage` object is a partially decoded representation of the SNMP request. The type of the request, the variables and their values all remain encoded as a byte array. This object also contains the default BER encoding and decoding methods. The `SnmpMessage` class is derived from the `Message` syntax in RFC 1157 and RFC 1902.

The `SnmpPduFactory` interface defines the method signatures for encoding PDUs into messages and decoding messages into PDUs. By providing an implementation of this class, you can fully control the contents of messages and see the contents of packets before they are processed.

Both the packet and message classes also contain port and address information of the SNMP manager. When implementing your own security mechanism, you also have access to the contents of the PDU or message you are handling. This lets you implement security based on several factors:

- The host or community of the SNMP manager in a message before it is decoded
- The type or contents of a request after it is decoded
- Some encryption of the raw data

Because message-based security gives you access to all these different factors, you can perform elaborate filtering of incoming requests. For example, you could limit the access of certain SNMP managers to certain variables, or you could filter variable bindings, such as untrusted IP addresses, before they are assigned.

If your security is based on encryption of the message data, your manager must of course be using the same encryption. Because the SNMP classes are also part of the SNMP manager API, you can reuse the same encryption code in your manager if it is developed in the Java programming language. Security in the SNMP manager API is implemented in the `SnmpPeer` and `SnmpEventReportDispatcher` objects, see "SNMP Manager Security" on page 288 for more information.

If your security scheme is based only on the sender of the message or contents of the PDU, it can be applied unilaterally by the agent, without requiring any coordination with the manager application. This is what is demonstrated in the `SecureAgent` example.

Implementing the `SnmpPduFactory` Interface

In the SNMP protocol adaptor, the task of translating an `SnmpMessage` object into an `SnmpPduPacket` object is delegated to an object which implements the `SnmpPduFactory` interface. This interface defines two methods, one for decoding messages into PDUs and one for encoding PDUs into messages:

- `decodePdu` takes an `SnmpMessage` and should return a decoded `SnmpPduPacket` object; if it returns null or raises an exception, the incoming message is assumed to have failed the security check
- `encodePdu` takes an `SnmpPduPacket` and should return an encoded `SnmpMessage` to send

In our example, the `SnmpPduFactoryImpl` class implements the `decodePdu` method to reject messages if they originate from certain hosts. The list of hosts to refuse is passed to the class constructor at instantiation. The `encodePdu` method only does the standard BER encoding of outgoing messages.

CODE EXAMPLE 18–2 Implementing the `SnmpPduFactory` Interface

```
public class SnmpPduFactoryImpl implements SnmpPduFactory {

    private String[] hostNames;

    // HostNames is the array of the host names whose requests will be
    // refused by the agent
    public SnmpPduFactoryImpl(String[] hostNames) {
        this.hostNames = hostNames;
    }

    public SnmpPduPacket decodePdu(SnmpMessage msg)
        throws SnmpStatusException {

        // Get the sender’s hostname
        String from = msg.address.getHostName();
        for (int i = 0; i < hostNames.length; i++) {
            if (from.equals(hostNames[i])) {
                java.lang.System.out.println("Pdu rejected from " + from);
                return null;
            }
        }
        // If the host is accepted, we return the standard BER decoding
        return msg.decodePdu();
    }

    // No security when sending, just do the standard BER encoding
    public SnmpMessage encodePdu(SnmpPduPacket pdu, int maxPktSize)
        throws SnmpStatusException, SnmpTooBigException {

        SnmpMessage result = new SnmpMessage();
        result.encodePdu(pdu, maxPktSize);
        return result;
```

(continued)

```
    }
}
```

Beyond our simple check of the sender's hostname, our example relies on the standard BER encoding and decoding of the `SnmpMessage` class. Even if you choose to implement encryption, it can still be implemented on top of the standard BER for simplicity. In this case, you only need to encrypt the message's byte array after the standard encoding and decrypt it before the standard decoding.

When you implement the `encodePdu`, you must ensure that it also handles trap PDUs, by encoding them as they will be expected by the manager application (see "SNMP Manager Security" on page 288). For example, trap messages are decoded separately from response messages in applications based on the SNMP manager API.

