Java™ Management Extensions Instrumentation and Agent Specification, v1.0

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Public Release 3, May 2000

Please
Recycle

Adobe PostScript

Contents

Preface

This document provides an introduction to the *Java™ Management extensions* and then gives the JMX™ instrumentation and agent specifications that define these extensions. It is not intended to be a programming guide nor a tutorial, but rather a comprehensive specification of the architecture, design patterns and programming interfaces for these components.

The complete JMX specification is composed of this document and the corresponding Javadoc™ API which completely defines all programming objects.

Who Should Use This Book

The primary focus of this specification is to define the extensions to the Java programming language for all actors in the software and network management field. Also, programmers who wish to build devices, applications, or implementations which conform to JMX will find this specification useful as a reference guide.

Before You Read This Book

This specification assumes a working knowledge of the Java programming language and of the development environment for the Java programming language. It is essential to understand the JDK™ software and be familiar with system or network management. A working knowledge of the JavaBeans™ model is also helpful.

All object diagrams in this book use the Unified Modeling Language (UML) for specifying the objects in the Java programming language that comprise the JMX specification. This allows a visual representation of the relation between classes and their components. For a complete description of UML see:

http://www.rational.com/uml/resources/documentation/

How This Book Is Organized

Chapter 1 provides an overview of the scope and goals of the JMX specification. It explains the overall management architecture and presents the main components.

Part I "JMX Instrumentation Specification"

Chapter 2 presents the standard and dynamic MBeans, their characteristics and design patterns, their naming scheme, the notification model and the MBean metadata classes.

Chapter 3 presents the optional open MBean components and their Java classes.

Chapter 4 presents the model MBean concept and the Java classes on which it relies.

Part II "JMX Agent Specification"

Chapter 5 presents the architecture of the JMX agent and its components.

Chapter 6 defines the foundation classes used by the interfaces of the JMX agent components.

Chapter 7 defines the MBean server and the methods available to operate on managed objects, including queries that retrieve specific managed objects.

Chapter 8 defines the m-let (management applet) service which loads classes and libraries dynamically from a URL over the network.

Chapter 9 defines the monitoring service which observes the value of an attribute in another managed object and signals when thresholds are reached.

Chapter 10 defines the timer service which provides scheduling capabilities.

Chapter 11 defines the relation service which creates relation types and maintains relations between MBeans based on these types.

Related Information

IBM has contributed the specification, reference implementation, and compatibility test suites for model MBeans in the JMX Instrumentation specification, including Chapter 4 of this document.

The definitive specification for all Java objects and interfaces of the JMX specification is the Javadoc API generated for these classes. It is available as a compressed archive file at the following URL:

http://java.sun.com/aboutJava/communityprocess/first/jsr003/index.html

The JMX additional management protocol APIs are described in separate specifications. They are given in separate Java Specification Requests (JSRs) which are being developed through the Java Community Process℠:

- The JSR for the *SNMP Manager API* is currently being reviewed; it will appear at: http://java.sun.com/aboutJava/communityprocess/accepted.html
- *JSR-000048 WBEM Services Specification* has been accepted and is described at: http://java.sun.com/aboutJava/communityprocess/jsr/jsr_048_wbem.html
- *JSR-000070 IIOP Protocol Adapter for JMX Specification* has been accepted: http://java.sun.com/aboutJava/communityprocess/jsr/jsr_070_jmxcorba.html

Additional information can be found on the JMX web site:

http://java.sun.com/products/JavaManagement

Typographic Conventions

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

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of literals and the underlined text of URLs (Universal Resource Locators).	Set the value of the `name` descriptor. See the `http://java.sun.com` web site
`AaBbCc123`	The names of interfaces, classes, fields or methods in the Java programming language.	The `Timer` class implements the `TimerMBean` interface.
AaBbCc123	Book titles, new words or terms, or words to be emphasized.	Read Chapter 6 in the *User's Guide.* These are called *class options.* You *must* implement this interface.

CHAPTER 1

Introduction to the JMX™ Specification

The Java™ Management extensions (also called the JMX™ specification) define an architecture, the design patterns, the APIs, and the services for application and network management in the Java programming language. This chapter introduces all of these elements, presenting the broad scope of these extensions. The rest of this document, along with its corresponding Javadoc API, represents the first phase of the JMX specification.

The JMX specification provides Java developers across all industries with the means to instrument Java code, create smart Java agents, implement distributed management middleware and managers, and smoothly integrate these solutions into existing management systems. In addition, the JMX specification is referenced by a number of Java APIs for existing standard management technologies.

The JMX architecture is divided into three levels:

- Instrumentation level
- Agent level
- Distributed services level

This chapter gives an introduction to each of these levels and describes their basic components.

In addition, the JMX specification will be associated with a number of Java APIs for existing, standardized management protocols. These APIs are independent of the three-level model, yet they are essential because they enable JMX applications in the Java programming language to link with current management technologies. Currently, two management protocol APIs are being developed as separate Java Specification Requests (JSRs) through the Java Community Process:

- The JSR for the *SNMP Manager API* is currently being reviewed
- *JSR-000048 WBEM Services Specification* for CIM / WBEM manager and provider APIs has been accepted

Benefits of the JMX Architecture

Through an implementation of the JMX specification, the JMX architecture provides the following benefits:

- **Enables Java applications to be managed without heavy investment**

  The JMX architecture relies on a core managed object server that acts as a 'management agent' and can run on most Java-enabled devices. This allows Java applications to be manageable with little impact on their design. A Java application simply needs to embed a managed object server and make some of its functionality available as one or several Manageable Beans registered in the object server; that is all it takes to benefit from the management infrastructure.

  JMX provides a standard way to enable manageability for any Java based application, service or device. For example, Enterprise JavaBeans™ (EJB) applications can conform to the JMX architecture to become manageable.

- **Provides a scalable management architecture**

  Every JMX agent service is an independent module that can be plugged into the management agent, depending on the requirements. This component-based approach means that JMX solutions can scale from small footprint devices to large telecommunications switches and beyond.

  The JMX specification provides a set of core agent services. Additional services will be developed by conformant implementations, as well as by the integrators of the management solutions. All of these services can be dynamically loaded, unloaded, or updated in the management infrastructure.

- **Integrates existing management solutions**

  JMX smart agents are capable of being managed through HTML browsers or by various management protocols such as SNMP and WBEM. The JMX API are open interfaces that any management system vendor can leverage.

  The JMX specification has spurred the definition of an SNMP manager API, a WBEM client API, and a TMN manager API. These associated specifications are being developed separately through the Java Community Process. They will provide the interfaces needed to write applications that manage SNMP agents or act as a Java/SNMP proxy, access a CIM Object Manager, or respond to a TMN manager.

- **Leverages existing standard Java technologies**

  Whenever needed, the JMX specification will reference existing Java specifications such as Java Naming and Directory Interface™ (JNDI), Java Database Connectivity API (JDBC™), Java Transaction Services (JTS), or others.

- **Can leverage future management concepts**

  The APIs of the JMX specification can implement flexible and dynamic management solutions through the Java programming language which can leverage emerging technologies. For example, JMX solutions can use lookup and discovery services and protocols such as Jini™ connection technology, Universal Plug'n'Play (Upnp), and the Service Location Protocol (SLP).

  In a demonstration given by Sun Microsystems, Jini provides spontaneous discovery of resources and services on the network, which are then managed by through a JMX application. The combination of these two capabilities is called Sun Spontaneous Management™.

- **Defines only the interfaces necessary for management**

  The Java Management extensions are not designed to be a general purpose distributed object system. Although it provides a number of services designed to fit into a distributed environment, these are focused on providing functionality for managing networks, systems, and applications.

Scope of this Specification

The JMX specification defines an architecture for management and a set of APIs that describe the components of this architecture. These APIs cover functionality, both on the manager and on the agent side, that compliant implementations will provide to the developer of management applications.

This JMX specification document addresses the first two levels of the management architecture. These parts are:

- The instrumentation specification
- The agent specification

This phase of the JMX specification only provides a brief overview of the distributed services, to show how and where they interact with the other two levels. Other related information may also appear in this specification, and it will be clearly stated when this information is outside the scope of the present specification.

The additional management protocol APIs are described in separate documents which are released independently through the Java Community Process, see "Related Information" on page xv.

Reference Implementation

The *reference implementation* (RI) is the first working application of the JMX specification, as mandated by the Java Community Process for defining extensions to the Java programming language. The RI for both the instrumentation and agent specifications has been developed by Sun Microsystems, Inc., in its role as the JMX specification lead.

The RI allows developers of JMX-based management solutions to prototype their applications easily. It also provides a working model for developers of an extended management infrastructure. This is especially useful for those providing additional services to the core JMX functionality.

Compatibility Test Suite

The *compatibility test suite* (CTS) for the JMX specification will check the conformance of JMX implementations; it is also mandated by the Java Community Process. The CTS verifies that applications claiming to conform to a specific part of JMX follow every point of the specification. The CTS for both the instrumentation and agent specifications has been developed by Sun Microsystems, Inc., again in its role as the JMX specification lead.

Since the classes defined by the JMX specification are optional packages of the Java platform, the CTS is implemented as a Technology Compatibility Kit (TCK) that is run by JavaTest™.

Each part of the JMX specification may identify mandatory and optional components. A JMX-compliant implementation must provide all mandatory services, and may provide any subset of the optional services, but those it does provide must conform to the specification.

When claiming JMX compliance, implementations list the optional services they support, and are tested by the CTS against their statement of conformance. This requires some modularity in the way the CTS can be run against various implementations which implement a number of subsets of the specification.

Architectural Overview

This section describes each part of the JMX specification and its relation to the overall management architecture:

- Instrumentation level
- Agent level

- Distributed services level
- Additional management protocol APIs

FIGURE 1-1 shows how the key components of JMX relate to one another within the three levels of the architectural model. These components are introduced in the following subsections and further discussed in the “Component Overview” on page 25.

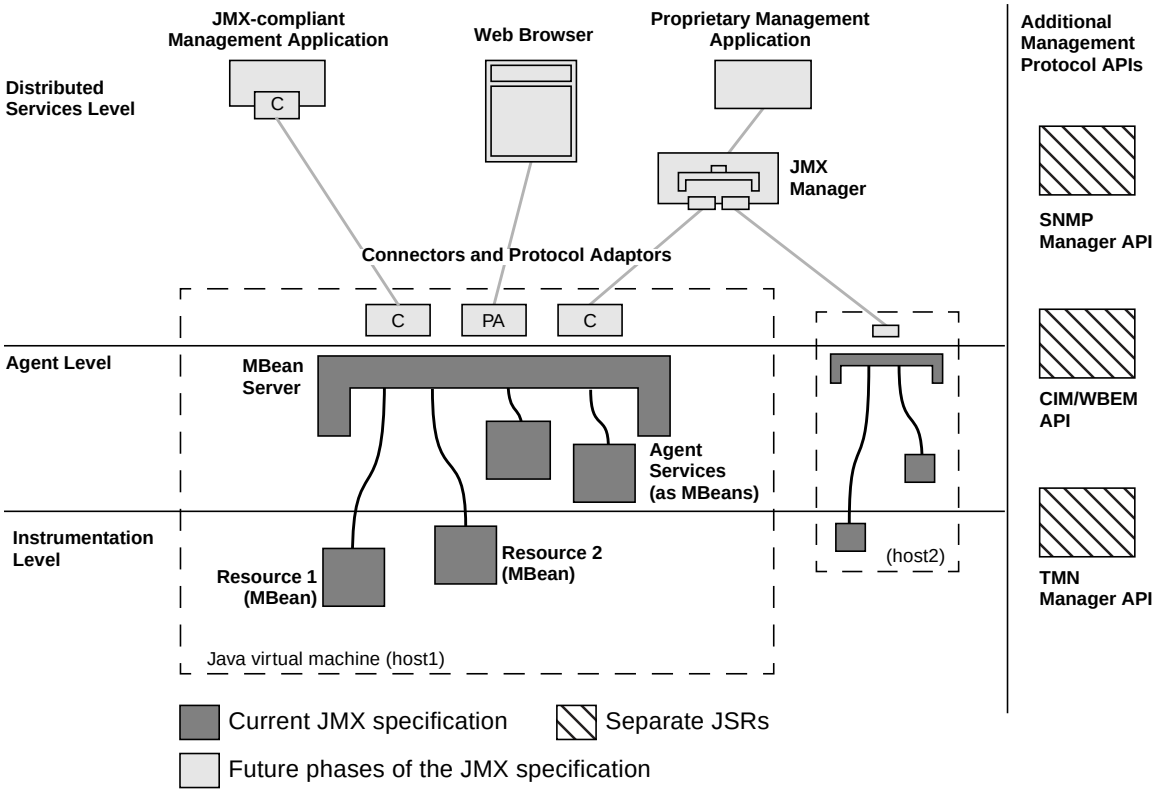


FIGURE 1-1 Relationship Between the Components of the JMX Architecture

Instrumentation Level

The instrumentation level provides a specification for implementing *JMX manageable resources*. A JMX manageable resource can be an application, an implementation of a service, a device, a user, and so forth. It is developed in Java, or at least offers a Java wrapper, and has been instrumented so that it can be managed by JMX-compliant applications.

The instrumentation of a given resource is provided by one or more Managed Beans, or MBeans, which are either standard or dynamic. Standard MBeans are Java objects that conform to certain design patterns derived from the JavaBeans™ component model. Dynamic MBeans conform to a specific interface which offers more flexibility at run-time. For further information, see “MBeans (Managed Beans)” on page 25.

The instrumentation of a resource allows it to be manageable through the agent level described in the next section. MBeans do not require knowledge of the JMX agent with which they operate.

MBeans are designed to be flexible, simple, and easy to implement. Developers of applications, services, or devices can make their products manageable in a standard way without having to understand or invest in complex management systems. Existing objects can easily be evolved to produce standard MBeans or wrapped as dynamic MBeans, thus making existing resources manageable with minimum effort.

In addition, the instrumentation level also specifies a notification mechanism. This allows MBeans to generate and propagate notification events to components of the other levels.

Since the instrumentation level consists of design patterns and Java interfaces, the reference implementation can only provide an example of the different MBeans and of their notification mechanism.

However, the compatibility test suite for the instrumentation level will check that MBeans being tested conform to the design patterns and implement the interfaces correctly.

Resources that provide instrumentation in conformance with the JMX specification (that is, they have been successfully tested against the CTS) are qualified as JMX manageable resources.

JMX manageable resources are compatible with the JDK™ 1.1.x programming environment, the EmbeddedJava™ environment, the PersonalJava™ environment, or the Java 2 Platform, Standard Edition, v 1.2.

JMX manageable resources are automatically manageable by JMX-compliant agents. They can also be managed by any non-JMX compliant system that supports the MBean design patterns and interfaces.

Agent Level

The agent level provides a specification for implementing agents. Management agents directly control the resources and make them available to remote management applications. Agents are usually located on the same machine as the resources they control, although this is not a requirement.

This level builds upon and makes use of the instrumentation level, in order to define a standardized agent to manage JMX manageable resources. The *JMX agent* consists of an MBean server and a set of services for handling MBeans. In addition, a JMX agent will need at least one communications adaptor or connector, but these are not specified in this phase. The MBean server implementation and the agent services are mandatory in an implementation of the specification.

The JMX agent can be embedded in the machine that hosts the JMX manageable resources when a Java Virtual Machine is available in that machine. Likewise, the JMX agent can be instantiated into a mediation/concentrator element when the managed resource only offers a proprietary (non-Java) environment. Otherwise, an JMX agent does not need to know which resources it will serve: any JMX manageable resource can use any JMX agent that offers the services it requires.

Managers access an agent's MBeans and use the provided services through a protocol adaptor or connector, as described in the next section. However, JMX agents do not require knowledge of the remote management applications that use them.

JMX agents are implemented by developers of management systems, who can build their products in a standard way without having to understand the semantics of the JMX manageable resources, or the functions of the management applications.

The reference implementation of the JMX agent is a set of Java classes which provide an MBean server and all of the agent services.

The agent compatibility test suite will check that agents being tested conform to the interfaces and functionality set forth in the agent specification. Agents that have been successfully tested against the agent CTS are qualified as JMX agents.

JMX agents run on the Java 2 Platform Standard Edition, and the objective is to be able to run JMX agents on smaller Java platforms, for example PersonalJava, and EmbeddedJava, once these are compatible with the Java 2 platform.

JMX agents will be automatically compatible with JMX distributed services, and can also be used by any non-JMX compliant systems or applications that support JMX agents.

Distributed Services Level

The detailed definition of the distributed services level is beyond the scope of this phase of the specification. A brief description is given here in order to complete the overview of the JMX architecture.

The distributed services level provides the interfaces for implementing JMX managers. This level defines management interfaces and components that can operate on agents or hierarchies of agents. These components can:

- Provide an interface for management applications to interact transparently with an agent and its JMX manageable resources through a connector
- Expose a management view of a JMX agent and its MBeans by mapping their semantic meaning into the constructs of a data-rich protocol (for example HTML or SNMP)
- Distribute management information from high-level management platforms to numerous JMX agents
- Consolidate management information coming from numerous JMX agents into logical views that are relevant to the end user's business operations
- Provide security

Management components cooperate with one another across the network to provide distributed, scalable management functions. Customized Java-based management functions can be developed on top of these components in order to deploy a management application.

The combination of the manager level with the other agent and instrumentation levels provides a complete architecture for designing and developing complete management solutions. The Java Management extensions technology brings unique facilities to such solutions: portability, on-demand deployment of management functionality, dynamic and mobility services, and security.

Additional Management Protocol APIs

The additional management protocol APIs provide a specification for interacting with existing management environments. These APIs are currently being developed as separate JSRs within the Java Community Process:

- SNMP manager API
- CIM/WBEM manager and provider API

These will be available on the JCP web site when ready: see "Related Information" on page xv. Additional interfaces to other important management environments, such as a TMN manager API, will be released in later phases.

Developers of management platforms and applications can use the APIs to interact with these management environments, possibly by encapsulating this interaction in a JMX manageable resource. For example, by developing an SNMP manager and instrumenting it as an MBean, portable Java management solutions can replace SNMP management applications on legacy systems.

These Java APIs thus help developers build platform-independent management applications for the most common industry standards. New management solutions can integrate the existing infrastructure and existing management can take advantage of a Java-based management application.

The additional management protocol APIs do not define the functions of the applications, or the architecture of the platforms, they only define standard Java APIs to access management technologies such as SNMP.

Component Overview

The key components of each architectural level are listed below and discussed in the subsequent sections.

- Instrumentation level
  - MBeans (standard, dynamic, open, and model MBeans)
  - Notification model
  - MBean metadata classes
- Agent level
  - MBean server
  - Agent services

Components of the Instrumentation Level

The key components of the instrumentation level are the Managed Bean (MBean) design patterns, the notification model, and the MBean metadata classes.

MBeans (Managed Beans)

An MBean is a Java object that implements a specific interface and conforms to certain design patterns. These requirements formalize the representation of the resource's *management interface* in the MBean. The management interface of a resource is the set of all necessary information and controls that a management application needs to operate on the resource.

The management interface of an MBean is represented as:

- Valued attributes which may be accessed
- Operations which may be invoked
- Notifications which may be emitted (see "Notification Model" on page 26)
- The constructors for the MBean's Java class

MBeans encapsulate attributes and operations through their public methods and follow the design patterns for exposing them to management applications. For example, a read-only attribute in a standard MBean will have just a *getter* method, whereas *getter* and a *setter* methods implement read-write access.

Any objects which are implemented as an MBeans and registered with the agent can be managed from outside the agent's Java virtual machine. Such objects include:

- The resources your application wishes to manage
- Value-added services provided to help manage resources
- Components of the JMX infrastructure that can be managed

Other JMX components, such as agent services, are specified as fully instrumented MBeans, which allows them to benefit from the JMX infrastructure and offer a management interface.

The JMX architecture does not impose any restrictions on where compiled MBean classes are stored. They can be stored at any location specified in the classpath of the agent's JVM, or at a remote site if class loading is used (see "Advanced Dynamic Loading" on page 127).

JMX defines four types of MBeans: standard, dynamic, open and model MBeans. Each of these corresponds to a different instrumentation need:

- **Standard MBeans** are the simplest to design and implement, their management interface is described by their method names.
- **Dynamic MBeans** must implement a specific interface, but they expose their management interface at run-time for greatest flexibility.
- **Open MBeans** are dynamic MBeans which rely on basic data types for universal manageability and which are self-describing for user-friendliness.
- **Model MBeans** are also dynamic MBeans that are fully configurable and self-described at run-time; they provide a generic MBean class with default behavior for dynamic instrumentation of resources.