Using a Custom PDU Factory

To use your custom PDU factory in your SNMP agent, you need to call the `usePduFactory` method of your `SnmpAdaptorServer` instance. First instantiate your PDU factory implementation and then pass it to this method. Your encoding and decoding scheme will then replace the standard one used by the SNMP protocol adaptor.

The `SecureAgent.java` file contains a simple agent like the one presented in "The SNMP Protocol Adaptor" on page 246. It only adds the call to force the SNMP adaptor to use the new PDU factory that we specify.

```
// Use SnmpPduFactoryImpl class for SnmpPduFactory to filter requests.
// Enter your list of refused hosts as arguments when launching this agent.
// The agent will reject requests coming from the specified hosts.
//
String[] refusedHosts = new String[args.length];
refusedHosts = args;
snmpAdaptor.usePduFactory(new SnmpPduFactoryImpl(refusedHosts));
```

The SNMP adaptor will then use this PDU factory to filter incoming requests based on their originating host. It will also encode all outgoing messages, including any traps that are sent, though it does nothing more than standard BER encoding. The secure agent example does not demonstrate traps, and the `LinkTrapGenerator` class is not written to function with the `SecureAgent` class. However, the `SnmpPduFactoryImpl` could be used as it is shown above in the SNMP `Agent` example.

Running the Secure Agent Example

You can only demonstrate the output of our custom PDU factory if you have an SNMP manager application which can connect to the secure agent. See “Developing an SNMP Manager” for example applications you can use.

The `SecureAgent` class takes command line arguments to create the list of hosts from which it will refuse SNMP requests. Use the following command to launch the secure agent example:

```
$ java -classpath classpath SecureAgent [host1..hostN]
```

Whenever one of the refused hosts sends a request, you should see the message displayed by our custom PDU factory implementation. Type “Control-C” when you are finished running the secure agent.

You can combine message-level security and access control defined by the presence of an ACL file. The ACL file indicates trusted hosts and communities from which messages are accepted. After they are accepted, they are decoded with the message-level security you provide. This lets you provide more precise security based on types of requests or the target variable, as well as any encryption.

SNMP Manager Security

Since the role of SNMP managers is that of a client, their security needs revolve around sending and receiving data safely. To do this, the SNMP manager API also relies on implementations of the `SnmpPduFactory` interface in order to control message-level encoding and decoding.

By default, the SNMP manager API relies on the standard BER encoding implemented by the `SnmpPduFactoryBER` class, which is the encoding used by default by all SNMP agents, not just those developed with the Java Dynamic Management Kit. You may change the encoding to use your own implementation of the `SnmpPduFactory` interface through the following two hooks:

- The `setPduFactory` method of an `SnmpPeer` instance lets you control how all requests are encoded and their responses decoded; because the custom PDU encoding is associated with a peer, you may have different encodings for the different peers that are accessed

- The `setPduFactory` method of an `SnmpEventReportDispatcher` instance lets you control how unsolicited inform requests and trap messages are decoded and the inform response encoded

Using these hooks, message-level security can be implemented regardless of whether the managers are synchronous or asynchronous. If you are using a symmetrical encryption of messages between your agent and manager, you may also reuse the same classes for your PDU factory implementation on both agent and manager sides, assuming they are both using the Java Dynamic Management Kit SNMP toolkit.

Otherwise, if your encryption is not symmetrical or if your agents do not use the SNMP protocol adaptor, your PDU factory implementation will necessarily be specific to the security scheme you choose to implement in your manager application.

The example applications do not cover the security features in the SNMP manager API. Please refer to the *Java Management Extensions SNMP Manager API* document and the Javadoc API for more details about using these hooks to implement manager security.

CHAPTER **19**

Implementing an SNMP Proxy

It is easier to manage a large number of SNMP agents when they have a hierarchical structure of master agents and sub-agents. Master agents concentrate and relay the information in their sub-agents and can provide their own specific information as well. Managers only communicate with the master agents and access the sub-agents transparently, as if the information actually resided in the master agent.

In order to do this, agents must contain an SNMP proxy for each sub-agent that they manage. The proxy is a Java class that looks like a MIB MBean to the agent, but which actually accesses the sub-agent to provide the information that is requested of it. In order to access sub-agents, the proxy object relies on the SNMP manager API.

The SNMP proxy is used in this simple example to allow a manager to access two MIBs through one agent. You can reuse the proxy class in your management solutions in order to implement any hierarchy of managers, master agents and sub-agents.

The source code for the proxy object and the sample application is available in the `Snmp/Proxy` example directory located in the main *examplesDir* (see "Directories and Classpath" in the preface).

Contents:

The Proxy Roles

As we saw in “MIB Development Process” on page 244, the MBeans that represent a MIB are generated from the MIB definition by the `mibgen` tool. This tool generates one main MBean representing the MIB and then one MBean for each group and each table entry. The main MBean extends the `SnmpMib` class whose implementation of the abstract `SnmpMibAgent` class processes a request on a MIB.

For example, if an agent receives a `get` request for a variable, it will call the `get` method that the main MBean inherits from `SnmpMibAgent`. The implementation of this method relies on the MIB structure to first find the MBean containing the corresponding attribute and to then call its getter to read the value of the variable.

A proxy is another implementation of the `SnmpMibAgent` class which, instead of resolving variables in local MBeans, reformulates an SNMP request, sends it to a designated sub-agent and forwards the answer that it receives. Since only the main MBean of a MIB is bound to the SNMP adaptor, we bind the proxy instead, and the master agent transparently exposes the MIB which actually resides in the sub-agent.

The Master Agent

The master agent needs to instantiate one SNMP proxy object for each sub-agent containing MIBs that it wishes to serve. The remote MIBs can be on any number of sub-agents, and the master agent can have several proxy objects. Sub-agents themselves may contain proxies: it is up to the designer to define the complexity of the agent hierarchy.

The master agent may also contain a mix of proxies and MBeans for other MIBs. In the proxy example, the master agent exposes the `DEMO` MIB through local MBeans and a subset of the `RFC1213` MIB through a proxy.

CODE EXAMPLE 19–1 The Master Agent of the Example

```
MBeanServerImpl server;
ObjectName snmpObjName;
ObjectName localMibObjName;
ObjectName remoteMibObjName;
int htmlPort = 8082;
int snmpPort = 8085;

// read sub-agent connection info from the command line
String host = argv[0];
String port = argv[1];

try {
```

(continued)

```
    server = MBeanServerFactory.createMBeanServer();
    String domain = server.getDefaultDomain();

    // Create and start the HTML adaptor.
    [...]

    // Create and start the SNMP adaptor.
    [...]

    // Create, initialize, and bind the local MIB Demo.
    //
    localMibObjName = new ObjectName("snmp:class=DEMO_MIB");
    server.registerMBean(localMib, localMibObjName);
    localMib.setSnmpAdaptorName(snmpObjName);

    // Create and initialize the SNMP proxy.
    //
    remoteMibObjName = new ObjectName("snmp:class=proxy");
    SnmpMibAgentImpl remoteMib = new SnmpMibAgentImpl();
    server.registerMBean(remoteMib, remoteMibObjName);
    remoteMib.initializeProxy(host, Integer.parseInt(port), "1.3.6.1.2.1");

    // Bind the MIB proxy to the SNMP adaptor
    //
    ((SnmpMibAgent)remoteMib).setSnmpAdaptorName(snmpObjName);

}
catch (Exception e) {
    e.printStackTrace();
    java.lang.System.exit(1);
}
```

We register the proxy object as for any other MBean and then initialize it. We call the `initialize` method of the proxy, giving the host and port of the sub-agent it must communicate with, and the root OID of the MIB or subset that it represents.

A single proxy object can serve several MIBs on a sub-agent, and in this case, the OID is the common prefix of all objects. However, the OIDs of all proxies and MIBs in an agent must be distinct, none may be a substring of another (see “Binding the MIB MBeans” on page 249). Finally, we bind the proxy to the SNMP adaptor just as we would a MIB MBean.

The Sub-Agent

The sub-agent in our example is a stand-alone agent which serves a subset of the `RFC1213` MIB. Since it implements no proxies of its own, it is just a plain agent which responds to SNMP management requests that happen to originate from a proxy object. Any SNMP manager could also send requests to this agent.