Notification Model

The JMX specification defines a generic notification model based on the Java event model. Notifications can be emitted by MBean instances, as well as by the MBean server. This specification describes the notification objects and the broadcaster and listener interfaces that notification senders and receivers must implement.

A JMX implementation may provide services that allow distribution of this notification model, thus allowing a management application to listen to MBean and MBean server events remotely. **How the distribution of the notification model is achieved is outside the scope of this phase of the specification. Further phases of this specification will address advanced notification services, such as forwarding and storing until further retrieval by a management application.**

MBean Metadata Classes

The instrumentation specification defines the classes that are used to describe the management interface of an MBean. These classes are used to build a standard information structure for publishing the management interface of an MBean. One of the functions of the MBean server at the agent level is to provide the metadata of its MBeans.

The metadata classes contain the structures to describe all of the components of an MBean's management interface: its attributes, operations, notifications and constructors. For each of these, the metadata includes a name, a description and its particular characteristics. For example, one characteristic of an attribute is whether it is readable, writable or both; a characteristic of an operation is the signature of its parameter and return types.

The different types of MBeans extend the metadata classes in order to provide additional information. Through this inheritance, the standard information will always be available and management applications which know how to access the subclasses can obtain the extra information.

Components of the Agent Level

The key components in the agent level are the MBean server which is a registry for objects in the instrumentation level, and the agent services which enable a JMX agent to incorporate management intelligence for more autonomy and performance.

MBean Server

The Managed Bean server, or *MBean server* for short, is a registry for objects which are exposed to management operations in an agent. Any object registered with the MBean server becomes visible to management applications. However, the MBean server only exposes an MBean's management interface, never its direct object reference.

Any resource that you want to manage from outside the agent's Java virtual machine must be registered as an MBean in the server. The MBean server also provides a standardized interface for accessing MBeans within the same JVM, giving local objects all of the benefits of manipulating manageable resources. MBeans can be instantiated and registered by:

- Another MBean
- The agent itself
- A remote management application (through the distributed services)

When you register an MBean, you must assign it a unique *object name.* A management application uses the object name to identify the object on which it is to perform a management operation. The operations available on MBeans include:

- Discovering the management interface of MBeans
- Reading and writing their attribute values
- Performing operations defined by the MBeans
- Getting notifications emitted by MBeans
- Querying MBeans based on their object name or their attribute values

The MBean server relies on *protocol adaptors* and *connectors* to make the agent accessible from management applications outside the agent's JVM. Each adaptor provides a view through a specific protocol of all MBeans registered in the MBean server. For example, an HTML adaptor could display an MBean on a Web browser. **The view provided by protocol adaptors is necessarily different for each protocol and none are addressed in this phase of the JMX specification.**

Connectors provide a manager-side interface which handles the communication between manager and agent. Each connector will provide the same remote interface though a different protocol. When a remote management application uses this interface, it can connect to an agent transparently through the network, regardless of the protocol. **The specification of the remote management interface will be addressed in a future phase of the Java Management extensions.**

Adaptors and connectors make all MBean server operations to be available to a remote management application. For an agent to be managed, it must include at least one protocol adaptor or connector. However, an agent can include any number of these, allowing it to be managed by multiple managers, through different protocols.

Agent Services

Agent services are objects that can perform management operations on the MBeans registered in the MBean server. By including management intelligence into the agent, JMX helps you build more powerful management solutions. Agent services are often MBeans as well, allowing them and their functionality to be controlled through the MBean server. The JMX specification defines the following agent services:

- **Dynamic class loading** through the m-let (management applet) service retrieves and instantiates new classes and native libraries from an arbitrary network location.
- **Monitors** observe an MBean attribute's numerical or string value and can notify other objects of several types of changes in the target.
- **Timers** provide a scheduling mechanism based on a one-time alarm-clock notification or on a repeated, periodic notification.

- **The relation service** defines associations between MBeans and enforces the cardinality of the relation based on predefined relation types.

All of the agent services are mandatory in a JMX-compliant implementation.

Conformance

This section specifies which components of the instrumentation and agent levels are mandatory or optional regarding compliance of implementations to the JMX Instrumentation and Agent Specification, v1.0.

Instrumentation Level

Implementations compliant to the JMX Instrumentation Specification, v1.0, shall provide all the components specified in Chapter 2 “MBean Instrumentation” and in Chapter 4 “Model MBeans” of this specification. This includes all associated classes as defined by their corresponding Javadoc API. These components provide support for the instrumentation of standard and dynamic MBeans.

Provision of the components described in Chapter 3 “Open MBeans” of this specification is not required of a JMX-compliant implementation. The intent is that further releases of this specification will finalize the definition of open MBeans. The support for open MBean instrumentation will then be mandatory for implementations compliant to the new version of the specification.

Agent Level

Implementations compliant to the JMX Agent Specification, v1.0 shall provide all the components specified in Part II “JMX Agents” of this specification. This includes an implementation of the MBean server, the agent services, and all associated classes as defined by their corresponding Javadoc API. Therefore, the implementation of all four agent services that are specified is mandatory. The general intent is to keep the number of optional components, and therefore the number of possible configurations, to a minimum in this specification.

What Has Changed

This section lists all of the changes to the JMX specification since the previous version (*Public Release 2, December 1999*). This sections includes:

- Modifications to the specification itself (what is mandatory)
- Modifications of the corresponding API
- Modifications that would impact the understanding of the specification (including changes in terminology)

It does not include cosmetic changes to the present document.

Instrumentation Level:

- The API for the instrumentation level includes all Java *interfaces* defined for the agent level services, except those of the relation service. This is necessary for MBeans to call upon these services.
- The `MBeanInfo` class and its associated classes are now referred to as MBean *metadata* classes. The term *descriptor* specifically refers to the concept and classes of model MBean descriptors.
- The metadata for MBean attributes now indicates if a boolean attribute has an *is* getter: the API for the `MBeanAttributeInfo` includes the new `isIs` method.
- Open MBeans are not yet fully specified in this phase and none of the corresponding Javadoc API is provided. Some implementation details have been intentionally removed from this document, pending further developments. Therefore, compliance to open MBean component specification cannot be claimed.
- Model MBeans are now a mandatory component of the instrumentation specification.
- The XML grammar for model MBean descriptors is no longer specified, rather it is left undefined.

Agent Level:

- The classes used by the query mechanism of the MBean server have been modified. In the API, `QueryExp` and `ValueExp` are now Java interfaces and no longer abstract classes. The classes that inherited from these classes previously now implement the corresponding interface instead.
- The monitoring services were erroneously identified as optional in theirchapter introduction. All agent services are now mandatory in a compliant implementation, as stated in “JMX Compliant Agent” on page 103.
- The relation service is completely new. Like all of the other services, its interfaces and classes have all been grouped in a separate package of the API, namely `javax.management.relation`. The relation model and the service classes are fully specified in “Relation Service” on page 147.

PART I JMX Instrumentation Specification

CHAPTER 2

MBean Instrumentation

The instrumentation level of the JMX specification defines how to instrument resources in the Java programming language so that they can be managed. Resources which are developed according to the rules defined in this chapter are said to be JMX manageable resources.

The Java objects which implement resources and their instrumentation are called Managed Beans, or MBeans for short. MBeans must follow the design patterns and interfaces defined in this part of the specification. This insures that all MBeans provide the instrumentation of managed resources in a standardized way.

MBeans are manageable by any JMX agent, but they may also be managed by non-compliant agents which support the MBean concept.

This part of the specification is primarily targeted at developers of applications or devices wishing to provide management capabilities to their resources.

Developers of applications and devices are free to choose the granularity of objects that should be instrumented as MBeans. An MBean might represent the smallest object in an application, or it could represent the entire application. Application components designed with their management interface in mind may typically be written as MBeans. MBeans may also be used as wrappers for legacy code without a management interface or as proxies for code with a legacy management interface.

Definition

An MBean is a concrete Java class that includes the following instrumentation:

- A public constructor
- The implementation of its own corresponding MBean interface *or* an implementation of the `DynamicMBean` interface
- Optionally, an implementation of the `NotificationBroadcaster` interface

A class which implements its own MBean interface is referred to as a *standard* MBean. This is the simplest type of instrumentation available when developing new JMX manageable resources. An MBean which implements the `DynamicMBean` interface specified in this chapter is known as a *dynamic* MBean, since certain elements of its instrumentation can be controlled at runtime.

Which interface the MBean implements determines how it will be developed, not how it will be managed. JMX agents provide the abstraction for handling both types of instrumentation transparently. In fact, when both types of MBeans are being managed in a JMX agent, management applications handle them in a similar manner.

When developing a Java class from the standard MBean interface, it exposes the resource to be managed directly through its attributes and operations. Attributes are internal entities which are exposed through getter and setter methods. Operations are the other methods of the class that are available to managers. All of these methods are defined statically in the MBean interface and visible to an agent through introspection. This is the most straightforward way of instrumenting a new resource.

When developing a Java class from the `DynamicMBean` interface, attributes and operations are exposed indirectly through method calls. Instead of introspection, JMX agents must call one method to find the name and nature of attributes and operations. Then when accessing an attribute or operation, the agent calls a generic getter, setter or invocation method whose argument is the name of the attribute or operation. Dynamic MBeans enable you to rapidly instrument existing resources and other legacy code objects you wish to manage.

Concrete Classes

The first requirement of all MBeans, no matter what type they are, is that they cannot be abstract classes. Abstract classes cannot be instantiated and can therefore not be managed. Therefore, an MBean must be a concrete Java class. The methods of an MBean must all be implemented so that the MBean class can be instantiated and the instance can be managed.

MBean Public Constructor

In order to be a JMX manageable resource, the Java class of an MBean, whether standard or dynamic, must also have at least one public constructor. This allows the MBean to be instantiated by a JMX agent on demand from a management application.

An MBean may have any number of constructors, provided at least one is declared public, in order to allow an agent to do an instantiation. An MBean may also have any number of public constructors, all of which are available to a management application through the MBean's JMX agent.

Public constructors of an MBean may have any number and type of arguments. It is the developer's and administrator's responsibility to guarantee that the classes for all argument types are available to the agent and manager when instantiating an MBean.

An MBean may omit all constructors and rely on the *default constructor* which the Java compiler provides automatically in such a case. The default constructor is public and takes no arguments, which complies with the specification of an MBean. The Java compiler will not provide a default public constructor if any other constructor, public or protected, is defined.

CODE EXAMPLE 2-1 shows a simple MBean example with two constructors, one of which is the public constructor.

CODE EXAMPLE 2-1 Constructors of the `Simple` MBean Example

```
public class Simple {

    private Integer state = new Integer (0);

    // Default constructor only accessible from sub-classes
    //
    protected Simple() {
    }

    // Public constructor: this class is an MBean candidate
    //
    public Simple (Integer s) {
        state = s;
    }
    ...
}
```

Standard MBeans

In order to be manageable through a JMX agent, a standard MBean explicitly defines its management interface. The management interface defines the handles on the resource that are exposed for management. An MBean's interface is made up of the methods it makes available for reading and writing its attributes and for invoking its operations.

Standard MBeans rely on a set of naming rules, called *design patterns*, that should be observed when defining the interface of their Java object. These naming rules define the concepts of attributes and operations which are inspired by the JavaBeans™ component model. However, the actual design patterns for JMX take into consideration the inheritance scheme of the MBean, as well as lexical design patterns to identify the management interface. As a result, the design patterns for MBeans are specific to the JMX specification.

The management interface of a standard MBean is composed of:

- Its constructors: only the public constructors of the MBean class are exposed
- Its attributes: the properties which are exposed through getter and setter methods
- Its operations: the remaining methods exposed in the MBean interface
- Its notifications: the notification objects and types that the MBean is likely to emit

As described in "MBean Public Constructor" on page 34, constructors are an inherent component of an MBean. The attributes and operations are methods of an MBean, but they are identified by the MBean interface, as described below. The notifications of an MBean are defined through a different interface: see "JMX Notification Model" on page 47.

The process of inspecting the MBean interface and applying these design patterns is called *introspection*. The JMX agent uses introspection to look at the methods and superclasses of a class, determine if it represents an MBean that follows the design patterns, and recognize the names of both attributes and operations.

MBean Interface

The Java class of a standard MBean must implement a Java interface that is named after the class. This interface mentions the complete signatures of the attribute and operation methods that are exposed. Only the public methods contained in this interface are exposed for management. All methods of the MBean's Java class which are not listed in this interface are not accessible to a management application.

The name of an MBean's Java interface is formed by adding the `MBean` suffix to the MBean's Java class name. For example, the Java class `MyClass` would implement the `MyClassMBean` interface. The interface of a standard MBean is referred to as its *MBean interface*.

By definition, the Java class of an MBean must implement all of the methods in its MBean interface. How it implements these methods determines its response to management operations. An MBean may also define any other methods, public or protected that do not appear in its MBean interface.

The MBean interface may list methods defined in the MBean, as well as methods which the MBean inherits from its superclasses. This enables MBeans to extend and instrument classes whose Java source code is inaccessible.

A standard MBean may also inherit its management interface if one of its superclasses implements a Java interface named after itself (the superclass). For example, if `MySuperClass` is an MBean and `MyClass` extends `MySuperClass` then `MyClass` is also an MBean. If `MyClass` does not implement a `MyClassMBean` interface, then it will have the same management interface as `MySuperClass`. Otherwise, `MyClass` can re-define its management interface by implementing its own `MyClassMBean` interface.

Since interfaces may also extend parent interfaces, all public methods in the inheritance tree of the interface are also considered. For more information about how an MBean inherits its management interface, see "Inheritance Patterns" on page 44.

Having to define and implement an MBean interface is the main constraint put on a standard MBean in order to be a JMX manageable resource.

The `MyClass` Example MBean

CODE EXAMPLE 2-2 gives a basic illustration of the explicit definition of the management interface for an MBean named `MyClass`. Among the public methods it defines, `getHidden` and `setHidden` will not be part of the management interface because they do not appear in the `MyClassMBean` interface.

CODE EXAMPLE 2-2 `MyClassMBean` interface and `MyClass` Example

```
public interface MyClassMBean {
    public Integer getState();
    public void setState(Integer s);
    public void reset();
}
```

```
public class MyClass implements MyClassMBean {
    private Integer state = null;
    private String hidden = null;

    public Integer getState() {
        return(state);
    }
    public void setState(Integer s) {
        state = s;
    }
    public String getHidden() {
        return(hidden);
    }
    public void setHidden(String h) {
        hidden = h;
    }
    public void reset() {
        state = null;
        hidden = null;
    }
}
```

Lexical Design Patterns

The lexical patterns for attribute and operation names rely on the method names in an MBean interface. They enable a JMX agent to identify the names of attributes and operations exposed for management in a standard MBean. They also allow the agent to make the distinction between read-only, write-only and read-write attributes.

Attributes

Attributes are the fields or properties of the MBean which are in its management interface. Attributes are discrete, named characteristics of the MBean which define its appearance or its behavior, or are characteristics of the managed resource that the MBean instruments. For example, an attribute named `ipackets` in an MBean representing an Ethernet driver could be defined to represent the number of incoming packets.

Attributes are always accessed via method calls on the object that owns them. For readable attributes, there is a *getter* method to read the attribute value. For writable attributes, there is a *setter* method to allow the attribute value to be updated.

The following design pattern is used to identify attributes:

```
public AttributeType getAttributeName();
public void setAttributeName(AttributeType value);
```

If a class definition contains a matching pair of `get`*AttributeNam*e and `set`*AttributeName* methods that take and return the same type, these methods define a read-write attribute called *AttributeName*. If a class definition contains only one of these methods, the method defines either a read-only or write-only attribute.

The *AttributeType* may be of any Java class, or an array of any Java class, provided that this type is valid in the MBean's run-time context or environment.

When the type of an attribute is an array type, the getter and setter methods operate on the whole array. The design patterns do not include any getter or setter method for accessing individual array elements. Such access methods for indexed attributes are treated as MBean operations.

In addition, for boolean type attributes, it is possible to define a getter method using the following design pattern:

```
public boolean isAttributeName();
```

In order to reduce redundancy, only one of the two getter methods for boolean types is allowed. An attribute may have either an `is`*AttributeName* method or a `get`*AttributeName* method, but not both in the same MBean.

Operations

Operations are the actions that a JMX manageable resource makes available to management applications. These actions can be any computation which the resource wishes to expose, and they can also return a value.

In a standard MBean, an operation is a Java method specified in its interface and implemented in the class itself. Any method in the MBean interface which doesn't fit an attribute design pattern is considered to define an operation.

A typical usage is shown in CODE EXAMPLE 2-2 on page 37 where the MBean exposes the `reset` method to re-initialize its exposed attributes and private fields. Simple operations can also be written to access individual elements of an indexed array attribute.

Case Sensitivity

All attribute and operation names derived from these design patterns are case-sensitive. For example, this means that the methods `getstate` and `setState` define two attributes, one called `state` that is read-only, and one called `State` that is write-only.

While case sensitivity applies directly to component names of standard MBeans, it is also applicable to all component names of all types of MBeans, standard or dynamic. In general, all names of classes, attributes, operations, methods, and internal elements defined in the JMX specification are case sensitive, whether they appear as data or as functional code when they are manipulated by management operations.

Dynamic MBeans

Standard MBeans are ideally suited for straightforward management structures, where the structure of managed data is well defined in advance and unlikely to change often. In such cases, standard MBeans provide the quickest and easiest way to instrument manageable resources. When the data structures are likely to evolve often over time, the instrumentation must provide more flexibility, such as being determined dynamically at run-time. Dynamic MBeans bring this adaptability to the JMX specification and provide an alternative instrumentation with more elaborate management capabilities.

Dynamic MBeans are resources that are instrumented through a pre-defined interface which exposes the attributes and operations only at run-time. Instead of exposing them directly through method names, dynamic MBeans implement a method which returns all attributes and operation signatures.

Since the names of the attributes and operations are determined dynamically, these MBeans provide great flexibility when instrumenting existing resources. An MBean which implements the `DynamicMBean` interface provides a mapping for existing resources which do not follow standard MBean design patterns. Instead of introspection, JMX agents call the method of the MBean which returns the name of the attributes and operations it exposes.

MBeans which implement the `DynamicMBean` interface are also known as JMX manageable resources. When managed through a JMX agent, dynamic MBeans offer all of the same capabilities that are available through standard MBeans. Management applications which rely on JMX agents can manipulate all MBeans in exactly the same manner regardless of their type.

DynamicMBean Interface

In order for a resource object to be recognized as a dynamic MBean by the JMX agent, its Java class or one of its superclasses must implement the DynamicMBean public interface. However, an MBean is not allowed to implement both the DynamicMBean interface and its own MBean interface with standard design patterns. The JMX agent verifies that all MBean are either dynamic or standard but never both at the same time. Instrumentation developers must choose the management scheme which fits with the nature of their manageable resources.

The DynamicMBean interface is defined by the UML diagram in FIGURE 2-1 below. Each of the methods it defines is described in the following subsections.

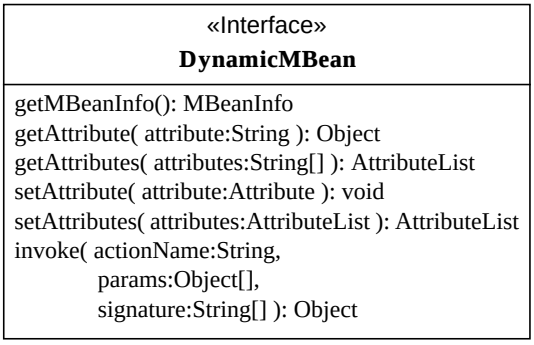


FIGURE 2-1 Definition of the DynamicMBean Interface

getMBeanInfo *Method*

This method returns an MBeanInfo object which contains the definition of the MBean's management interface. Conceptually, dynamic MBeans have both attributes and operations, only they are not exposed through method names. Instead, dynamic MBeans expose attribute names and types and operation signatures through the return value of this method at runtime.

This method returns an MBeanInfo object which contains a list of attribute names and their types, a list of operations and their parameters, and other management information. This type and its constituent classes are further described in "MBean Metadata Classes" on page 53.

`getAttribute` *and* `getAttributes` *Methods*

These methods take either an attribute name or a list of attribute names and return the value of the corresponding attribute(s). These are like a standard getter method, except the caller supplies the name of the attribute requested. It is up to the implementation of the dynamic MBean to properly map the exposed attributes names to their values through these methods.