Stand-alone agents are covered in "Stand-Alone SNMP Agents" on page 257. As this stand-alone agent contains no code that is specific to its role as a sub-agent, we will not repeat its program listing here. The `StandAloneAgent.java` file only contains some extra code for reading its assigned port from the command line. We will use this to launch the agent on a known port to which the proxy can connect.

The Manager Application

The manager application is not affected by proxies in the agents to which it sends requests. It sends the requests to the session of the master agent, in the same way that it would a request for a MIB that is not served by a proxy. In fact, the SNMP manager cannot even distinguish between a MIB served by an agent and another MIB served through a proxy in the agent, except perhaps by analyzing the values returned.

There is one consideration for proxies, and that is the timeout interval. Since the proxy issues another request and potentially answers only at the end of its timeout, the manager must have a longer timeout. The manager should be designed with some knowledge of all sub-agents in a hierarchy, so that the timeout interval can take into account all proxy delays and the multiple communication times.

As with any manager application written with the SNMP manager API, the manager in the proxy example may use the OID table object to refer to the MIB variables by name. Here is the code to initialize the manager:

CODE EXAMPLE 19–2 Initialization of the SNMP Proxy Manager

```
String host = argv[0];
String port = argv[1];

// Initialize the SNMP Manager API.
// Specify the OidTables containing all the MIB Demo and MIB II knowledge.
//
 Use the OidTables generated by mibgen when compiling MIB Demo and MIB II.
//
SnmpOidDatabaseSupport oidDB = new SnmpOidDatabaseSupport();
SnmpOid.setSnmpOidTable(oidDB);
oidDB.add(new RFC1213_MIBOidTable());
oidDB.add(new DEMO_MIBOidTable());

SnmpPeer agent = new SnmpPeer(host, Integer.parseInt(port));
SnmpParameters params = new SnmpParameters("public", "private");
SnmpSession session = new SnmpSession("Manager session");
```

(continued)

```
// We update the time out to let the agent enough time
// to do his job before retry.
//
agent.setTimeout(100000);
agent.setSnmpParam(params);
session.setDefaultPeer(agent);
```

The rest of the manager application is the code for synchronous `get` and `set` requests, similar to the one shown in “The Synchronous Manager Example” on page 264.

The SNMP Proxy Implementation

The SNMP proxy is an extension of the abstract `SnmpMibAgent` which implements all of its abstract methods and can be instantiated. The proxy implements a synchronous SNMP manager that forwards the requests to the sub-agent. It does some error fixing for SNMPv2 requests but doesn’t claim to be extensive. The SNMP proxy implementation is only provided as an example and Sun Microsystems makes no claim as to its suitability for any particular usage.

Note - The implementation of the example proxy does not support the `getBulk` request nor the `check` method.

Before it is used, the proxy must be initialized with the hostname and port of the sub-agent. It uses this information to create the corresponding SNMP parameter object and SNMP peer object. It then creates three SNMP sessions:

- One for SNMPv1 requests
- One for SNMPv2 requests
- One for failed SNMPv2 requests that are retried as v1 requests

CODE EXAMPLE 19–3 Internal Initialization of the SNMP Proxy

```
public void initializeProxy(String h, int p,
                            String strOid, String name)
    throws UnknownHostException, SnmpStatusException {

    host = h;
    port = p;
    oid = strOid;
    mibName = name;

    // Initialization for SNMP v1 protocol.
    //
    SnmpParameters paramsV1 = new SnmpParameters("public", "private");
    paramsV1.setProtocolVersion(SnmpDefinitions.snmpVersionOne);
    SnmpPeer peerV1 = new SnmpPeer(host, port);
    peerV1.setSnmpParam(paramsV1);
    sessionV1 = new SnmpSession("SnmpMibAgentImpl session V1");
    sessionV1.setDefaultPeer(peerV1);

    // Using SNMP v1 protocol, errors are not fixed and
    // forwarded to the manager
    sessionV1.snmpOptions.setPduFixedOnError(false);

    // Initialization for SNMP v2 protocol.
    //
    SnmpParameters paramsV2 = new SnmpParameters("public", "private");
    paramsV2.setProtocolVersion(SnmpDefinitions.snmpVersionTwo);
    SnmpPeer peerV2 = new SnmpPeer(host, port);
    peerV2.setSnmpParam(paramsV2);
    // If we get an error, we don’t retry the request using SNMP v2,
    // but we try the request using SNMP v1
    peerV2.setMaxRetries(0);
    sessionV2 = new SnmpSession("SnmpMibAgentImpl session V2");
    sessionV2.setDefaultPeer(peerV2);
    // Using SNMP v2 protocol, the error is fixed
    //
    sessionV2.snmpOptions.setPduFixedOnError(true);

    // Initialization for SNMP v2 protocol simulated
    // using SNMP v1 protocol
    //
    sessionV2WithV1 = new SnmpSession("SnmpMibAgentImpl session V2 with V1");
    sessionV2WithV1.setDefaultPeer(peerV1);
    // Simulating SNMP v2 with SNMP v1 protocol, the error is fixed.
    //
    sessionV2WithV1.snmpOptions.setPduFixedOnError(true);
}
```