The classes which describe attribute names, values and lists of names and values are described in "`Attribute` and `AttributeList` Classes" on page 109. These data types are also used by the `setAttribute` methods below.

`setAttribute` *and* `setAttributes` *Methods*

These methods take attribute name-value pairs and, like standard setter methods, they write these values to the corresponding attribute. When setting several attributes at a time, the list of attributes for which the write operation succeeded is returned. When setting only one attribute, there is no return value and any error is signaled by raising an exception. Again, it is up to the implementation of the dynamic MBean to properly map the new values to the internal representation of their intended attribute target.

`invoke` *Method*

The invoke method lets management applications call any of the operations exposed by the dynamic MBean. Here the caller must provide the name of the intended operation, the objects to be passed as parameters, and the types for these parameters. By including the operation signature, the dynamic MBean implementation may verify that the mapping is consistent between the requested operation and that which is exposed.

If the requested operation is successfully mapped to its internal implementation, this method returns the result of the operation. The calling application will expect to receive the return type exposed for this operation in the `MBeanInfo` method.

Behavior of Dynamic MBeans

When registered in a JMX agent, a dynamic MBean is treated in exactly the same way as a standard MBean. Typically, a management application will first obtain the management interface through the `getMBeanInfo` method, in order to have the names of the attributes and operations. The application will then make calls to getters, setters and the `invoke` method of the dynamic MBean.

In fact, the interface for dynamic MBeans is very similar to that of the MBean server in the JMX agent (see “Role of the MBean Server” on page 115). A dynamic MBean provides the management abstraction that the MBean server provides for standard MBeans. This is why management applications can manipulate both kinds of MBeans indifferently: the same management operations are applied to both.

In the case of the standard MBean, the MBean server uses introspection to find the management interface and then call the operations requested by the manager. In the case of the dynamic MBean, these tasks are taken over by the dynamic MBean’s implementation. In effect, the MBean server delegates the self-description functionality to the `getMBeanInfo` method of a dynamic MBean.

Coherence

With this delegation comes the responsibility of ensuring coherence between the dynamic MBean’s description and its implementation. The MBean server does not test or validate the self-description of a dynamic MBean in any way. Its developer must guarantee that the advertised management interface is accurately mapped to the internal implementation. For more information about describing an MBean, see “MBean Metadata Classes” on page 53.

From the manager’s perspective, how the dynamic MBean implements the mapping between the declared management interface and the returned attribute values and operation results is not important, it only expects the advertised management interface to be available. This gives much flexibility to the dynamic MBean to build more complex data structures, expose information which it can gather off-line, or provide a wrapper for resources not written in the Java programming language.

Dynamics

Since the management interface of a dynamic MBean is returned at runtime by the `getMBeanInfo` method, the management interface itself may be dynamic. That is, subsequent calls to this method may not describe the same management interface.

It is not the intention of the JMX instrumentation specification to define an MBean with a dynamic management interface. In the JMX architecture, a management application which retrieves the management interface of an MBean can expect this interface to be applicable throughout the life of the MBean. When instrumenting a resource, the set of attributes, operations and notifications available through its MBean instance should never change.

However, the MBean server in the JMX agent is not required to enforce a static management interface. That is, the MBean server is not responsible for regularly analyzing and comparing the management interface of its MBeans. Therefore, truly dynamic MBeans are possible, though they can only be managed by proprietary management applications designed specifically to handle them.

Inheritance Patterns

The *introspection* of an MBean is the process that JMX agents use to determine its management interface. This algorithm is applied at run-time by a JMX compliant agent, but it is described here since it determines how the inheritance scheme of an MBean influences its management interface.

When introspecting a standard MBean, the management interface is defined by the design patterns used in its MBean interface. Since interfaces may also extend parent interfaces, all public methods in the inheritance tree of the interface are also considered. When introspecting a dynamic MBean, the management interface is given through the `DynamicMBean` interface. In either case, the algorithm determines the names of the attributes and operations that are exposed for the given resource.

The introspection algorithm used is the following:

1. Verify that the MBean is not an abstract class and that it provides at least one public constructor. By nature, constructors are not inherited, meaning that even if a class inherits its management interface, it must explicitly provide a public constructor in order to be a valid JMX manageable resource.

2. If `MyClass` implements *both* a `MyClassMBean` interface and the `DynamicMBean` interface, then `MyClass` is *not* a JMX manageable resource.

3. If the `MyClass` MBean implements a `MyClassMBean` interface, then only the methods listed in, or inherited by, the interface are considered among all the methods of the MBean. Among the methods of the MBean, those that it inherits are also considered. The design patterns are then used to identify the attributes and operations from the method names in the `MyClassMBean` interface and its ancestors.

4. If `MyClass` implements the `DynamicMBean` interface, then the return value of its `getMBeanInfo` method will list the attributes and operations of the resource.

5. If the MBean implements neither `MyClassMBean` nor `DynamicMBean`, the inheritance tree of `MyClass` is examined, looking for the nearest superclass that implements either its own `MBean` interface or `DynamicMBean`.

   a. If there is an ancestor called `SuperClass` that implements `SuperClassMBean`, the design patterns are used to derive the attributes and operations from `SuperClassMBean`. In this case, the MBean `MyClass` then has the same management interface as the MBean `SuperClass`.

   b. If there is an ancestor called `SuperClass` that implements the `DynamicMBean` interface, then its `getMBeanInfo` method will list the attributes and operations. In this case, the MBean `MyClass` also has the same management interface as the MBean `SuperClass`.

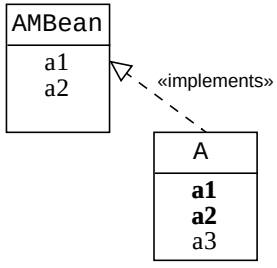
c. When there is no superclass implementing its own `MBean` interface or the `DynamicMBean` interface, `MyClass` is *not* a JMX manageable resource.

As a general rule, the management interface is defined by the MBean class or the nearest ancestor which implements *either* its own MBean interface *or* the `DynamicMBean` interface. If the class or the nearest ancestor with a management interface implements both, then there is an introspection error which is signalled to the management application. If the class and none of its superclasses implement neither interface, it is not a JMX manageable resource and the JMX agent will raise an MBean error (see "JMX Exceptions" on page 109).

These rules do not exclude the rare case of a class inheriting one type of management interface and overriding it by implementing the other type. For example, `MyClass` may have an ancestor which is a dynamic MBean and yet implement its own `MyClassMBean`, thereby becoming a standard MBean. Inversely, `MyClass` may implement the `DynamicMBean` interface even if one of its ancestors is a standard MBean. Whichever management interface is overridden will not be exposed for management.

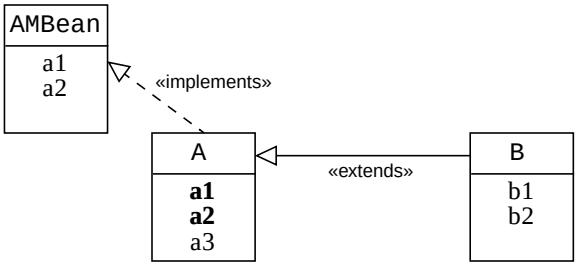
Standard MBean Inheritance

For standard MBeans, the management interface may be built up through inheritance of both the class and its interface. This is shown in the following examples, where the class fields a1, a2, ... stand for attributes or operations recognized by the design patterns for standard MBeans. Various combinations of these example cases are also possible.



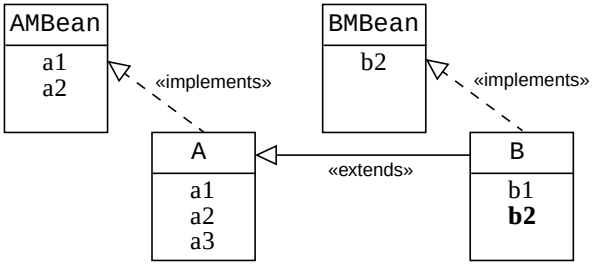
In the simplest case, class A implements class AMBean, which therefore defines the management interface for A: {a1, a2}.

FIGURE 2-2 Standard MBean Inheritance (Case 1)



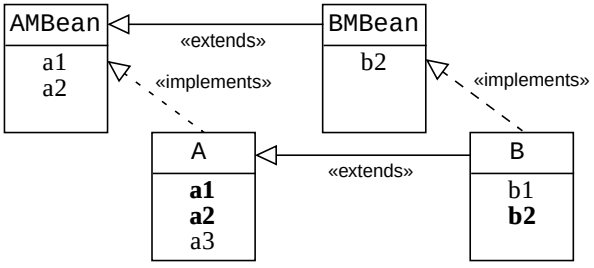
If class B extends A without defining its own MBean interface, then B is also an MBean, provided it defines a public constructor. B has the same management interface as A: {a1, a2}

FIGURE 2-3 Standard MBean Inheritance (Case 2)



If class B does implement the BMBean interface, then this defines the only management interface considered: {b2}.

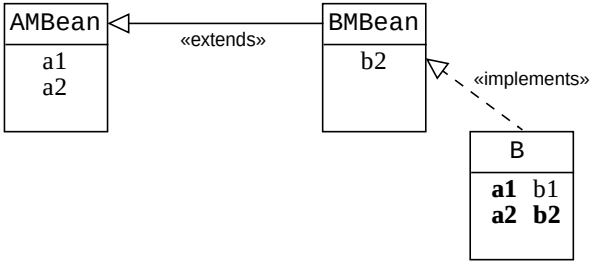
FIGURE 2-4 Standard MBean Inheritance (Case 3)



The BMBean interface and all interfaces it extends make up the management interface for the elements which B defines or inherits: {a1, a2, b2}.

Whether or not A implements AMBean makes no difference with regards to B.

FIGURE 2-5 Standard MBean Inheritance (Case 4)



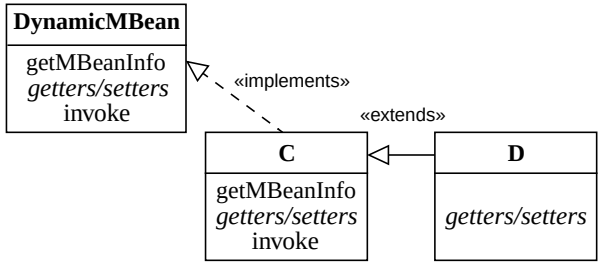
The class B must implement all methods defined in or inherited by the BMBean interface. If it does not inherit them, it must implement them explicitly: {a1, a2, b2}.

FIGURE 2-6 Standard MBean Inheritance (Case 5)

Dynamic MBean Inheritance

Like standard MBeans, dynamic MBeans can also inherit their instrumentation from a superclass. However, the management interface cannot be composed from the inheritance tree of the dynamic MBean class. Instead, the management interface is defined in its entirety by the `getMBeanInfo` method or the nearest superclass implementation of this method.

In the same way, subclasses may also redefine the getters, setters and `invoke` method, thus providing a different behavior for the same management interface. It is the MBean developer's responsibility that the subclass' implementation of the attributes or operations matches the management interface which is inherited or exposed.



If class `D` extends `C` without redefining the `getMBeanInfo` method, then `D` is a dynamic MBean with the same management interface. However, `D` overrides the getter and setter methods of `C`, thus providing a different implementation of the same attributes.

FIGURE 2-7 Dynamic MBean Inheritance

JMX Notification Model

The management interface of an MBean allows its agent to perform control and configuration operations on the managed resources. However, such interfaces provide only part of the functionality necessary to manage complex, distributed systems. Most often, management applications need to react to a state change or to a specific condition when it occurs in an underlying resource.

This section defines a model which allows MBeans to broadcast such management events, which are called *notifications*. Management applications and other objects register as *listeners* with the *broadcaster* MBean. The MBean notification model of JMX enables a listener to register only once and still receive all different events that may occur in the broadcaster.

This notification model only covers the transmission of events between MBeans within the same JMX agent. **How notifications are transmitted to remote management applications is not covered in this phase of the specification.**

The JMX notification model relies on the following components:

- A generic event type, `Notification`, which can signal any type of management event. The `Notification` event may be used directly, or may be sub-classed, depending on the information which needs to be conveyed with the event.
- The `NotificationListener` interface, which needs to be implemented by objects requesting to receive notifications sent by MBeans.
- The `NotificationFilter` interface, which needs to be implemented by objects which act as a *notification filter*. This interface lets notification listeners provide a filter to be applied to notifications emitted by an MBean.
- The `NotificationBroadcaster` interface, which needs to be implemented by each MBean wanting to emit *notifications*. This interface allows listeners to register their interest in the notifications emitted by an MBean.

By using a generic event type, this notification model allows any one listener to receive all types of events from a broadcaster. The filter is provided by the listener to specify only those events which are needed. Using a filter, a listener only needs to register once in order to receive all selected events of an MBean.

Any type of MBean, standard or dynamic, may be either a notification broadcaster, a notification listener, or both at the same time. Notification filters are usually implemented as callback methods of the listener itself, but this is not a requirement.

Notification Type

The *type* of a notification, not to be confused with its Java class, is the characterization of a generic notification object. The type is assigned by the broadcaster object and conveys the semantic meaning of a particular notification. The type is given as a `String` field of the `Notification` object. This string is interpreted as any number of dot-separated components, allowing an arbitrary, user-defined structure in the naming of notification types.

All notification types prefixed by "`jmx.`" are reserved for the notifications emitted by the components of the JMX infrastructure defined in this specification, such as `jmx.mbean.registered`. Otherwise, notification broadcasters are free to define all types they wish to use when naming the notifications they emit. Usually, MBeans will use type strings that reflect the nature of their notifications within the larger management structure in which they are involved.

For example, a vendor who provides JMX manageable resources as part of a management product might prefix all its notification types with *vendorName*. FIGURE 2-8 below shows a tree representation of the structure induced by the dot notation in notification type names.

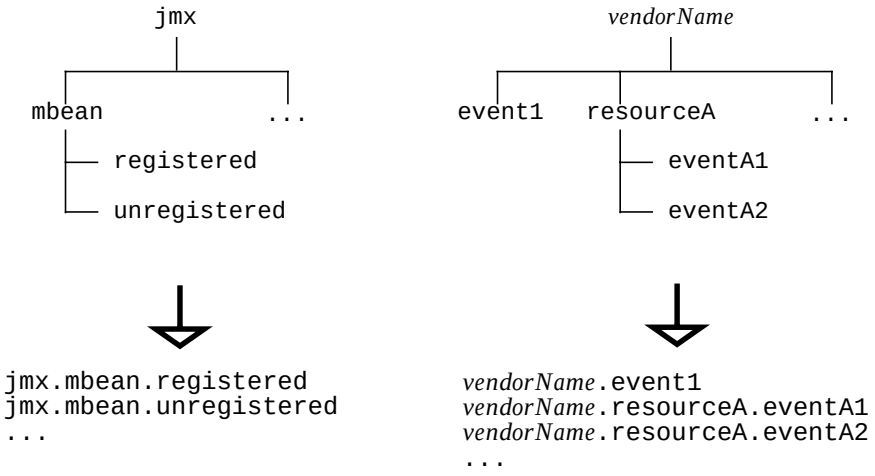


FIGURE 2-8 Structure of Notification Type Strings

`Notification` Class

The `Notification` class extends the `java.util.EventObject` base class and defines the minimal information contained in a notification. It contains the following fields:

- The *notification type*, which is a string expressed in a dot notation similar to Java properties, for example: *vendorName*`.resourceA.eventA1`
- A *sequence number*, which is a serial number identifying a particular instance of notification in the context of the notification broadcaster
- A *time stamp*, indicating when the notification was generated
- A *message* contained in a string, which could be the explanation of the notification for displaying to a user
- *User data* is used for whatever other data the notification broadcaster wishes to communicate to its listeners

Notification broadcasters should use the notification type to indicate the nature of the event to their listeners. Additional information that needs to be transmitted to listeners should be placed in the message or in the user data fields.

In most cases, this information is sufficient to allow broadcasters and listeners to exchange instances of the `Notification` class. However, subclasses of the `Notification` class may be defined when additional semantics are required within the notification object.

`NotificationBroadcaster` Interface

This interface specifies three methods which MBeans acting as notification broadcasters must implement:

- `getNotificationInfo` gives a potential listener the description of all notifications this broadcaster may emit. This method returns an array of `MBeanNotificationInfo` objects, each of which describes a notification. For more information about this class, see "`MBeanNotificationInfo` Class" on page 57.
- `addNotificationListener` registers a listener's interest in notifications sent by this MBean. This method takes a reference to a `NotificationListener` object, a reference to a `NotificationFilter` object, and a hand-back object.

  The hand-back object is provided by the listener upon registration and is opaque to the broadcaster MBean. The implementation of the broadcaster interface must store this object and return its reference to the listener with each notification. This hand-back object can allow the listener to retrieve context information for use while processing the notification.

  The same listener object may be registered more than once, each time with a different hand-back object. This means that the `handleNotification` method of this listener will be invoked several times, with different hand-back objects.

  The MBean has to maintain a table of listener, filter and hand-back triplets. When the MBean emits a notification, it invokes the `handleNotification` method of all the registered `NotificationListener` objects, with their respective hand-back object.

  If the listener has specified a `NotificationFilter` when registering as a `NotificationListener` object, the MBean will invoke the filter's `isNotificationEnabled` method first. Only if the filter returns an affirmative (`true`) response will the broadcaster then call the notification handler.
- `removeNotificationListener` unregisters the listener from a notification broadcaster. This method takes a reference to a `NotificationListener` object, as well as a hand-back object.

  If the hand-back object is provided, only the entry corresponding to this listener and hand-back pair will be removed. The same listener object may still be registered with other hand-back objects. Otherwise, if the hand-back is not provided, all entries corresponding to the listener will be removed.

Any type of MBean may implement the `NotificationBroadcaster` interface. This may lead to a special case of a standard MBean which has an empty management interface: its role as a manageable resource is to be a broadcaster of notifications. It must be a concrete class with a public constructor, and it must implement an MBean interface, which in this case defines no methods. The only methods in its class are those implementing the `NotificationBroadcaster` interface. This MBean may be registered in a JMX agent, and its management interface only contains the list of notifications that it may send.

`NotificationListener` Interface

This interface must be implemented by all objects interested in receiving notifications sent by any broadcaster. It defines a unique callback method, `handleNotification`, which will be invoked by a broadcaster MBean when it emits a notification.

Besides the `Notification` object, the listener's hand-back object is passed as an argument to the `handleNotification` method. This is a reference to the same object that the listener provided upon registration. It is stored by the broadcaster and returned unchanged with each notification.

Since all notifications are characterized by their type string, notification listeners only implement one handler method for receiving all notifications from all potential broadcasters. This method should then rely on the type string, other fields of the notification object and on the hand-back object to determine the broadcaster and the meaning of the notification.

`NotificationFilter` Interface

This interface is implemented by objects acting as a notification filter. It defines a unique method, `isNotificationEnabled`, which will be invoked by the broadcaster before it emits a notification. This method takes the Notification object that the broadcaster intends to emit and, based on its contents, returns `true` or `false`, indicating whether or not the listener should receive this notification.

The filter object is provided by the listener when it registers for notifications with the broadcaster, so each listener may provide its own filter. The broadcaster must apply each listener's filter, if defined, before calling the `handleNotification` method of the corresponding listener.

Listeners rely on the filter to screen all possible notifications and only handle the ones in which they are interested. An object may be both a listener and a filter by implementing both the `NotificationListener` and the `NotificationFilter` interfaces. In this case, the object reference will be given for both the listener and the filter object when registering it with a broadcaster.

Attribute Change Notifications

This section introduces a specific family of notifications, the *attribute change notifications*, which allows management services and applications to be notified whenever the value of a given MBean attribute is modified.

In the JMX architecture, the MBean has the full responsibility of sending notifications when an attribute change occurs. The mechanism for detecting changes in attributes and triggering the notification of the event is not part of the JMX specification. The attribute change notification behavior is therefore dependent upon the implementation of each MBean's class.

MBeans are not required to signal attribute changes, but if they wish to do so within the JMX architecture, they should rely on the following components:

- A specific event class, `AttributeChangeNotification`, which can signal any attribute change event.
- A specific filter support, `AttributeChangeNotificationFilter`, which allows attribute change notification listeners to filter the notifications depending on the attributes of interest.

Otherwise, attribute change notification broadcasters and listeners are defined by the same interfaces as in the standard notification model. Any MBean wishing to send attribute change notifications must implement the `NotificationBroadcaster` interface, as described in the "JMX Notification Model" on page 47. Similarly, the `NotificationListener` interface must be implemented by all objects interested in receiving attribute change notifications sent by an MBean.

`AttributeChangeNotification` Class

The `AttributeChangeNotification` class extends the `Notification` class and defines the following additional fields:

- The name of the attribute which has changed
- The type of the attribute which has changed
- The old value of the attribute
- The new value of the attribute

When implementing the attribute change notification model, broadcaster MBeans must use this class when sending notifications of attribute changes. They may also send other `Notification` objects for other events. The additional fields of this class provide the listener with information about the attribute which has changed. The notification type of all attribute change notifications must be `jmx.attribute.change`. This type is defined by the static string `ATTRIBUTE_CHANGE` declared in this class.

`AttributeChangeNotificationFilter` Class

The `AttributeChangeNotificationFilter` class implements the `NotificationFilter` interface and defines the following additional methods:

- `enableAttribute` - Enables notifications for the given attribute name.
- `disableAttribute` - Filters out notifications for the given attribute name.
- `disableAllAttributes` - Effectively disables all attribute change notifications.
- `getEnabledTypes` - Returns a list of all attribute names which are currently enabled for receiving notifications

Notification listeners wishing to observe certain attributes for changes may instantiate this class, configure the set of “enabled” attributes and use this object as the filter when registering as a listener with a known attribute change broadcaster. The attribute change filter allows the listener to receive attribute change notifications only for those attributes which are desired.

MBean Metadata Classes

This section defines the classes that describe an MBean. These classes are used both for the introspection of standard MBeans and for the self-description of all dynamic MBeans. These classes describe the management interface of an MBean in terms of its attributes, operations, constructors and notifications.

The JMX agent exposes all of its MBeans, regardless of their type, through the MBean metadata classes. All clients, whether management applications or other local MBeans wishing to view the management interface of an MBean, need to be able to interpret these objects and their constructs. Certain MBeans may provide additional data by extending these classes (see “Open MBean Metadata Classes” on page 63 and “Model MBean Metadata Classes” on page 73).

In addition to providing an internal representation of any MBean, these classes can be used to construct a visual representation of any MBean. One approach to management is to present all manageable resources to an operator through a graphical user interface. To this end, the complete description of all MBeans includes a descriptive text for each of their components. How this information is displayed is completely dependent upon the application which manages the MBean and is outside the scope of this specification.

The following classes define an MBean’s management interface; they are referred to collectively as the *MBean metadata classes* throughout this document:

- `MBeanInfo` - lists the attributes, operations, constructors and notifications
- `MBeanFeatureInfo` - superclass for the following classes
- `MBeanOperationInfo` - describes the method of an operation
- `MBeanConstructorInfo` - describes a constructor
- `MBeanParameterInfo` - describes a method parameter
- `MBeanAttributeInfo` - describes an attribute
- `MBeanNotificationInfo` - describes a notification

The following UML diagram shows the relationship between these classes as well as the components of each. Each class is fully described in the subsequent sections.

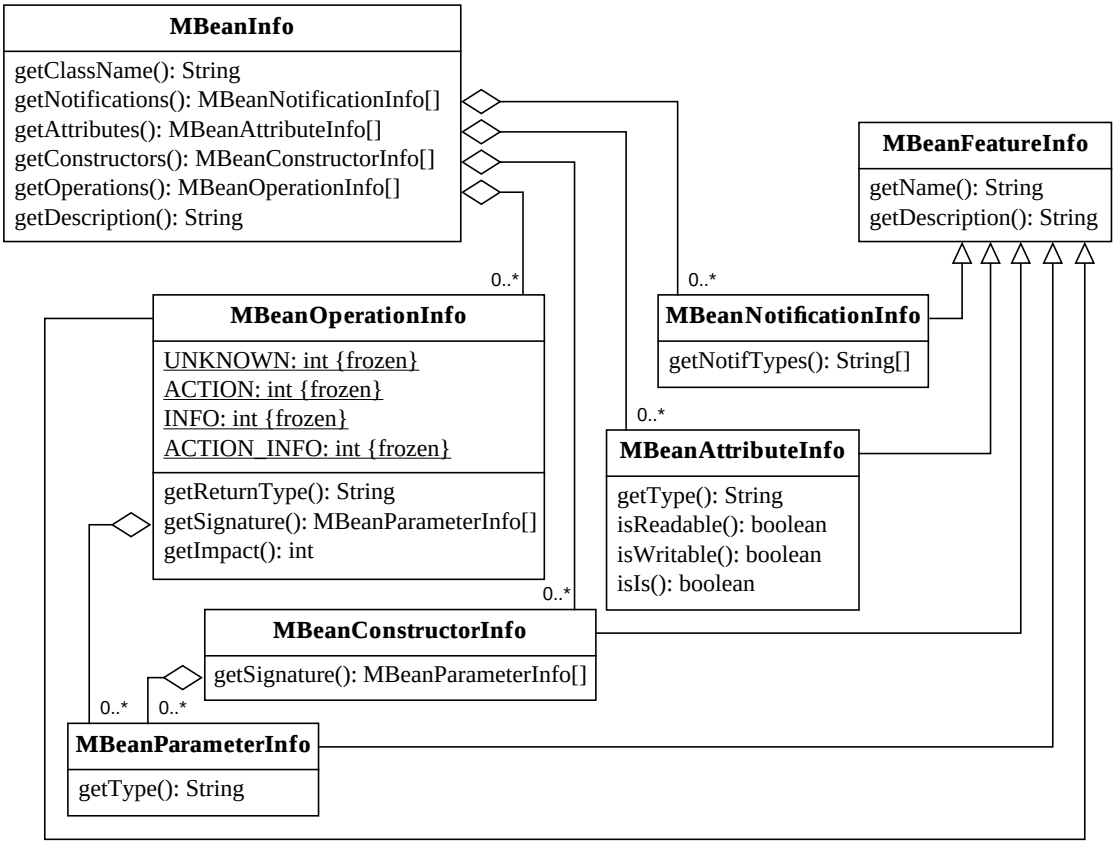


FIGURE 2-9 The MBean Metadata Classes

`MBeanInfo` Class

This class is used to fully describe an MBean: its attributes, operations, its constructors, and the notification types it may send. For each of these categories, this class stores an array of metadata objects for the individual components. If an MBean has no component in a certain category, for example no notifications, the corresponding method returns an empty array.

Each metadata object is a class which contains information which is specific to the type of component. For example, attributes are characterized by their type and read-write access, and operations by their signature and return type. All components have a case-sensitive name and a description string.

Besides the array of metadata objects for each component category, the `MBeanInfo` class has two descriptive methods. The `getClassName` method returns a string containing the Java class name of this MBean. The `getDescription` method should be used to return a string describing the MBean that is suitable for displaying to a user in a GUI. It should describe the MBean's overall purpose or functionality.

In the case of a standard MBean, the information contained in the `MBeanInfo` class is provided by the introspection mechanism of the JMX agent. Introspection can determine the components of the MBean, but it cannot provide a qualitative description. The introspection of standard MBeans provides a simple generic description string for the `MBeanInfo` object and all of its components. Therefore, all standard MBeans will have the same description. For dynamic MBeans, it is the developer's responsibility to ensure that the description strings for the `MBeanInfo` object and all of its components provide correct and useful information about the MBean.

`MBeanFeatureInfo` Class

This class is not directly returned by an `MBeanInfo` object

, but it is the parent of all of the other component metadata classes. All of the subsequent objects subclass `MBeanFeatureInfo` and inherit its two methods, `getName` and `getDescription`.

The `getName` method returns a string with the name of the component. This name is case-sensitive and identifies the given component within the MBean. For example, if an MBean interface exposes the `getstate` method, it will be described by an `MBeanAttributeInfo` object whose inherited `getName` method will return "`state`".

The `getDescription` method returns a string which provides a human readable explanation of a component. In the case of dynamic MBeans, this string must be provided by the developer. This string should be suitable for displaying to an operator through the user interface of a management application, for example.

`MBeanAttributeInfo` Class

The `MBeanAttributeInfo` class describes an attribute in the MBean's management interface. An attribute is characterized by its type and by how it is accessed.

The type of an attribute is the Java class which is used to represent it when calling its getter or setter methods. The `getType` method returns the name of this class as a string. The type string includes a terminating pair of brackets ("`[]`") when the attribute is an array type.

MBean access is either readable, writable or both. Read access implies that a manager may get the value of this attribute, and write access that it may set its value:

- The `isReadable` method will return `true` if this attribute has a getter method in its MBean interface or if the `getAttribute` method of the `DynamicMBean` interface will succeed with this attribute's name as the parameter; otherwise it will return `false`.
- The `isWritable` method will return `true` if this attribute has a setter method in its MBean interface or if the `setAttribute` method of the `DynamicMBean` interface will succeed with this attribute's name as a parameter; otherwise it will return `false`.
- The `isIs` method will return `true` if this attribute has a boolean type and a getter method with the *is* prefix (versus the *get* prefix); otherwise it will return `false`. Note that this information is only relevant for a standard MBean.

See "Lexical Design Patterns" on page 38 for the definition of getter and setter methods in standard MBeans.

Note – By this definition, the access information does not take into account any read or write access to an attribute's internal representation which an MBean developer might provide through one of the operations.

`MBeanConstructorInfo` Class

MBean constructors are described solely by their signature: the order and types of their parameter. This class describes a constructor and contains one method, `getSignature`, which returns an array of `MBeanParameterInfo` objects. This array has no elements if the given constructor has no parameters. Elements of the parameter array are listed in the same order as constructor parameters, and each element gives the type of its corresponding parameter (see "`MBeanParameterInfo` Class" on page 57).

`MBeanOperationInfo` Class

The `MBeanOperationInfo` class describes an individual operation of an MBean. An operation is defined by its signature, return type, and its impact.

The `getImpact` method returns an integer that can be mapped using the static fields of this class. Its purpose is to communicate the impact this operation will have on the managed entity represented by the MBean. A method described as `INFO` will not modify the MBean, it is a read-only method which only returns data. An `ACTION` method has some effect on the MBean, usually a write operation or some other state modification. The `ACTION_INFO` method has both read and write roles.

The `UNKNOWN` value is reserved for the description of all operations of a standard MBean, as introspected by the MBean server.

Impact information is very useful for making decisions on which operations to expose to users at different times. It can also be used by some security schemes. It is the dynamic MBean developer's responsibility to correctly and consistently assign the impact of each method in its metadata object. Indeed, the difference between "information" and "action" is dependent on the design and usage of each MBean.

The `getReturnType` method returns a string containing the class name of the Java object returned by the operation being described. The return type may be an array of objects, in which case the string name includes a terminating pair of brackets ("`[]`").

The `getSignature` method returns an array of `MBeanParameterInfo` objects where each element describes a parameter of the operation. The array elements are listed in the same order as the operation's parameters, and each element gives the type of its corresponding parameter (see below).

`MBeanParameterInfo` Class

The `MBeanParameterInfo` class is used to describe a parameter of an operation or of a constructor. This class gives the class type of the parameter and also extends the `MBeanFeatureInfo` class in order to provide a name and description.

The `getType` method returns a string which identifies the Java class of the object being described. The parameter may be an array of objects, in which case the type name includes a terminating pair of brackets ("`[]`").

`MBeanNotificationInfo` Class

The `MBeanNotificationInfo` class is used to describe the notifications that are sent by an MBean. This class extends the `MBeanFeatureInfo` class in order to provide a name and a description. The name should give the fully qualified class name of the notification objects that are actually broadcast.

The `getNotifTypes` method returns an array of strings containing the notification types that the MBean may emit. The notification type is a string containing any number of elements in dot notation, not the name of the Java class which

implements this notification. As described in “JMX Notification Model” on page 47, a single notification class may be used to send several notification types. All of these types are returned in the string array returned by this method.

CHAPTER 3

Open MBeans

This chapter defines a way of instrumenting resources that MBeans should conform to if they wish to be “open” to the widest range of management applications. These MBeans are called open MBeans.

This chapter is incomplete and open MBeans are not fully specified in the JMX instrumentation specification, v1.0. As a consequence none of the Javadoc API for open MBean components is provided with the specification, even for those classes presented in this chapter. Concepts and classes of open MBeans are presented in this chapter are still subject to change.

Hence, compliance to open MBeans is not required, and in fact, compliance to open MBeans is impossible to claim in this phase of the specification. Open MBeans are intended to be fully defined and to become mandatory in the next release.

Overview

The goal of open MBeans is to provide a mechanism that will allow management applications and their human administrators to understand and use new managed objects as they are discovered at runtime. These MBeans are called “open” because they rely on small, predefined set of universal Java types and they advertise their functionality.

Management applications and open MBeans are thus able to share and use management data and operations at runtime without requiring the recompilation, reassembly or expensive dynamic linking of management applications. In the same way, human operators are able to intelligently use the newly discovered managed object without having to consult additional documentation. Thus, open MBeans contribute to the flexibility and scalability of management systems.

In order to provide its own description to management applications, an open MBean must be a dynamic MBean (see “Dynamic MBeans” on page 40). Beyond the `DynamicMBean` interface, there is no corresponding “open” interface that must be implemented. Instead, an MBean earns its “openness” by providing a descriptively rich metadata and by using only certain predefined data types in its management interface.

An open MBean has attributes, operations, constructors and possibly notifications like any other MBeans. It is a dynamic MBean with the same behavior and all of the same functionality. It also has the responsibility of providing its own description. However, all of the object types that the MBean manipulates, its attribute types, its operation parameters and return types, and its constructor parameters, must belong to the set defined in “Basic Data Types” below. It is the developer’s responsibility to fully implement the open MBean using these data types exclusively.

An MBean indicates whether it is open or not through the `MBeanInfo` object it returns. Open MBeans return an `OpenMBeanInfo` object which is a subclass of `MBeanInfo`. Other component metadata classes are also subclassed and it is the developer’s responsibility to fully describe the open MBean using the proper classes. If an MBean is marked as open in this manner, it is a guarantee that a JMX-compliant management application can immediately make use of all attributes and operations without requiring additional classes.

Since open MBeans are also dynamic MBeans and provide their own description, the MBean server does not check the accuracy of the `OpenMBeanInfo` object (see “Behavior of Dynamic MBeans” on page 42). The developer of an open MBean must guarantee that the management interface relies on the basic data types and provides a rich, human-readable description. As a rule, the description provided by the various parts of an open MBean must be suitable for displaying to a user through a Graphical User Interface (GUI).

Basic Data Types

In order for management applications to immediately make use of MBeans without recompilation, re-assembly, or dynamic linking, *all* MBean attributes, method return values, and method arguments must be limited to a universal set of data types. This set is called the *basic data types* for open MBeans. This set is defined as: the wrapper objects that correspond to the Java primitive types (such as Integer, Long, Boolean, etc.), String, CompositeData, TabularData, and arrays of these data types.

The following list specifies all data types that are allowed in open MBeans:

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.String`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `javax.management.openmbean.CompositeData`
- `javax.management.openmbean.TabularData`
- `java.lang.Boolean[]`
- `java.lang.Byte[]`
- `java.lang.Character[]`
- `java.lang.String[]`
- `java.lang.Short[]`
- `java.lang.Integer[]`
- `java.lang.Long[]`
- `java.lang.Float[]`
- `java.lang.Double[]`
- `javax.management.openmbean.CompositeData[]`
- `javax.management.openmbean.TabularData[]`

All of the wrapper classes for the primitive types are defined and implemented in all Java virtual machines. The `CompositeData` and `TabularData` classes are defined in the `javax.management.openmbean` package. They are used to form aggregates of the basic data types and provide a mechanism for expressing complex data objects in a consistent manner.

Since `CompositeData` and `TabularData` are also basic data types, these structures can contain other composite or tabular structures and have arbitrary complexity. The `TabularData` class can be used to represent tables, a very common structure in the management domain. The basic data types can therefore be used alone or in combination to satisfy most data representation requirements.

Representing Complex Data

This section presents the two non-primitive classes that are included in the set of basic data types: `CompositeData` and `TabularData`. These two classes are not fully specified at this time, but their main characteristics are presented here.

These classes represent complex data types within open MBeans. Both of these classes are used to create aggregate structures which are built up from the primitive data types and these classes themselves. This means that any JMX agent or any JMX-compliant management application may manipulate any open MBean and use the arbitrarily complex structures it contains.

The two classes provide some semantic structure to build aggregates from the basic data types. An instance of the `CompositeData` class is equivalent to a hash table: values are retrieved by giving the name of the desired data item. An instance of `TabularData` contains an array of `CompositeData` instances which can be

retrieved individually by giving a unique key. However, instances of both classes are immutable, as are the composite structures they define. For example, once a `CompositeData` object is instantiated, you cannot add an item to it and you cannot change the value of an existing item. You must instantiate another `CompositeData` object with the desired structure and values.

`CompositeData` Class

The `CompositeData` class defines an immutable hash table with an arbitrary number of entries, called data items, which can be of any type. In order to comply with the design patterns for open MBeans, all data items must have a type among the set of basic data types. Since this set also includes `CompositeData`, complex hierarchies may be represented by creating composite types which contain other composite types.

A `CompositeData` instance associates string keys with the values of each data item. Since the instances are immutable, the class constructor takes arguments to fully describe the contents of all items. The methods of the class then search for and return data items based on their string key. The enumeration of all data items is also possible.

`TabularData` Class

The `TabularData` class defines a table structure with an arbitrary number of rows which can be indexed by any number of columns. Each row is a `CompositeData` instance, but all rows must have the same composite data description. The columns of the table are headed by the names of the data items which make up the uniform `CompositeData` rows. The index consists of a subset of the data items in the common composite data structure, with the requirement that this subset must be a key which uniquely identifies each row of the table.

A `TabularData` instance defines an immutable table: its structure and contents are given to the constructor for instantiation, and thereafter, no methods are provided to modify the structure, add a row or modify its contents. The constructor verifies that all rows are uniform and that they can be uniquely indexed by a given subset of the data items.

The methods of the `TabularData` class take an array of objects representing a key value which indexes one row and returns the `CompositeData` instance which makes up the designated row. All rows of the table can also be retrieved in an enumeration.

Open MBean Metadata Classes

To distinguish open MBeans from other MBeans, JMX provides a set of metadata classes which are used specifically to describe open MBeans. These classes are subclasses of the `MBeanInfo` class and its components; the `MBeanInfo` classes are fully described in “MBean Metadata Classes” on page 53. The present section discusses only those components which are particular to open MBeans.

The following classes of the `javax.management.openmbean` package define the management interface of an open MBean:

- `OpenMBeanInfo` - lists the attributes, operations, constructors and notifications
- `OpenMBeanOperationInfo` - describes the method of an operation
- `OpenMBeanConstructorInfo` - describes a constructor
- `OpenMBeanParameterInfo` - describes a method parameter
- `OpenMBeanAttributeInfo` - describes an attribute

All of the above classes directly extend the MBean metadata class whose name is given by removing the `Open` prefix. Each of these classes describes a category of components in an open MBean. However, open MBeans do not have a specific metadata object for notifications: they use the `MBeanNotificationInfo` class described on page 57.

In order to describe the aggregate data types specific to the open MBean model, the `javax.management.openmbean` package also defines the following classes. These extend the parameter and attribute metadata classes, respectively, and must be used when a parameter or attribute is an aggregate data type:

- `CompositeParameterInfo`
- `CompositeAttributeInfo`
- `TabularParameterInfo`
- `TabularAttributeInfo`

Through methods inherited from their superclasses, the open MBean metadata objects describe the management interface of an open MBean. Beyond this description, they override certain methods to provide the extra information required of open MBeans and to return the metadata of the new aggregate data types.

Since open MBeans are meant to be a universal way of exchanging management functionality, their description must be rich enough for an operator to understand and use their functionality. All of the open MBean metadata classes inherit the `getDescription` method which must return a non-empty string. Each component of an open MBean must use this method to provide a description of itself, for example, the side-effects of an operation or the significance of an attribute. All descriptions should be suitable for displaying to a user in a GUI.

The extra information that the open MBean model requires the developer to provide is a list of legal values and one default value for all attributes and all operation parameters. This information allows any user to intelligently manipulate a new or unfamiliar open MBean.

Finally, the metadata classes for composite and tabular types provide the structure for describing these aggregate types which are specific to open MBeans. These structures are recursive, that is can contain instances of themselves, since aggregate types can be built from other aggregate objects. All of the other basic data types are adequately described by the information structure for simple types inherited from the basic MBean metadata classes.