The proxy exposes the public methods for handling requests, and then implements this algorithm for reducing errors by internal methods. Roughly, the proxy must determine the version of the incoming request and handle it as promised. Version 1 requests that timeout or fail are dropped, and v2 requests that timeout or fail are retried as v1 requests. Here we only show the code for implementing the `get` method.

CODE EXAMPLE 19–4 Implementing a `get` Request in the Proxy

```
// the exposed method
public void get(SnmpMibRequest inRequest) throws SnmpStatusException {

    java.lang.System.out.println(
        "Proxy: Sending get request to SNMP sub-agent on " +
            host + " using port " + port);

    // Get the protocol version
    final int version = inRequest.getVersion();

    // Request using SNMP v1 protocol
    if (version == SnmpDefinitions.snmpVersionOne) {
        get(inRequest, version, sessionV1);
    }

    // Request using SNMP v2 protocol.
    if (version == SnmpDefinitions.snmpVersionTwo) {
        get(inRequest, version, sessionV2);
    }
}

// the internal implementation
private void get(SnmpMibRequest inRequest, int version, SnmpSession session)
    throws SnmpStatusException {

    // Construction of the SnmpVarBindList for the request.
    final SnmpVarbindList varbindlist =
        new SnmpVarbindList("SnmpMibAgentImpl varbind list",
                            inRequest.getSubList() );

    SnmpRequest request = null;
    try {
        request = session.snmpGet(null, varbindlist);
    }
    catch (SnmpStatusException e) {
        throw new SnmpStatusException(SnmpDefinitions.snmpRspGenErr, 0);
    }
    java.lang.System.out.println("\nRequest:\n" + request.toString());

    boolean completed = request.waitForCompletion(10000);
    if (completed == false) {

        // If the completion failed using SNMP v1, we give up
        if (version == SnmpDefinitions.snmpVersionOne) {
            java.lang.System.out.println(
                "\nRequest timed out: check reachability of sub-agent.");
            return;
        }

        // If the completion failed using SNMP v2, we try again using v1
```

(continued)

(Continuation)

```
        if (version == SnmpDefinitions.snmpVersionTwo) {
            get(inRequest, SnmpDefinitions.snmpVersionOne, sessionV2WithV1);
            return;
        }
    }

    // Check the request result
    int errorStatus = request.getErrorStatus();
    int errorIndex = request.getErrorIndex() + 1;
    if (errorStatus != SnmpDefinitions.snmpRspNoError) {

        // If there is an error status using v1, we throw an exception
        if (version == SnmpDefinitions.snmpVersionOne) {
            throw new SnmpStatusException(errorStatus, errorIndex);
        }
        // If there is an error status using v2, we try again using v1
        if (version == SnmpDefinitions.snmpVersionTwo) {
            get(inRequest, SnmpDefinitions.snmpVersionOne, sessionV2WithV1);
            return;
        }
    }
    // Get and display the returned values
    final SnmpVarbindList result = request.getResponseVarBindList();
    java.lang.System.out.println("\nResult: \n" +
                             result.varBindListToString());

    // Update the list parameter with the result
    // The varbinds in the result list are expected to be in the same
    // order as in the request, so we can safely loop sequentially
    // over both lists.
    Enumeration l = list.elements();
    for (Enumeration e = result.elements(); e.hasMoreElements();) {
        SnmpVarBind varres = (SnmpVarBind) e.nextElement();
        SnmpVarBind varbind = (SnmpVarBind) l.nextElement();
        varbind.value = varres.value;
    }
}
```