`OpenMBeanInfo` Class

The `OpenMBeanInfo` class provides the main information structure for describing an open MBean. It directly extends the `MBeanInfo` class and thus inherits the methods for specifying the class name and overall MBean description. It also inherits the method for returning an array of notification metadata objects, as notifications are described in the same way as for dynamic MBeans.

However, this class overrides all other methods which describe each category of MBean component: attributes, operations and constructors. Their new implementation still describes all components of a given category, but they now rely on the open MBean metadata classes. Since each of the open MBean metadata objects subclasses the original metadata object, each method returns an array of the subclass type to describe an open MBean. The open MBean metadata classes for each category of component are described in the subsequent sections.

`OpenMBeanOperationInfo` and `OpenMBeanConstructorInfo` Classes

The `OpenMBeanOperationInfo` and `OpenMBeanConstructorInfo` classes extend the `MBeanOperationInfo` and `MBeanConstructorInfo` classes, respectively (see their definition on page 56). The former describes an operation of an open MBean, and the latter describes one of its constructors.

Both of these classes override the `getSignature` method of their respective superclass, again only to describe their parameters with open MBean metadata objects. The `getSignature` method nominally returns an array of `MBeanParameterInfo` objects, but both implementations actually return instances of the `OpenMBeanParameterInfo` class described in the next section.

Only the `OpenMBeanOperationInfo` class inherits the `getImpact()` method, and in the case of open MBean, it cannot return `UNKNOWN`. This means that all operations must be identified as `ACTION`, `INFO`, or `ACTION_INFO` when instantiating their metadata objects. It is the open MBean developer's responsibility to correctly assign the impact of each operation. The "impact" provides information to the user about an operation's side effects, as a complement to its self-description.

`OpenMBeanParameterInfo` and `OpenMBeanAttributeInfo` Classes

The `OpenMBeanParameterInfo` and `OpenMBeanAttributeInfo` classes extend the `MBeanParameterInfo` and `MBeanAttributeInfo` classes, respectively (see their definition on page 57 and page 55). The former describes one parameter of an operation or constructor, and the latter describes an attribute of an open MBean.

Both of these classes inherit the `getType` method from their superclass, and the attribute metadata information inherits `isReadable`, `isWritable`, and `isIs` for defining attribute access. None of these methods are overridden and therefore have the same functionality as in the superclass.

Both classes define the `getDefaultValue` and `getLegalValues` methods to provide additional information about the parameter or attribute. These methods are implemented identically in both classes and have exactly the same functionality in each object.

The `getDefaultValue` method is used to indicate a default value for a given parameter or attribute. At run-time, it returns an `Object` which must be assignment compatible with the type named by the `getType` method of the same parameter or attribute description object. The default value can be used to initialize an attribute or to provide a parameter value when the operation's caller has no particular preference for some parameter.

The `getLegalValues` method is used to return a list of permissible values for a given parameter or attribute. It returns an `Object` array, the elements of which must assignment compatible with the type named by the `getType` method of the same parameter or attribute description object. The legal values can be used to provide the user with a list of choices when editing writable attributes or filling in operation parameters. For readable attributes, this method provides a list of legal values that may be expected. If a set of legal values is supplied, then the MBean server will verify that any value written to the attribute or used for this parameter is a member of this set.

Since these classes are specific to open MBeans, all parameter and attribute types are necessarily one of the basic data types. However, these classes have specific subclasses for describing aggregate types: `CompositeParameterInfo` and

CompositeAttributeInfo must be used to describe a composite data object, and TabularParameterInfo and TabularAttributeInfo must be used to describe a tabular data object. These classes are described in the next two sections.

CompositeParameterInfo and CompositeAttributeInfo Classes

The CompositeParameterInfo and CompositeAttributeInfo classes extend the OpenMBeanParameterInfo and OpenMBeanAttributeInfo classes, respectively (see the previous section). They describe an instance of the CompositeData class when used either as a parameter or as an attribute type.

Both classes define a method for obtaining a description of all data items in the given composite data: getParameterInfo and getAttributeInfo, respectively. These return an array of metadata objects, one element for each data item in the composite data object. Each data item of a composite parameter is described by an OpenMBeanParameterInfo instance, and each of a composite attribute by an OpenMBeanAttributeInfo instance. In this way, every data item in a composite data type will inherit the methods for describing its own default and legal values.

Composite data objects may also have data items which are other aggregate data objects (see “CompositeData Class” on page 62). In this case, the corresponding element of the returned array will be an instance of either CompositeAttributeInfo, CompositeParameterInfo, TabularParameterInfo, or TabularAttributeInfo, depending on the case. Such a structure will recursively define complex data types until all data items belong to one of the basic data types.

TabularParameterInfo and TabularAttributeInfo Classes

The TabularParameterInfo and TabularAttributeInfo classes extend the CompositeParameterInfo and CompositeAttributeInfo classes, respectively (see the previous section). They describe an instance of the TabularData class when used either as a parameter or as an attribute type.

These classes inherit the getParameterInfo and getAttributeInfo methods, respectively, which return an array describing a composite data. In the superclass, this array described the composite data itself. In the tabular data metadata objects, this array describes the table structure, one element for each column of the table. Each element describes a data item and the array defines the composite data to which all table rows must conform.

Both classes then define the `getIndexNames` method which returns an array of strings. Each element is the name of a column, and the group of columns make up the index for the table. All index names must be found among the names given by the metadata information of the table columns. The index must be a key which can uniquely identify each row of the table.

Open MBean Requirements Summary

To summarize, an open MBean must possess the following properties:

- It must fully implement the `DynamicMBean` interface.
- All attributes, method arguments, and non-void return values must be objects in the set of basic data types for open MBeans.
- The implementation of the getMBeanInfo method must return an `OpenMBeanInfo` instance which fully describes the MBean components using the open MBean metadata objects.
- All objects of type `CompositeData` that are described in the `OpenMBeanInfo` object must be fully described using the `CompositeParameterInfo` or `CompositeAttributeInfo` class. Using only its superclass is insufficient.
- All objects of type `TabularData` that are described in the `OpenMBeanInfo` must be fully described using the `TabularParameterInfo` or `TabularAttributeInfo` class. Using any of their superclasses is insufficient.
- All of the following methods must return valid, meaningful data (non-empty strings) suitable for display to users:
  - `OpenMbeanInfo.getDescription`
  - `OpenMBeanOperationInfo.getDescription`
  - `OpenMBeanConstructorInfo.getDescription`
  - `OpenMBeanParameterInfo.getDescription`
  - `OpenMBeanAttributeInfo.getDescription`
  - `CompositeParameterInfo.getDescription`
  - `CompositeAttributeInfo.getDescription`
  - `TabularParameterInfo.getDescription`
  - `TabularAttributeInfo.getDescription`
  - `MBeanNotificationInfo.getDescription`
- Instances of `OpenMBeanOperationInfo.getImpact` must return one of the constant values `ACTION`, `INFO`, or `ACTION_INFO`. The value `UNKNOWN` may not be used.

Note – As with other dynamic MBeans, the MBean server does not verify the proper usage of the open MBean metadata classes. It is up to the MBean developer to insure that all composite data and tabular data metadata provide coherent default values, legal values and indexes.
The developer must also insure that all MBean components are adequately described in a meaningful way for the intended users. This qualitative requirement cannot be programmatically enforced.

CHAPTER 4

Model MBeans

A *model MBean* is a generic, configurable MBean which anyone can use to rapidly instrument almost any resource. Model MBeans are dynamic MBeans which also implement the interfaces specified in this chapter. These interfaces define structures that, when implemented, provide an instantiable MBean with default and configurable behavior.

Further, the Java Management extensions specify that a model MBean implementation must be supplied as part of all conforming JMX agents. This means that resources, services and applications can rely on the presence of a generic, template for creating manageable objects on-the-fly. Users only need to instantiate a model MBean, configure the exposure of the default behavior, and register it in a JMX agent. This significantly reduces the programming burden for gaining manageability. Developers can instrument their resources with JMX in as little as 3 to 5 lines of code.

Instrumentation with model MBeans is universal because instrumentors are guaranteed that there will be a model MBean appropriately adapted to all environments which implement the Java Management extensions.

Overview

The model MBean specification is set of interfaces that provides a management template for managed resources. It is also a set of concrete classes provided in conjunction with the JMX agent. The JMX agent must provide an implementation class named `javax.management.modelmbean.RequiredModelMBean`. This model MBean implementation is intended to provide ease of use and extensive default management behavior for the instrumentation.

The MBean server is a repository and a factory for the model MBean, so the managed resource obtains its model MBean object from the JMX agent. The managed resource developer does not have to supply his own implementation of this class. Instead, the resource is programmed to create and configure its model MBean at run-time, dynamically instrumenting the management interface it needs to expose.

Resources to be managed add custom attributes, operations, and notifications to the basic model MBean object by interfacing with the JMX agent and model MBeans that represent the resource. There may be one or more instances of a model MBean for each instance of a resource (application, device, and so forth) to be managed in the system. The model MBean is a dynamic MBean, meaning that it implements the `DynamicMBean` interface. As such, the JMX agent will delegate all management operations to the model MBean instances.

The model MBean instances are created and maintained by the JMX agent, like other MBean instances. The managed resource instantiating the model MBean does not have to be aware of the specifics of the implementation of the model MBean. Implementation differences between environments include the JVM, persistence, transactional behavior, caching, scalability, throughput, location transparency, remoteability, and so on. The `RequiredModelMBean` implementation will always be available, but there may be other implementations of the model MBean available, depending upon the needs of the environment in which the JMX agent is installed.

For example, a JMX agent running in a J2ME™ (Java 2 Platform, Micro Edition) environment may provide a `RequiredModelMBean` with no persistence or remoteability. A JMX agent running in an application server's JVM supporting J2EE™ (Java 2 Platform, Enterprise Edition) technologies may provide a `RequiredModelMBean` that handles persistence, transactions, remote access, location transparency, and security. In either case, the instrumentation programmer's task is the same. The MBean developer does not have to provide different versions of its MBeans for different Java environments, nor does he have to program to a specific Java environment.

The model MBean, in cooperation with its JMX agent, will be implemented to support its own persistence, transactionalism, location transparency, and locatability, as applicable in its environment. The instrumentation developer does not need to develop an MBean with its own transactional and persistence characteristics. He merely instantiates his model MBean in the JMX agent and trusts the implementation of the model MBean that the JMX agent has is appropriate for the environment in which the JMX agent currently exists.

Any implementation of the model MBean must implement the `ModelMBean` interface which extends the `DynamicMBean`, `PersistentMBean` and `ModelMBeanNotificationBroadcaster` interfaces. The model MBean must expose its metadata in a `ModelMBeanInfoSupport` object which extends `MBeanInfo` and implements the `ModelMBeanInfo` interface. A model MBean instances sends attribute change notifications and generic notifications for which

both the managed resource and management applications may listen. The model MBean has both a default constructor and a constructor which takes a `ModelMBeanInfo` instance.

The model MBean information includes a *descriptor* for each attribute, constructor, operation, and notification in its management interface. A descriptor is an essential component of the model MBean. It contains dynamic, extensible, configurable behavior information for each MBean component. This includes, but is not limited to, logging policy, notification responses, persistence policy, value caching policy. Most importantly, the descriptors of a model MBean provide the mapping between the attributes and operations in the management interface and the actual methods that need to be called to satisfy the get, set, or invoke request.

Allowing methods to be associated with the attribute allows for dynamic, runtime delegation. For example, a `getAttribute("myApplStatus")` call may actually invoke the `myAppl.StatusChecker` method on another object that is part of the managed resource. The object `myAppl` may be in this JVM, or it may be in another JVM on this host or another host, depending on how the model MBean has been configured through its descriptors. In this way, distributed, component oriented applications are supported.

The `ModelMBean` interface extends the `DynamicMBean` interface. The implementation of the `DynamicMBean` methods should use the policy in the descriptors to guide how the requests are satisfied. How this should be done is described in greater detail in “`DynamicMBean` Implementation” on page 85.

The `ModelMBean` interface also extends the `PersistentMBean` interface specific to model MBeans. The `load` and `store` methods of this interface are responsible for analyzing and complying with the persistence policy in the descriptors. The persistence policy should be specifiable at both the MBean level and at the attribute level. These methods should be called when appropriate by the model MBean implementation itself and not necessarily by the managed resource or a management application. The implementation may choose to not support any actual, direct persistence, in which case these methods will do nothing.

Generic Notifications

The `ModelMBean` interface extends the `ModelMBeanNotificationBroadcaster` interface. This interface defines a `sendNotification` method which sends any `Notification` object to all registered listeners. It also overloads the `sendNotification` method to accepts a text message and wraps it in a notification named `Generic` of type `jmx.modelmbean.general`. This makes it easier for managed resources to signal important events as well as informational events. Finally, this interface also provides methods for sending the attribute change notifications for which the model MBean’s implementation is responsible.

Interaction with Managed Resources

When a managed resource is instrumented through a model MBean, it uses the `ModelMBeanInfo` interface to expose its intended management interface. At initialization, the managed resource obtains access to the JMX agent through the static `findMBeanServer` method of the `MBeanServerFactory` class (see “MBean Server Factory” on page 115). The managed resource then will create or find and reference one or more instances of the model MBean using the `instantiate`, `create`, `getObjectInstance`, or `queryMBeans` methods. The predefined attributes that are part of the model MBean’s name are meant to establish a unique managed resource (MBean) identity.

The managed resource then configures the model MBean object with its management interface. This includes the custom attributes, operations, and notifications that it wants management applications to access through the JMX agent. The resource specific information can thus be dynamically determined at execution time. The managed resource sets and updates any type of data as an attribute in the model MBean at will with a single `setAttribute` method invocation. The attribute is now published for use by any management system.

The model MBean has an internal caching mechanism for storing attribute values that are provided by the management resource. Maintaining values of fairly static attributes in the model MBean allows it to return that value without calling the managed resource. The resource may also set its model MBean to disable caching, meaning that the resource will be called whenever an attribute is accessed. In this case, the managed resource is invoked and it returns the attribute values and to the model MBean. In turn, the model MBean returns these values to the MBean server which returns them to the request originator, usually a management application. Since the model MBean can be persistent and is locatable, critical but transient managed resources can retain any required counters or state information within the JMX agent. Likewise, if persistence is supported, the managed resource’s data survives recycling of the JMX agent.

The model MBean implements the `NotificationBroadcaster` interface. One `sendNotification` API call on the model MBean by the managed resource sends notifications to all “interested” management systems. Predefined or unique notifications can be sent for any managed resource or management system defined significant event. These notifications must be documented in the `ModelMBeanNotificationInfo` object. Notifications are typically sent by a managed resource when operator intervention is required or the application’s state is unacceptable. Notifications can also be sent based on MBean life cycle, attribute changes, or for informative reasons. The model MBean sends attribute change notifications whenever a custom attribute is set through the model MBean. The managed resource can capture change requests initiated by the management system by listening for the attribute change notification as well. The managed resource can then choose to implement the attribute change from the model MBean into the resource.

Interaction with Management Applications

Management applications access model MBeans in the same way that they access dynamic or standard MBeans. However, if the manager understands model MBeans it will be able to get additional information out of the descriptors that are part of the model MBean. This additional metadata makes it easier for an arbitrary management consoles to understand and treat managed resources that are instrumented as model MBeans. As with any MBean, the management application will “find” the JMX agent and model MBean objects through the methods of the MBean server.

The manager may then interact with the model MBean through the JMX agent. It will find the available attributes and operations through the `MBeanInfo` provided by the managed resource. For model MBeans, the manager will find out behavior details about supported attributes, operations, and notifications through the `ModelMBeanInfo` and `Descriptor` interfaces. Like any other MBean, attributes are accessed through the getter and setter methods of the MBean server, and operations through its invoke method. Since the model MBean is a notification broadcaster, management notification may be added as listeners for any notifications or attribute change notifications from the managed resource.

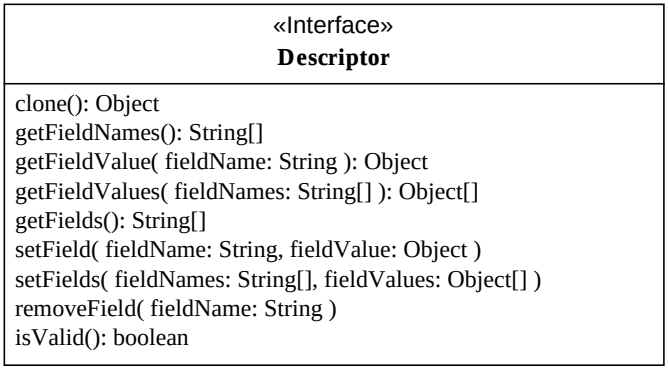
Model MBean Metadata Classes

The management interface of a model MBean is described by its `ModelMBeanInfo` instance. The `getMBeanInfo` method of a model MBean (specified by the `DynamicMBean` interface) must return an extension of `MBeanInfo` which also supports the `ModelMBeanInfo` interface. The `ModelMBeanInfo` interface adds MBean *descriptor* and *managedResource* components to the `MBeanInfo`. `ModelMBeanInfo` returns arrays of `ModelMBeanAttributeInfo`, `ModelMBeanOperationInfo`, `ModelMBeanConstructorInfo`, and `ModelMBeanNotificationInfo` instances. These classes extend the MBean metadata classes whose name is given by removing the `Model` prefix.

The model MBean extensions of the MBean metadata classes implement the `DescriptorAccess` interface. This essentially adds a `Descriptor` for each attribute, constructor, operation, and notification in its management interface. The descriptor is accessed through the metadata object for each component.

`Descriptor` Interface

A descriptor defines behavioral and run-time metadata that is specific to model MBeans. The descriptor data is kept as a set of fields, each consisting of a name-value pair. The `Descriptor` interface must be implemented by the class representing a descriptor. The `DescriptorAccess` interface defines how to get and set the `Descriptor` from within the model MBean metadata classes. The `Descriptor` interface describes how to interact with a descriptor instance returned by the `DescriptorAccess` interface. See “Predefined Descriptor Fields” on page 94 for a discussion of the valid field names and values that must be supported.



`Descriptor` Interface Implementation

The `Descriptor` interface implementation must have the following constructors and methods:

Descriptor()

Default constructor which returns an empty descriptor.

Descriptor(with Descriptor)

Copy constructor for the `Descriptor` class.

Descriptor(with field names and values)

Constructor that verifies that the field names include a descriptor type. It verifies that the predefined fields contain valid values.

Descriptor(with field list)

Constructor that verifies that the field list includes a name and descriptor type. It verifies that the predefined fields contain valid values.

getFieldNames

Returns all the field names of the descriptor in a `String` array.

getFieldValue(s)

Finds the given field name(s) in a descriptor and returns its (their) value.

setField(s)

Finds the given field name(s) in a descriptor and sets it (them) to the provided value.

getFields

Returns the descriptor information as an array of strings, each with the `fieldName=fieldValue` format. If the field value is null then the field is defined as `fieldName=`.

removeFields

Removes a descriptor field from the descriptor.

clone

Returns a new `Descriptor` instance which is a duplicate of the descriptor.

isValid

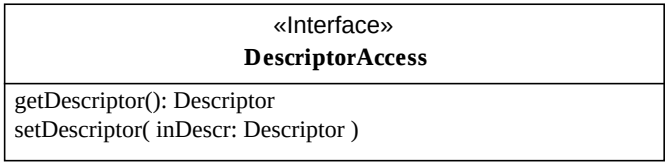
Returns `true` if this descriptor is valid for its `descriptorType` field.

toString

Returns a human readable string containing the descriptor information.

DescriptorAccess Interface

This interface must be implemented by the ModelMBeanAttributeInfo, ModelMBeanConstructorInfo, ModelMBeanOperationInfo, and ModelMBeanNotification classes.



getDescriptor

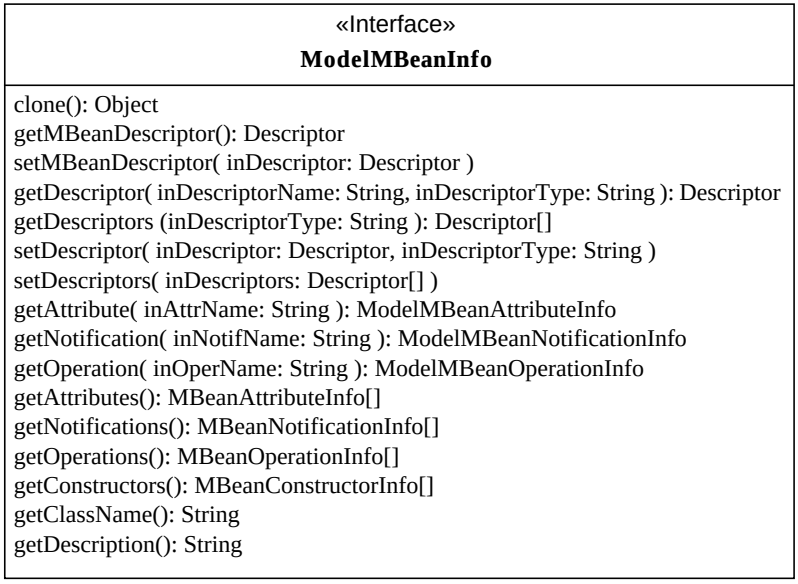
This method returns a copy of the descriptor associated with the metadata class.

setDescriptor

This method replaces the descriptor associated with the metadata class with a copy of the one passed in. This is a full replacement, not a merge.

ModelMBeanInfo Interface

The ModelMBeanInfo interface is defined to allow the association of a descriptor with the model MBean, attribute, constructor, operation, and notification metadata classes. This descriptor is used to define behavioral characteristics of the model MBean instance. The descriptor is accessed through the DescriptorAccess interface. When the getMBeanInfo method of the DynamicMBean interface is invoked on a model MBean, it must return an instance of a class which implements the ModelMBeanInfo interface.



`ModelMBeanInfo` Implementation

The requirements of the `ModelMBeanInfo` implementation are the following:

- It should extend the `MBeanInfo` class.
- It must implement the `ModelMBeanInfo` interface.
- Its `getAttributes`, `getConstructors`, `getOperations`, and `getNotifications` methods must return `ModelMBeanAttributeInfo`, `ModelMBeanConstructorInfo`, `ModelMBeanOperationInfo`, and `ModelMBeanNotificationInfo` arrays, respectively.
- The `ModelMBeanAttributeInfo`, `ModelMBeanConstructorInfo`, `ModelMBeanOperationInfo`, and `ModelMBeanNotificationInfo` classes it returns must extend their respective `MBeanAttributeInfo`, `MBeanConstructorInfo`, `MBeanOperationInfo`, and `MBeanNotificationInfo` classes.
- The `ModelMBeanAttributeInfo`, `ModelMBeanConstructorInfo`, `ModelMBeanOperationInfo`, and `ModelMBeanNotificationInfo` classes it returns must implement the `DescriptorAccess` interface. This interface associates a configurable `Descriptor` object with the metadata class. The descriptor allows the definition of behavioral policies for the MBean component.
- It must implement the following constructors:

ModelMBeanInfo

The default constructor which constructs a `ModelMBeanInfo` with empty component arrays and a default MBean descriptor.

ModelMBeanInfo (with ModelMBeanInfo)

Constructs a `ModelMBeanInfo` which is a duplicate of the one passed in.

ModelMBeanInfo (with className, description, ModelMBeanAttributeInfo[], ModelMBeanConstructorInfo[], ModelMBeanOperationInfo[], ModelMBeanNotificationInfo[])

Creates a `ModelMBeanInfo` with the provided information, but the MBean descriptor is a default. The MBean descriptor must not be null. The default descriptor should at least contain the `name` and `descriptorType` fields. The name should match the MBean name.

ModelMBeanInfo (with className, description, ModelMBeanAttributeInfo[], ModelMBeanConstructorInfo[], ModelMBeanOperationInfo[], ModelMBeanNotificationInfo[], MBeanDescriptor)

Creates a `ModelMBeanInfo` with the provided information. The MBean descriptor is verified: if it is not valid, an exception will be thrown and a default MBean descriptor will be set.

- It must implement the following model MBean-specific methods:

getMBeanDescriptor

Returns the MBean descriptor. This descriptor contains default configuration and policies that apply to the whole MBean and to its components by default. The `descriptorType` field will be "`MBean`".

setMBeanDescriptor

Sets the MBean descriptor. This descriptor contains MBean-wide default configuration and policies. This is a full replacement, no merging of fields is done. The descriptor is verified before it is set: if it is not valid, the change will not occur.

getDescriptor(s)

Returns a descriptor from a model MBean metadata object by name and descriptor type (as found in the `descriptorType` field on the descriptor).

setDescriptor(s)

Sets a descriptor in the model MBean in a model MBean metadata object by name and descriptor type (found in the `descriptorType` field on the descriptor). Replaces the descriptor in its entirety.

getAttribute

Returns a `ModelMBeanAttributeInfo` by name.

getOperation

Returns a `ModelMBeanOperationInfo` by name.

getNotification

Returns a `ModelMBeanNotificationInfo` by name.

- It must implement the following methods specified in the `ModelMBeanInfo` interface but identical to those of the `MBeanInfo` class (see "`MBeanInfo` Class" on page 54):

getAttributes

Returns an array of all `ModelMBeanAttributeInfo` objects.

getNotifications

Returns an array of all `ModelMBeanNotificationInfo` objects.

getOperations

Returns an array of all `ModeMBeanOperationInfo` objects.

getConstructors

Returns an array of all `ModelMBeanConstructorInfo` objects.

getClassName

Returns the name of the managed resource class.

getDescription

Returns the description of this model MBean instance.

`ModelMBeanAttributeInfo` Implementation

The `ModelMBeanAttributeInfo` must extend the `MBeanAttributeInfo` class and implement the `DescriptorAccess` interface. The `DescriptorAccess` interface associates a `Descriptor` instance with the existing metadata of the `MBeanAttributeInfo` class.

This descriptor must have a `name` field which matches the name given by the `getName` method of the corresponding metadata object. It must have a `descriptorType` with the value "`attribute`". It may also contain the following defined fields: `value`, `default`, `displayName`, `getMethod`, `setMethod`, `protocolMap`, `persistPolicy`, `persistPeriod`, `currencyTimeLimit`, `lastUpdatedTimeStamp`, `iterable`, `visibility`, and `presentationString`. See "Attribute Descriptor Fields" on page 95 for a detailed description of each of these fields.

The `ModelMBeanAttributeInfo` class must have the following constructors:

- A constructor accepting a name, description, getter `Method`, and setter `Method` which sets the descriptor to a default value with at least the `name` and `descriptorType` fields set.
- A constructor accepting a name, description, getter `Method`, setter `Method`, and a `Descriptor` instance which has at least its `name` and `descriptorType` fields set.
- A constructor accepting a name, type, description, `isReadable`, `isWritable`, and `isIs` boolean parameters which sets the descriptor to a default value with at least the `name` and `descriptorType` fields set.
- A constructor accepting a name, description, `isReadable`, `isWritable`, and `isIs` boolean parameters, and a `Descriptor` instance which has at least its `name` and `descriptorType` fields set.
- A copy constructor accepting a `ModelMBeanAttributeInfo` object.

ModelMBeanConstructorInfo Implementation

The ModelMBeanConstructorInfo must extend the MBeanConstructorInfo class and implement the DescriptorAccess interface. The DescriptorAccess interface associates a Descriptor instance with the existing metadata of the MBeanConstructorInfo class.

This descriptor must have a name field which matches the name given by the getName method of the corresponding metadata object. It must have a descriptorType with the value "operation" and a role of "constructor". It may also contain the following defined fields: displayName, class, visibility, and presentationString. See "Operation Descriptor Fields" on page 97 for a detailed description of each of these fields.

The ModelMBeanConstructorInfo class must have the following constructors:

- A constructor accepting a description and Constructor object which sets the descriptor to a default value with at least name and descriptorType fields set.
- A constructor accepting a description, a Constructor object, and a Descriptor instance which has at least the name and descriptorType fields set.
- A constructor accepting a name, a description, and an MBeanParameterInfo array which sets the descriptor to a default value with at least the name and descriptorType fields set.
- A constructor accepting a name, description, MBeanParameterInfo array, and a Descriptor instance which has at least its name and descriptorType fields set.
- A copy constructor accepting a ModelMBeanConstructorInfo object.

ModelMBeanOperationInfo Implementation

The ModelMBeanOperationInfo must extend the MBeanOperationInfo class and implement the DescriptorAccess interface. The DescriptorAccess interface associates a Descriptor instance with the existing metadata of the MBeanOperationInfo class.

This descriptor must have a name field which matches the name given by the getName method of the corresponding metadata object. It must have a descriptorType with the value "operation" and a role of "operation", "getter", or "setter". It may also contain the following defined fields: displayName, class, targetObject, targetType, lastReturnedValue, currencyTimeLimit, lastReturnedTimeStamp, visibility, and presentationString. See "Operation Descriptor Fields" on page 97 for a detailed description of each of these fields.

The ModelMBeanOperationInfo class must have the following constructors:

- A constructor accepting a description and a `Method` object which sets the descriptor to a default value with at least its `name` and `descriptorType` fields set.
- A constructor accepting a description, a `Method` object, and a `Descriptor` instance which at has least its `name` and `descriptorType` fields set.
- A constructor accepting a name, description, `MBeanParameterInfo` array, type, and an impact which sets the descriptor to a default value with at least the `name` and `descriptorType` fields set.
- A constructor accepting a name, description, `MBeanParameterInfo` array, type, impact and a `Descriptor` instance which has at least its `name` and `descriptorType` fields set.
- A copy constructor accepting a `ModelMBeanOperationInfo` object.

`ModelMBeanNotificationInfo` Implementation

The `ModelMBeanNotificationInfo` must extend the `MBeanNotificationInfo` class and implement the `DescriptorAccess` interface. The `DescriptorAccess` interface associates a `Descriptor` instance with the existing metadata of the `MBeanNotificationInfo` class.

This descriptor must have a `name` field which matches the name given by the `getName` method of the corresponding metadata object. It must have a `descriptorType` with the value "`notification`". It may also contain the following defined fields: `displayName`, `severity`, `messageID`, `log`, `logfile`, `visibility`, and `presentationString`. See "Notification Descriptor Fields" on page 98 for a detailed description of each of these fields.

The `ModelMBeanNotificationInfo` class must have the following constructors:

- A constructor accepting an array of notification types, a name and a description which sets the descriptor to a default value with at least its `name` and `descriptorType` fields set.
- A constructor accepting an array of notification types, a name, a description, and a `Descriptor` instance which has at least its `name` and `descriptorType` fields set.
- A copy constructor accepting a `ModelMBeanNotificationInfo`.

Model MBean Specification

All JMX agents must have an implementation class of a model MBean which is called `javax.management.modelmbean.RequiredModelMBean`. The `RequiredModelMBean` and any other compliant model MBean must comply with the following requirements:

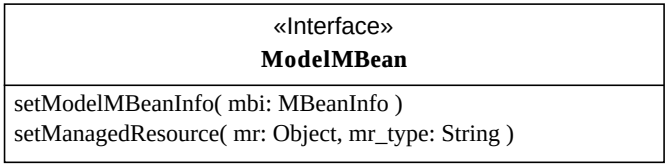
- Implement the `ModelMBean` interface which extends the following interfaces:
  - `DynamicMBean`
  - `PersistentMBean`
  - `ModelMBeanNotificationBroadcaster`
- Return an object from the `getMBeanInfo` method of the `DynamicMBean` interface which:
  - Implements the `ModelMBeanInfo` interface
  - Extends `MBeanInfo`
  - Returns `ModelMBeanAttributeInfo` objects from the `getAttributes` method
  - Returns `ModelMBeanConstructorInfo` objects from the `getConstructors` method
  - Returns `ModelMBeanOperationInfo` objects from the `getOperations` method
  - Returns `ModelMBeanNotificationInfo` objects from the `getNotifications` method
- Have the following constructors:
  - A default constructor having an empty parameter list
  - A constructor accepting a `ModelMBeanInfo`

`ModelMBean` Interface

Java technology-based resources wishing to be manageable instantiate the `RequiredModelMBean` or another compliant model MBean using the MBean server's `createMBean` method. The resource then sets the `ModelMBeanInfo` (including its descriptors) for the `ModelMBean` instance. The attributes and operations exposed via the `ModelMBeanInfo` for the model MBean are accessible to other MBeans, and to management applications. Through the `ModelMBeanInfo` descriptors, values and methods in the managed application can be defined and

mapped to attributes and operations of the model MBean. This mapping can be defined during development in a file, or dynamically and programmatically at runtime.

The `ModelMBean` interface extends `DynamicMBean`, `PersistentMBean`, and `ModelMBeanNotificationBroadcaster` and its unique methods are defined by the following UML diagram.



`ModelMBean` Implementation

The following sections describe how the `ModelMBean` interface is implemented by the `RequiredModelMBean` class and should be implemented by compliant model MBeans. This combines both the meaning of the methods and the implementation details.

setModelMBeanInfo (with `ModelMBeanInfo`*)*

Creates the model MBean to reflect the given `ModelMBeanInfo` interface. Sets the `ModelMBeanInfo` object for the model MBean to the provided `ModelMBeanInfo` object. Initializes a `ModelMBean` instance using `ModelMBeanInfo` passed in.

The model MBean must be instantiated, but not yet registered with the MBean server. Only after the model MBean's `ModelMBeanInfo` and its `Descriptor` objects are customized, should the model MBean be registered with the MBean server.

setManagedResource (with ManagedResourceObject, Type)

Sets the managed resource attribute of the model MBean to the supplied object. Sets the instance of the object against which to execute all operations in this model MBean management interface (metadata and descriptors). The `String` field encodes the target object type of reference for the managed resource. This can be: `ObjectReference`, `Handle`, `IOR`, `EJBHandle`, or `RMIReference`. If the MBean server cannot process the given target object type, this method will throw an `InvalidTargetTypeException`.

If the `targetObject` field of an operation's descriptor is set and is valid, then it overrides the managed resource setting for that operation's invocation.

DynamicMBean Implementation

The DynamicMBean interface defines the following methods:

- getMBeanInfo
- getAttribute and getAttributes
- setAttribute and setAttributes
- invoke

The description of these methods is given in "DynamicMBean Interface" on page 41. Here, we define how the model MBean implementation expresses the functionality of each method of the interface.

getMBeanInfo

Returns the ModelMBeanInfo object which implements the ModelMBeanInfo interface for the ModelMBean. Valid attributes, constructors, operations, and notifications defined by the managed resource can be retrieved from the ModelMBeanInfo with the getOperations, getConstructors, getAttributes, and getNotifications methods.

The ModelMBeanInfo instance returns ModelMBeanOperationInfo, ModelMBeanConstructorInfo, ModelMBeanAttributeInfo, and ModelMBeanNotificationInfo arrays, respectively. These classes extend MBeanOperationInfo, MBeanConstructorInfo, MBeanAttributeInfo, and MBeanNotificationInfo, respectively. These extensions must implement the DescriptorAccess interface which sets and returns the descriptor associated with each of these metadata classes. The ModelMBeanInfo also maintains a descriptor for the model MBean referred to as the *MBean descriptor*.

getAttribute and getAttributes

Invoked to get attribute information from this instance of the ModelMBean implementation synchronously. Model MBeans which support attribute value caching will do cache checking and refreshing in this method. Model MBean caching policy is set and values are cached in the descriptor for each attribute. If the model MBean supports the getMethod field of the descriptor (assignment of an operation to be invoked when a get is requested for an attribute) then this method will invoke that operation and return its results as the attribute value. If no value or getMethod descriptor fields are defined the defaultValue field is returned. If no default value is defined then null will be returned.

If caching is supported, then the following algorithm will be used. The model MBean will check for attribute value staleness. Staleness is determined from the CurrencyTimeLimit and LastUpdatedTime fields in the descriptor for the

attribute in its ModelMBeanAttributeInfo object. If CurrencyTimeLimit is 0, then the value will always be stale. If CurrencyTimeLimit is -1, then the value will never be stale.

If the value in the model MBean is set and not stale, then it will return this value without invoking any methods on the managed resource. If the attribute value is stale, then the model MBean will invoke the operation defined in the getMethod field of the attribute descriptor. The returned value from this invocation will be stored in the model MBean as the current value. LastUpdatedTime will be reset to the current time. If a getMethod is not defined and the value is stale, then the defaultValue from the Descriptor for the attribute will be returned.

setAttribute and setAttributes

Invoked to set information for an attribute of this instance of the ModelMBean implementation synchronously. The model MBean will invoke the operation defined in the setMethod field of the attribute descriptor. If no setMethod operation is defined then only the value field of the attribute's descriptor will be set. Invocation of this method where the new attribute value does not match the current attribute value causes an AttributeChangeNotification to be generated.

If caching is supported by the model MBean, the new attribute value will be cached in the value field of the descriptor if the currencyTimeLimit field of the descriptor is not 0. The LastUpdatedTime field should be set whenever the value field is set.

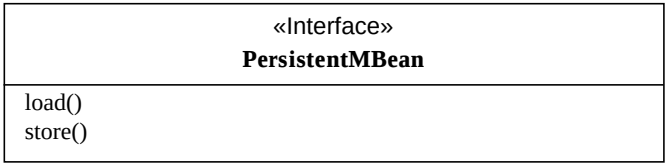
invoke

The invoke method will execute the operation name passed in with the parameters passed in, according to the DynamicMBean interface. The method will be invoked on the model MBean's managed resource (as set by the setManagedResource method). If the targetObject field of the descriptor is set and the value of the targetObjectType field is valid for the implementation, then the method will be invoked on the value of the targetObject instead. Valid values for targetObjectType include, but are not limited to, ObjectReference, IOR, EJBHandle, and RMIReference.

If operation caching is supported, the response from the operation will be cached in the lastReturnedValue and LastUpdatedTime fields of the operation's descriptor if the currencyTimeLimit field in the operation's descriptor is not 0. If the invoke is executed for a method and lastReturnedValue field does not contain a stale value then it will be returned and the associated method will not actually be executed.

`PersistentMBean` Interface

This interface is implemented by all model MBeans. If the model MBean is not persistent or not responsible for its own persistence then these methods may do nothing. The methods of the `PersistentMBean` interface are not intended to be called directly by management applications. Rather, they are called by the required model MBean to implement the persistence policy advertised by the MBean descriptor, to the level that it is supported by the JMX agent's runtime environment.



load

Locates the MBean in a persistent store and primes this instance of the MBean with the stored values. Any currently set values are overwritten. Should only be called by an implementation of the `ModelMBean` interface. (optional)

store

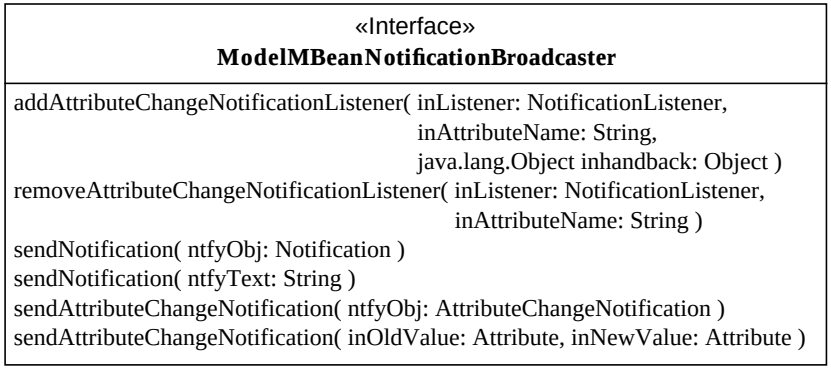
Writes the MBean in a persistent store. Should only called by an implementation of the `ModelMBean` interface to store itself according to persistence policy for the MBean. When used, it may be called with every invocation of `setAttribute` or on a periodic basis. (optional)

`ModelMBeanNotificationBroadcaster` Interface

This interface extends the `NotificationBroadcaster` interface and must be implemented by any MBean wishing to broadcast custom, generic, or attribute change notifications to listeners. Model MBeans must implement this interface.

In the model MBean, `AttributeChangeNotifications` are sent to a separate set of listeners than those that other notifications would go to. All other notifications should go to listeners who registered using the methods defined in the `NotificationBroadcaster` interface. `AttributeChangeNotifications` can also be sent to all notification listeners simply by using the `NotificationBroadcaster` interface alone.

The model MBean sends an `AttributeChangeNotification` to all registered notification listeners whenever a value change for the attribute in the model MBean occurs. By default, no `AttributeChangeNotification` will be sent unless a listener is explicitly registered for them. Normally, the `setAttribute` on the model MBean invokes the set method defined for the attribute on the managed resource directly. Alternatively, managed resources can use the attribute change notification to trigger internal actions to implement the intended effect of changing the attribute value on the model MBean.