The complete list of methods that a proxy must implement are the same as for MIB MBeans represented by instances of the `SnmpMib` class:

- `long[] getRootOid()` - Gets the root object identifier of the MIB.
- `void get(SnmpMibRequest req)` - Processes a `get` operation.
- `void getNext(SnmpMibRequest req)` - Processes a `getNext` operation.
- `void getBulk(SnmpMibRequest req, int nonRepeat, int maxRepeat)` - Processes a `getBulk` operation.

- `void check(SnmpMibRequest req)` - Prepares a `set` operation.
- `void set(SnmpMibRequest req)` - Processes a `set` operation.

Running the SNMP Proxy Example

First, we generate the MBeans for our MIBs using the `mibgen` tool:

```
$ cd examplesDir/Snmp/Proxy/
$ mibgen -a -d . mib_II_subset.txt mib_demo.txt
```

Since the proxy uses the SNMP manager API, the master agent application needs the SNMP manager classes in its classpath, for both compilation and execution. In addition, if the proxy object uses the SNMP OID tables that are generated for its MIBs by the mibgen tool, these classes must also be found in the classpath.

These issues are resolved in our case by having all classes in the example directory and using dot (`.`) in our usual classpath. Replace the two MIB files with those from the `patchfiles` directory and then compile all of the classes in the example directory:

```
$ cp -i patchfiles/*_MIB.java .
cp: overwrite ./DEMO_MIB.java (yes/no)? y
cp: overwrite ./RFC1213_MIB.java (yes/no)? y
$ javac -classpath classpath -d . *.java
```

We can run all three applications on the same machine as long as we choose different port numbers. Here we give commands for launching the applications from the same terminal window running the Korn shell. On the Windows NT platform, you will have to launch each application in a separate window, in which case you will not see the sequence of the merged output.

▼ Running the SNMP Proxy Example

1. **First we launch the sub-agent; by default it will reply to port 8086, or we can specify a different one on the command line:**

```
$ java -classpath classpath StandAloneAgent 8090 &
Adding SNMP adaptor using port 8090

Initializing the MIB RFC1213_MIB
```

2. **Then we launch the master agent, giving it the sub-agent's hostname and port number:**

```
$ java -classpath classpath Agent localhost 8090 &
NOTE: HTML adaptor is bound on TCP port 8082
NOTE: SNMP Adaptor is bound on UDP port 8085

Initializing the MIB snmp:class=DEMO_MIB

Initializing the SNMP proxy snmp:class=proxy to query host localhost using
port 8090
```

3. **Now you can view the master agent through its HTML adaptor. In your web browser, go to the following URL:** `http://localhost:8082/`**.**

   The `class=proxy` MBean in the `snmp` domain is the proxy object, but we cannot see the MBean that it represents. In order to manage the actual MIB, we would have to connect to the sub-agent, but in our example it is a stand-alone and therefore unreachable.

   The `name=Demo` MBean in the `DEMO_MIB` domain is the MIB that is served locally. If we click on its name, we can see its initial values.

4. **Finally, we launch the manager application, giving it the master agent's hostname and port:**

```
$ java -classpath classpath Manager localhost 8085
```

The manager will run through its requests to the master agent. If the output is in the correct order, there are messages from the manager issuing a request, and then the proxy which relays a request for two of the variable to the sub-agent. The proxy then prints the result it received for the two variables, then the manager receives the final response with all of the variables, including the same two that the proxy just forwarded.

5. **In the** `name=Demo` **MBean on the web browser, you should see the new values that were set in the DEMO_MIB as part of the manager's set operation.**

6. **Don't forget to stop the agent applications with the following commands (just use "Control-C" if they are in separate terminal windows):**

```
$ fg
java [...] Agent localhost 8090 <Control-C>
^C$ fg
java [...] StandAloneAgent 8090 <Control-C>
^C$
```