`ModelMBeanNotificationBroadcaster` Implementation

The `ModelMBeanNotificationBroadcaster` interface extends the `NotificationBroadcaster` interface for its `addNotificationListener` and `removeNotificationListener` methods. The following methods are specific to open MBeans.

addAttributeChangeNotificationListener

Registers an object which implements the `NotificationListener` interface as a listener for `AttributeChangeNotifications` from this MBean.

removeAttributeChangeNotificationListener

Removes a listener for `AttributeChangeNotifications` from the MBean.

sendAttributeChangeNotification (with AttributeChangeNotification)

Sends the given `AttributeChangeNotification` object to all registered listeners.

sendAttributeChangeNotification (with new and old Attributes)

Creates and sends an `AttributeChangeNotification` to all registered listeners.

sendNotification (with Notification)

Sends the given `Notification` object to all registered listeners.

sendNotification (with String)

Creates a `Notification` named "`generic`" of type `jmx.modelmbean.generic` and sends it to all registered listeners. The source of the notification is this `ModelMBean` instance, sequence 1, and severity of 5 (informative).

Descriptors

The `ModelMBeanInfo` interface publishes metadata about the attributes, operations, and notifications in the management interface. The model MBean *descriptors* contain behavioral information and policies about the same management interface. A descriptor consists of a set of fields, each of which is a `String` name and `Object` value pair. They can be used to store any additional metadata about the management information. The managed resource or management applications can add, modify, or remove fields in any model MBean descriptor at run time.

Some standard field names are reserved and predefined in this specification to handle common data management policies such as caching and persistence. The descriptors also contain the names for the getter and setter operations for attributes. This allows applications to distribute attribute support naturally across the application, regardless of class, and to change that responsibility at runtime.

Descriptors are objects which implement the `Descriptor` interface. They are accessible through the methods defined in the `DescriptorAccess` interface and implemented in the `ModelMBeanAttributeInfo`, `ModelMBeanOperationInfo`, `ModelMBeanConstructorInfo`, and `ModelMBeanNotificationInfo` classes. Arrays of these classes are accessed through the `ModelMBeanInfo` instance. Each of these returns a descriptor which contains information about the component it describes. A managed resource can define the values in the descriptors by constructing a `ModelMBeanInfo` object and using it to define its model MBean through the `setModelMBeanInfo` method or through the `ModelMBean` constructor.

Attribute Behavior

For an attribute, if the descriptor in the `ModelMBeanAttributeInfo` for it has no method signature associated with it, then no managed resource method can be invoked to satisfy it. This means that for `setAttribute` the value is simply recorded in the descriptor, and any attribute change notification listeners are sent a `AttributeChangeNotification`. For `getAttribute`, the current value for the attribute in the model MBean is simply returned from the descriptor and its value cannot be refreshed from the managed resource. This can be useful to minimize managed resource interruption for static resource information. The attribute descriptor also include policy for managing its persistence, caching, and protocol mapping. For operations, the method signature must be defined. For notifications, type, id, severity, and logging policy are optionally defined.

Notification Logging Policy

The model MBean will log notifications if the `log` fields of the MBean descriptor or of the `ModelMBeanNotificationInfo` descriptor is set to true. A `logfile` field must also be defined with a fully qualified file name at one of these levels to indicate where the notifications should be logged. The setting at the `ModelMBeanNotificationInfo` level will take precedence over the setting at the MBean descriptor level. If the `ModelMBean` implementation or the JMX agent does not support logging, then the `log` and `logfile` fields are ignored.

Persistence Policy

Persistence is handled within the model MBean. However, this does not mean that a model MBean *must* implement persistence itself. Different implementations of the JMX agent may have different levels of persistence. When there is no persistence, objects will be completely transient in nature. In a simple implementation, the `ModelMBeanInfo` may be serialized into a flat file. In a more complex environment persistence may be handled by the JMX agent in which the model MBean has been instantiated. If the JMX agent is not transient and the model MBean is persistable it should support persistence policy at the attribute level and model MBean level.

The persistence policy may switch persistence off, force persistence on checkpoint intervals, allow it to occur whenever the model MBean is updated, or throttle the update persistence so that it does not write out the information any more frequently than a certain interval. If the model MBean is executing in an environment where management operations are transactional, this should be shielded from the managed resource. If the managed resource must be aware of the transaction, then this will mean that the managed resource depends on a proprietary version of the JMX agent and model MBean to be accessible.

The `ModelMBean` constructor will attempt to prime itself by calling the `ModelMBean.load` method. This method must determine if this model MBean has a persistent representation by invoking the `findPersistent` method. Then the load method must determine where this data is located, retrieve it, and initialize the model MBean. For simpler representations, the directory and filename to be used for persistence can be defined right in the MBean descriptor's `PersistLocation` and `PersistName` fields. The model MBean can, through JDBC™ (Java Database Connectivity) operations, write data to and populate the model MBeans from any number of data storage options such as an LDAP server, a database application, a flat file, an NFS file, an FAS file, or an internal high performance cache.

The `load` method allows the JMX agent to be independent and ignorant of data locale information and knowledge. This allows data location to vary from one installation to another depending on how the JMX agent and managed resource are installed and configured. It also permits managed resource configuration data to be defined within the directory service for use by multiple managed resource instances or JMX agent instances. In this way, data locale has no impact on the interaction between the managed resource, its model MBean, the JMX agent, the adaptor or the management system. As with all data persistence issues, the platform data services characteristics may have an impact upon performance and security.

Since the persistence policy can be set at the model MBean attribute level, all or some of the model MBean attributes can be stored by the `ModelMBean`. The model MBean will detect that it has been updated and invoke its own `store` method. If the model MBean service is configured to periodically checkpoint model MBeans by invoking the `ModelMBean.store` method. Like the `load` method, the `store` method must determine where the data should reside and store it there appropriately.

The JMX agent's persistence setting would apply to all of its model MBeans unless one of them defines overriding policies. The model MBean persistence policy provides a specified persistence event (update/checkpoint) and timing granularity concerning how the designated attributes, if any, are stored. The model MBean persistence policy will allow persistence on a "whenever updated" basis, a "periodic checkpoint" basis, or a "never persist" basis. If no persistence policy for a model MBean is defined, then its instance will be transient.

Cached Values Behavior

The descriptor for an attribute or operation contains the cached value and default value for the data along with the caching policy. In general, the adaptors access the application's `ModelMBean` as it is returned by the JMX agent. If the data requested by the adaptor is current, then the managed resource is not interrupted with a data retrieval request. Therefore, direct interaction with the managed resource is not required for each interaction with the management system. This helps minimize the impact of management activity on runtime application resources and performance.

The attribute descriptor contains `currencyTimeLimit` and `lastUpdatedTimeStamp` fields which are expressed in units of seconds. If the current time is past `lastUpdateTimeStamp` + `currencyTimeLimit`, then the attribute value is *stale*. If a `getAttribute` is received for an attribute with a stale value (or no value) in the descriptor, then the `getMethod` for the attribute will be invoked and the returned value will be recorded in the `value` field in the descriptor for the attribute, `lastUpdatedTimeStamp` will be reset, and the caller will be handed the new value. If there is no `getMethod` defined, then the default value from `default` field in the descriptor for the attribute will be returned.

Protocol Map Support

The model MBean's default behavior and simple APIs satisfies the management needs of most applications. However, the interfaces of a model MBean also allow complex managed resource management scenarios. The model MBean APIs allow mapping of the application's model MBean attributes to existing management data models, i.e. specific MIBs or CIM objects through the `ProtocolMap` field of the descriptor. Conversely, the managed resource can take advantage of generic mappings to MIBs and CIM objects generated by tools interacting with the JMX agent. For example, a MIB generator can interact with the JMX agent and create a MIB file that is loaded by an SNMP management system. The generated MIB file can represent the resources known by the JMX agent. The applications represented by those resources do not have to be cognizant of how the management data is mapped to the MIB. This scenario will also work for other definition files required by management systems.

The `ProtocolMap` field of an attribute's descriptor must contain a reference to an instance of a class which implements the `Descriptor` interface. The contents (or mappings) of the `ProtocolMap` must be appropriate for the attribute. The entries in the `ProtocolMap` can be updated or augmented at runtime.

Export Policy

If the JMX agent implementation supports operating in a multi-JMX agent environment, then the JMX agent will need to advertise its existence and availability with the appropriate directory or lookup service. The JMX agent may also need to register MBeans that wish to be locatable from other JMX agents without advance knowledge about which JMX agent the MBean is currently registered with. MBeans that want to be locatable in this type of environment should define an `export` field in the MBean descriptor in its `ModelMBeanInfo` object.

The value of the `export` field should be the external name or object required to export the MBean appropriately. If the JMX agent does not support interoperating with a directory or lookup service and the `export` field is defined, then the field will be ignored. If the value of the `export` field is `F` or `false` or the `export` field is undefined, then the MBean will not be exported.

Visibility Policy

Model MBeans in the JMX specification provide developers of managed resources with the ability to instrument manageability that supports both their custom, stand-alone, domain manager as well as interchangeable enterprise managers. However, the level of detail that should be available from these types of managers can be significantly different. Enterprise managers may want to interact with higher level management objects. Domain managers generally manage all details of the application. Most management systems show large grain objects on a user interface screen and show small grain objects on a detailed or advanced screen. The `visibility` field in the descriptor is a hint about the level of granularity an MBean, attribute, or operation represents. The `visibility` field can be used by a custom implementation of a protocol adaptor/connector or by a management system to filter out MBeans, attributes, or operations that it doesn't need to represent.

The `visibility` field's value is an integer ranging from `1` to `4`. The largest grain is `1`, for an MBean or a component which is nearly always visible. The smallest grain is `4`, for an MBean or a companionate which is only visible in special cases. The JMX specification does not further define these levels.

Presentation Behavior

A `PresentationString` field can be defined in any descriptor. This string is an XML formatted string meant to provide hints to a console so that it can generate user interfaces for a management object. A standard set of presentation fields have not yet been defined.

Predefined Descriptor Fields

The fields in each descriptor describe standard and custom information about model MBean components. All predefined fields for the each of the descriptors are specified below. The fields defined here are standardized so that the management instrumentation is portable between implementations of model MBeans. More fields may be defined in a management to store custom information as needed.

MBean Descriptor Fields

These are the predefined fields for the MBean descriptor. These values are valid for the entire model MBean. These values may be overridden by descriptor fields with the same name defined at the attribute, operation, or notification level. Optional fields are in italics:

`name` - The case-sensitive name of the MBean.

`descriptorType` - String which always contains the value "`MBean`".

`displayName` - Displayable name of attribute. In the absence of a value, the value of the `name` field should be used instead.

`persistPolicy` - Takes on one of the following values:

- `Never` - The attribute is never stored. This is useful for highly volatile data or data that only has meaning within the context of a session or execution period.
- `OnTimer` - The attribute is stored whenever the model MBean's persistence timer, as defined in the `persistPeriod` field, expires.
- `OnUpdate` - The attribute is stored every time the attribute is updated.
- `NoMoreOftenThan` - The attribute is stored every time it is updated unless the updates are closer together than the `persistPeriod`. This acts as an update throttling mechanism that helps prevent temporarily highly volatile data from impacting performance.

`persistPeriod` - Valid only if the `persistPolicy` field's value is `OnTimer` or `NoMoreOftenThan`. For `OnTimer`, the attribute is stored at the beginning of each `persistPeriod` starting from when the value is first set. For `NoMoreOftenThan`, the attribute will be stored every time it is updated unless the `persistPeriod` hasn't elapsed since the previous storage.

persistLocation - The fully qualified directory where files representing the persistent MBeans should be stored (for this reference implementation). For other implementations this value may be a keyword or value to assist the appropriate persistence mechanism.

persistName - The filename into which this MBean should be stored. This should be the same as the MBean's name (for this reference implementation). For other implementations this value may be a keyword/value to assist the appropriate persistence mechanism.

log - A boolean where `true` indicates that all sent notifications should be logged to a file, and `false` indicates that no notification logging should be done. This setting can be overridden for a particular notification by defining the `log` field in the notification descriptor.

logFile - The fully qualified file name where notifications should be logged. If logging is `true` and the `logFile` is not defined or invalid, no logging will be performed.

currencyTimeLimit - Time period in seconds from when an attribute value is current and not stale. If the saved value is current then that value is returned and the `getMethod` (if defined) is not invoked. If the `currencyTimeLimit` is `0`, then the value must be retrieved on every request. If `currencyTimeLimit` is `-1` then the value is never stale.

export - Its value can be any object that is serializable and contains the information necessary to make the MBean locatable. `null` indicates that the MBean should not be exposed to other JMX Agents. A defined value indicates that the MBean should be exposed to other JMX Agents and also be findable when the JMX agent address is unknown. If exporting MBeans and MBean servers is not supported, then this field is ignored.

visibility - Integer set from `1` to `4` which indicates a level of granularity for the MBean. `1` is for the large grain and most often viewed MBeans. `4` is the smallest grain and possibly the least often viewed MBeans. This value may be used by adaptors or management applications.

presentationString - XML-encoded string which describes how the attribute should be presented.

Attribute Descriptor Fields

An attribute descriptor represents the metadata for one of the attributes of a model MBean. Optional fields are in italics:

name - The case-sensitive name of the attribute.

descriptorType - A string which always contains the value "`attribute`".

value - The value of this field is the object representing the current value of attribute, if set. This is, in effect, the cached value of the attribute that will be returned if the currencyTimeLimit is not stale.

default - An object which is to be returned if the value is not set and the getMethod is not defined.

displayName - The displayable name of the attribute.

getMethod - Operation name from the operation descriptors to be used to retrieve the value of the attribute. The returned object is saved in the value field.

setMethod - Operation name from the operation descriptors to be used to set the value of the attribute in the managed resource. The new value will also be saved in the value field.

protocolMap - The value of this field must be a Descriptor object. It contains the set of protocol name and mapped protocol value pairs. This allows the attribute to be associated with a particular identifier (CIM schema, SNMP MIB Oid, etc.) for a particular protocol. This descriptor should be set by the managed resource and used by the adaptors as hints for representing this attribute to management applications.

persistPolicy - Takes on of one of the following values:

- Never - The attribute is never stored. This is useful for highly volatile data or data that only has meaning within the context of a session or execution period.
- OnTimer - The attribute is stored whenever the model MBean service persistPeriod expires.
- OnUpdate - The attribute is stored every time the attribute is updated.
- NoMoreOftenThan - The attribute is stored every time it is updated unless the updates are closer together than the persistPeriod. This acts as an update throttling mechanism that helps prevent temporarily highly volatile data from impacting performance.

persistPeriod - Valid only if persistPolicy is OnTimer or NoMoreOftenThan. For OnTimer the attribute is stored at the beginning of each persistPeriod starting from when the value is first set. For NoMoreOftenThan the attribute will be stored every time it is updated as long as the updates are not closer together than the persistPeriod.

currencyTimeLimit - Time period in seconds from when a value is set that the value is current and not stale. If the value is current then the saved value is returned and the getMethod (if defined) is not invoked. If the currencyTimeLimit is 0, then the value must be retrieved for every request. If currencyTimeLimit is -1 then the value is never stale.

lastUpdatedTimeStamp - Time stamp from when the value field was last updated.

iterable - A boolean value where `true` indicates the `value` is enumerable, and `false` indicates the `value` is not enumerable.

visibility - Integer set from `1` to `4` which indicates a level of granularity for the MBean attribute. `1` is for the large grain and most often viewed MBean attributes. `4` is the small grain and the least often viewed MBean attributes. This value may be used by adaptors or management applications.

presentationString - XML-encoded string which describes how the attribute should be presented.

Operation Descriptor Fields

The operation descriptor represents the metadata for operations of a model MBean. Optional fields are in italics:

name - The case-sensitive operation name.

descriptorType - A string which always contains the value "`operation`".

displayName - Display name of the operation.

lastReturnedValue - The value that was returned from the operation the last time it was executed. This allows the caching of operation responses. Operation responses are only cached if the `currencyTimeLimit` field is not `0`.

currencyTimeLimit - The period of time in seconds that the `lastReturnedValue` is current and not stale. If the `lastReturnedValue` is current then it is returned without actually invoking the method on the managed resource. If the `value` is stale then the method is invoked. If `currencyTimeLimit` is `0`, then the value is always stale and is not cached. If the `currencyTimeLimit` is `-1`, then the value is never stale.

lastReturnedTimeStamp - The time stamp of when the `lastReturnedValue` field was updated.

visibility - Integer set from `1` to `4` which indicates a level of granularity for the MBean operation. `1` is for the large grain and most often viewed MBean operations. `4` is the smallest grain and the least often viewed MBean operations. This value may be used by adaptors or management applications.

presentationString - XML-encoded string that defines how to present the operation, parameters, and return type to a user.

Notification Descriptor Fields

Notification `Descriptor` represents the metadata for the notifications of a model MBean. Optional fields are in italics:

`name` - The case-sensitive name of the notification.

`descriptorType` - A string which always contains the value "`notification`".

`severity` - Integer range of `0` to `6` interpreted as follows:

- `0` • Unknown, Indeterminate
- `1` • Non recoverable
- `2` • Critical, Failure
- `3` • Major, Severe
- `4` • Minor, Marginal, or Error
- `5` • Warning
- `6` • Normal, Cleared, or Informative

`messageId` - ID for the notification. Usually used to retrieve text to match the ID to minimize message size or perform client side translation.

`log` - A boolean that is `true` if this notification should be logged to a file and `false` if not. There can be a default value for all notifications of an MBean by defining the `log` field in the MBean descriptor.

`logFile` - The fully qualified file name where notifications should be logged. If `log` is `true` but the `logFile` is not defined or invalid, no logging will be performed. This setting can also have an MBean-wide default by defining the `logFile` field in the MBean descriptor.

`PresentationString` - XML-encoded string which describes how to present the notification to a user.

PART II JMX Agent Specification

CHAPTER 5

Agent Architecture

This chapter gives an overview of the JMX agent architecture and its basic concepts. It serves as an introduction to the agent specification of the Java Management extensions.

Overview

A JMX agent is a management entity which runs in a JVM and acts as the liaison between the MBeans and the management application. A JMX agent is composed of an *MBean server*, a set of MBeans representing *managed resources*, a minimum number of *agent services* implemented as MBeans, and typically at least one *protocol adaptor* or *connector.*

The key components in the JMX agent architecture can be further defined as follows:

- MBeans which represent managed resources, as specified in Part I, "JMX Instrumentation Specification" on page 31.
- The MBean server which is the key-stone of this architecture and the central registry for MBeans. All management operations applied to MBeans need to go through the MBean server.
- Agent services which can either be components defined in this specification or services developed by third parties. The agent service MBeans defined by the JMX specification provide:
  - *Dynamic loading* services which allow the agent to instantiate MBeans using java classes and native libraries dynamically downloaded from the network
  - *Monitoring* capabilities for attribute values in MBeans; the service notifies its listeners upon detecting certain conditions
  - A *timer* service that can send notifications at pre-determined intervals and act as a scheduler

- A *relation* service that defines associations between MBeans and maintains the consistency of the relation

Remote management applications may access an agent through different protocol adaptors and connectors. These objects are part of the agent application but they are not part of the JMX agent specification. They will be defined in the distributed services specification in the next phase of Java Management extensions.

FIGURE 5-1 shows how the agent's components relate to each other and to a management application.

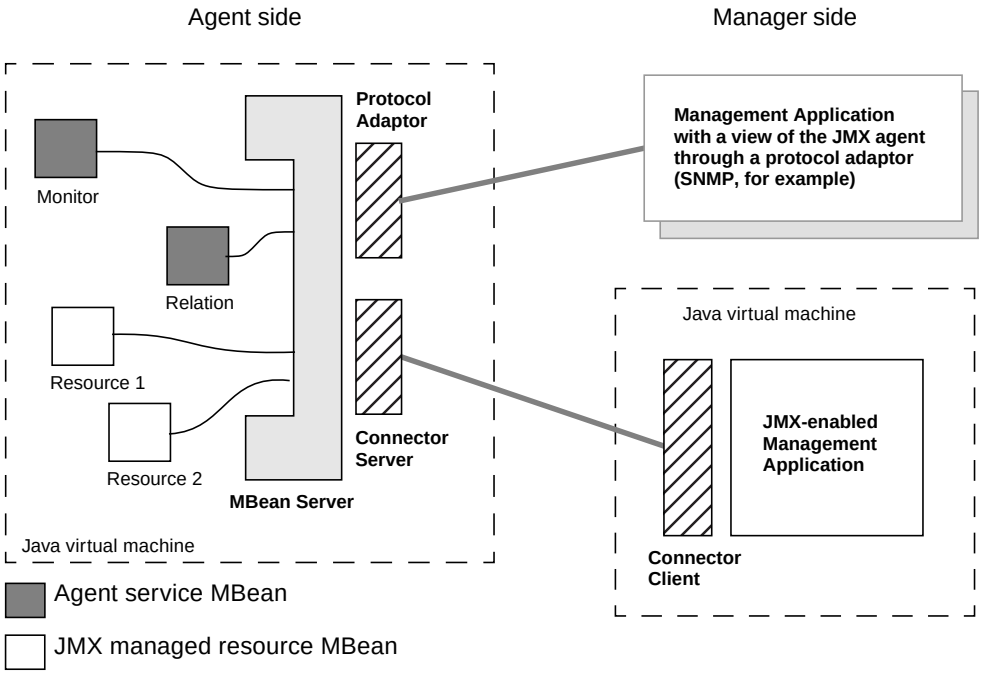


FIGURE 5-1 Key Concepts of the JMX Agent Architecture

The JMX architecture allows objects to perform the following operations on a JMX agent. These objects may either be in the agent-side application or in a remote management application. They can:

- Manage existing MBeans by:
  - Getting their attribute values
  - Changing their attribute values
  - Invoking operations on them
- Get notifications emitted by any MBean
- Instantiate and register new MBeans from:

- Java classes already loaded into the agent JVM
- New classes downloaded from the local machine or from the network

- Use the agent services to implement management policies involving existing MBeans

In the JMX architecture, all of these operations are performed, either directly or indirectly through the MBean server of the JMX agent.

JMX Compliant Agent

All of the agent components and classes described in the agent specification are mandatory. In order to conform to the agent specification, a JMX agent implementation must therefore provide the following components:

- The MBean server implementation
- All of the agent services:
  - Dynamic class loading
  - Monitoring
  - Timer
  - Relation

All of these components are specified in this document and in the associated Javadoc API. The Agent Compatibility Test Suite will check that all components are actually provided by an implementation of the specification.

Protocol Adaptors and Connectors

Protocol adaptors and connectors are not covered in this phase of the specification, they are described here to provide a more complete overview of the agent's architecture.

Protocol adaptors and *connectors* make the agent accessible from remote management applications. They provide a view through a specific protocol of the MBeans instantiated and registered in the MBean server. They enable a management application outside the JVM to:

- Get or set attributes of existing MBeans
- Perform operations on existing MBeans
- Instantiate and register new MBeans
- Register for and receive notifications emitted by MBeans

Connectors are used to connect an agent with a remote JMX-enabled management application; that is: a management application developed using the JMX distributed services. This kind of communication involves a connector server in the agent and a connector client in the manager.

These components convey management operations transparently point-to-point over a specific protocol. The distributed services on the manager side provide a remote interface to the MBean server through which the management application can perform operations. A connector is specific to a given protocol, but the management application can use any connector indifferently since they have the same remote interface.

Protocol adaptors provide a management view of the JMX agent through a given protocol. They adapt the operations of MBeans and the MBean server into a representation in the given protocol, and possibly into a different information model, for example SNMP.

Management applications that connect to a protocol adaptor are usually specific to the given protocol. This is typically the case for legacy management solutions that rely on a specific management protocol. They access the JMX agent not through a remote representation of the MBean server, but through operations that are mapped to those of the MBean server.

Both connector servers and protocol adaptors use the services of the MBean server in order to apply the management operation they receive to the MBeans, and in order to forward notifications to the management application.

For an agent to be manageable, it must include at least one protocol adaptor or connector server. However, an agent can include any number of these, allowing it to be managed remotely through different protocols simultaneously.

The adaptors and connectors provided by a JMX implementation should be implemented as MBeans. This allows them to be managed as well as to be loaded and unloaded dynamically, as needed.

CHAPTER 6

Foundation Classes

The foundation classes describe objects which are used as argument types or returned values in methods of various JMX APIs. The foundation classes described in this chapter are:

- `ObjectName`
- `ObjectInstance`
- `Attribute` and `AttributeList`
- JMX exceptions

The following classes are also considered as foundation classes; they are described in “MBean Metadata Classes” on page 53:

- `MBeanInfo`
- `MBeanFeatureInfo`
- `MBeanAttributeInfo`
- `MBeanOperationInfo`
- `MBeanConstructorInfo`
- `MBeanParameterInfo`
- `MBeanNotificationInfo`

All foundation classes are included in the JMX instrumentation API so that MBeans may be developed solely from the instrumentation specification, yet be manipulated by a JMX agent.

`ObjectName` Class

An object name uniquely identifies an MBean within an MBean server. Management applications use this object name to identify the MBean on which to perform management operations. The class `ObjectName` represents an object name which consists of two parts:

- A domain name

- An unordered set of one or more key properties

The components of the object name are described below.

Domain Name

The domain name is a case-sensitive string. It provides a structure for the naming space within a JMX agent or within a global management solution. The domain name part may be omitted in an object name, as the MBean server is able to provide a *default domain*. When an exact match is required (see “Pattern Matching” on page 107), omitting the domain name will have the same result as using the default domain defined by the MBean server.

How the domain name is structured is application-dependent. The domain name string may contain any characters except those which are object name separators or wildcards, namely the colon, comma, equals sign, asterisk or question mark (`:,=*?`). JMX always handles the domain name as a whole, therefore any semantic sub-definitions within the string are opaque to a JMX implementation.

Key Property List

The key property list enables you to assign unique names to the MBeans of a given domain. A key property is a property-value pair, where the property does not need to correspond to an actual attribute of an MBean.

The key property list must contain at least one key property. It may contain any number of key properties, whose order is not significant.

String Representation of Names

Object names are usually built and displayed using their string representation, which has the following syntax:

```
[domainName]:property=value[,property=value]*
```

The domain name may be omitted to designate the default domain.

The *canonical name* of an object is a particular string representation of its name where the key properties are sorted in lexical order. This representation of the object name is used in lexicographic comparisons performed in order to select MBeans based on their object name.

Pattern Matching

Most of the basic MBean operations (for example, create, get and set attribute) need to uniquely identify one MBean by its object name. In that case, *exact matching* of the name is performed.

On the other hand, for query operations, it is possible to select a range of MBeans by providing an object name expression. The MBean server will use *pattern matching* on the names of the objects. The matching features for the name components are as follows:

Domain Name

The matching syntax is consistent with standard file globing, in other words:

* matches any character sequence, including an empty one
? matches any one single character

Key Property List

There is no wildcard matching performed on property names nor on property values. Only complete property-value pairs are used in pattern matching. While key properties are atomic, the list of key properties may be incomplete and used as a pattern.

The * is the wildcard for key properties, it replaces any number of key properties which may take on any value. If the whole key property list is given as *, this will match all the objects in the selected domain(s). If at least one key property is given in the list pattern, the wildcard may be located anywhere in the given pattern, provided it is still a comma-separated list: "`:property=value,*`" and "`:*,property=value`" are both valid patterns. In this case, objects having the given key properties as subsets of their key property list will be selected.

If no wildcard is used, only object names matching the complete key property list will be selected. Again, the list is unordered, so the key properties in the list pattern can be given in any order.

Pattern Matching Examples

Assuming that the MBeans with the following names are registered in the MBean server:

```
MyDomain:description=Printer,type=laser
MyDomain:description=Disk,capacity=2
DefaultDomain:description=Disk,capacity=1
DefaultDomain:description=Printer,type=ink
```

```
DefaultDomain:description=Printer,type=laser,date=1993
Socrates:description=Printer,type=laser,date=1993
```

Here are some examples of queries that can be performed using pattern matching:

- "*:*" will match all the objects of the MBean server. A null string object or empty string ("") name used as a pattern is equivalent to "*:*".
- ":*" will match all the objects of the default domain
- "MyDomain:*" will match all objects in MyDomain
- "??Domain:*" will also match all objects in MyDomain
- "*Dom*:*" will match all objects in MyDomain and DefaultDomain
- "*:description=Printer,type=laser,*" will match the following objects:

  ```
  MyDomain:description=Printer,type=laser
  DefaultDomain:description=Printer,type=laser,date=1993
  Socrates:description=Printer,type=laser,date=1993
  ```

- "*Domain:description=Printer,*" will match the following objects:

  ```
  MyDomain:description=Printer,type=laser
  DefaultDomain:description=Printer,type=ink
  DefaultDomain:description=Printer,type=laser,date=1993
  ```

ObjectInstance Class

The ObjectInstance class is used to represent the link between an MBean's object name and its Java class. It is the full description of an MBean within an MBean server, though it does not allow you to access the MBean by reference.

The ObjectInstance class contains the following elements:

- The Java class name of the corresponding MBean
- The ObjectName registered for the corresponding MBean
- A test for equality with another ObjectInstance

An ObjectInstance is returned when an MBean is created and is used subsequently for querying.

`Attribute` and `AttributeList` Classes

These classes are used to represent MBean attributes and their value. They contain the attribute name string and its value cast as an `Object` instance.

JMX defines the following classes:

- The `Attribute` class represents a single attribute-value pair
- The `AttributeList` class represents a list of attribute-value pairs

The `Attribute` and `AttributeList` objects are typically used for conveying the attribute values of an MBean, as the result of a getter operation, or as the argument of a setter operation.

JMX Exceptions

The JMX exceptions are the set of exceptions that are thrown by different methods of the JMX interfaces. This section describes what error cases are encapsulated by these exceptions.

JMX exceptions mainly occur:

- While the MBean server or JMX agent services perform operations on MBeans
- When the MBean code raises user defined exceptions

The organization of the defined JMX exceptions is based on the nature of the error case (runtime or not) and on the location where it was produced (manager, agent, or during communication).

Only exceptions raised by the agent are within the scope of this release of the specification. This section only describes exceptions that are thrown by the MBean server. Agent services also define and throw particular exceptions, these are described in their respective Javadoc API.

`JMException` Class and Subclasses

As shown in FIGURE 6-1 the base exception class is named `JMException` and it extends the `java.lang.Exception` class. The `JMException` represents all the exceptions thrown by methods of a JMX agent implementation.

In order to characterize the `JMException` and to give information for the location of the exception's source, some subclass exceptions are defined. They are grouped by exceptions thrown while performing operations in general (`OperationsException`), exceptions thrown during the use of the reflection API for invoking MBean methods (`ReflectionException`) and exceptions thrown by the MBean code (`MBeanException`).

The `ReflectionException` wraps the actual core Java exception thrown when using the reflection API. The `MBeanException` should also wrap the actual user defined exception thrown by an MBean method.

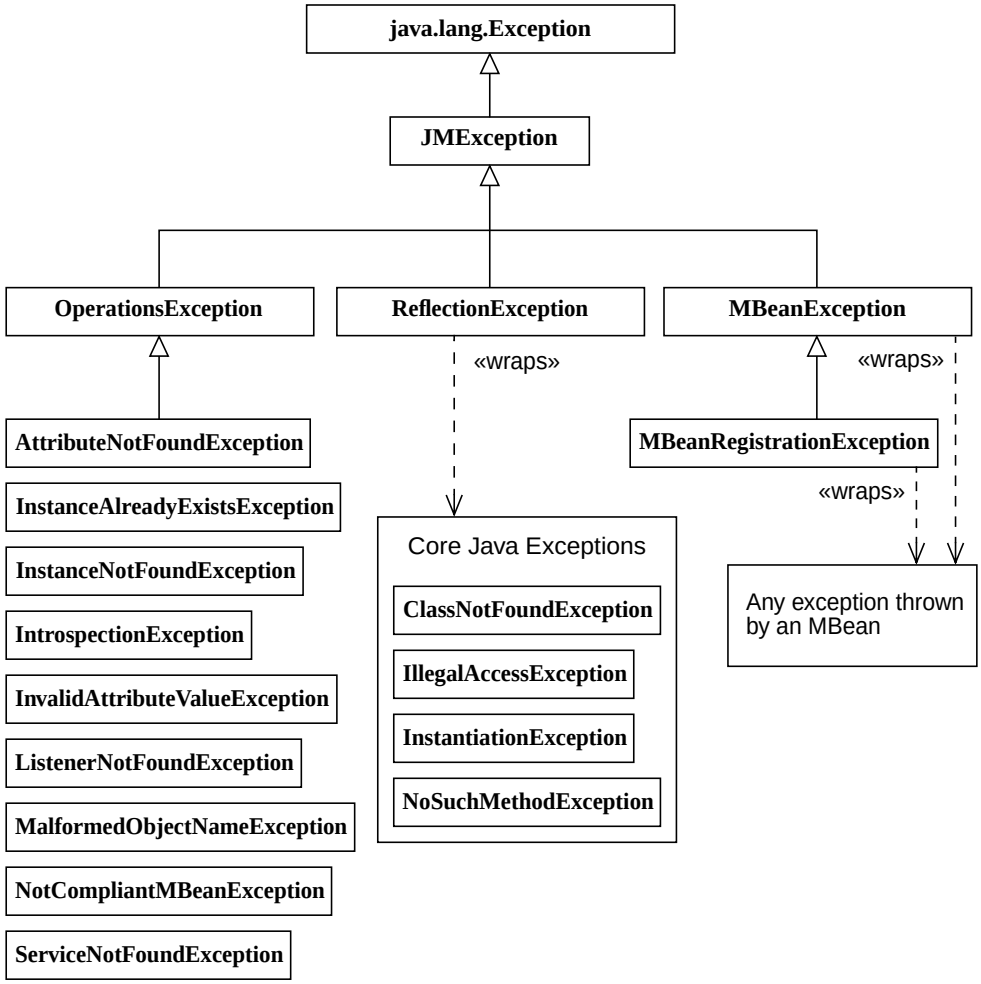


FIGURE 6-1 The JMX Exceptions Object Model

JMRuntimeException Class and Subclasses

As shown in FIGURE 6-2 the base JMX runtime exception defined is named `JMRuntimeException` and it extends the `java.lang.RuntimeException` class. The `JMRuntimeException` represents all the runtime exceptions thrown by methods of a JMX implementation. Like the `java.lang.RuntimeException`, a method of a JMX implementation is not required to declare in its throws clause any subclasses of `JMRuntimeException` that might be thrown during the execution of the method but not caught.

The `JMRuntimeException` is specialized into `OperationsRuntimeException` for representing the runtime exceptions thrown while performing operations in the agent, `MBeanRuntimeException` representing the runtime exceptions thrown by the MBean code, and `RuntimeErrorException` representing errors thrown in the agent and re-thrown as runtime exceptions.

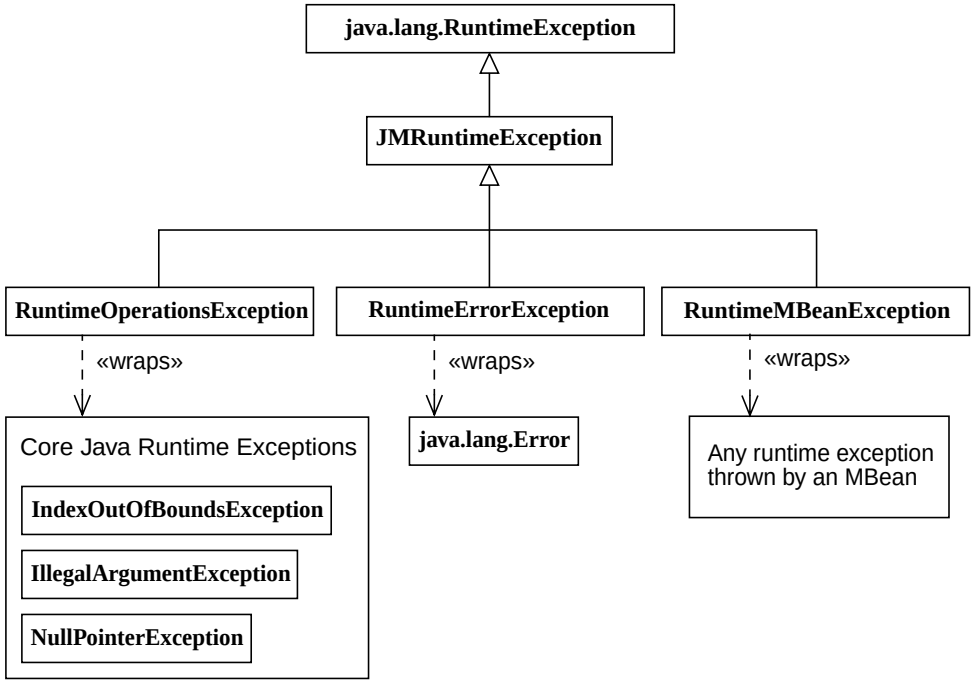


FIGURE 6-2 The JMX Runtime Exceptions Object Model

Description of JMX Exceptions

`JMException` Class

This class represents exceptions thrown by JMX implementations. It does not include the runtime exceptions.

`ReflectionException` Class

This class represents exceptions thrown in the agent when using the java.lang.reflect classes to invoke methods on MBeans. It “wraps” the actual `java.lang.Exception` thrown.

The following are the exception classes that may be “wrapped” in a `ReflectionException`:

- `ClassNotFoundException` - Thrown when an application tries to load in a class through its string name using the `forName` method in class “`Class`”.
- `InstantiationException` - Thrown when an application tries to create an instance of a class using the `newInstance` method in class “`Class`”, but the specified class object cannot be instantiated because it is an interface or an abstract class.
- `IllegalAccessException` - Thrown when an application tries to load in a class through its string name using the `forName` method in class “`Class`”.
- `NoSuchMethodException` - Thrown when a particular method cannot be found.

`MBeanException` Class

This class represents “user defined” exceptions thrown by MBean methods in the agent. It “wraps” the actual “user defined” exception thrown. This exception will be built by the MBean server when a call to an MBean method results in an unknown exception.

`OperationsException` Class

This class represents exceptions thrown in the agent when performing operations on MBeans. It is the superclass for all of the following exception classes, except for the runtime exceptions.

`InstanceAlreadyExistsException` Class

The MBean is already registered in the repository.

`InstanceNotFoundException` Class

The specified MBean does not exist in the repository.

`InvalidAttributeValueException` Class

The specified value is not a valid value for the attribute.

`AttributeNotFoundException` Class

The specified attribute does not exist or cannot be retrieved.

`IntrospectionException` Class

An exception occurred during introspection of the MBean, when trying to determine its management interface.

`MalformedObjectNameException` Class

The format or contents of the information passed to the constructor does not allow a valid `ObjectName` instance to be built.

`NotCompliantMBeanException` Class

This exception occurs when trying to register an object which is not a JMX compliant MBean in the MBean server.

`ServiceNotFoundException` Class

This class represents exceptions raised when a requested service is not supported.

MBeanRegistrationException Class

This class wraps exceptions thrown by the `preRegister` and `preDeregister` methods of the `MBeanRegistration` interface.

JMRuntimeException Class

This class represents runtime exceptions emitted by JMX implementations.

RuntimeOperationsException Class

This class represents runtime exceptions thrown in the agent when performing operations on MBeans. It wraps the actual `java.lang.RuntimeException` thrown.

Here are the exception classes that may be "wrapped" in a `RuntimeOperationsException`:

- `IllegalArgumentException` - Thrown to indicate that a method has been passed an illegal or inappropriate argument.
- `IndexOutOfBoundsException` - Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.
- `NullPointerException` - Thrown when an application attempts to use null in a case where an object is required.

RuntimeMBeanException Class

This class represents runtime exceptions thrown by MBean methods in the agent. It "wraps" the actual `java.lang.RuntimeException` exception thrown. This exception will be built by the MBean server when a call to an MBean method throws a runtime exception.

RuntimeErrorException Class

When a `java.lang.Error` occurs in the agent it should be caught and re-thrown as a `JRuntimeErrorException`.

CHAPTER 7

MBean Server

This chapter describes the Managed Bean server, or MBean server for short, which is the core component of the JMX agent infrastructure.

Role of the MBean Server

The MBean server is a registry for MBeans in the agent. The MBean server is the component which provides the services for manipulating MBeans. All management operations performed on the MBeans are done through the `MBeanServer` interface.

In general, the following kinds of MBeans would be registered in an MBean server:

- MBeans which represent managed resources for management purposes. These resources may be of any kind: application, system, or network resources which provide a Java interface or a Java wrapper.
- MBeans which add management functionality to the agent. This functionality may be fully generic, providing for instance a logging or a monitoring capability, or it may be specific to a technology or to a domain of application. Some of these MBeans are defined by the JMX specification, others will be provided by management application developers.
- Some components of the infrastructure, such as the connector clients and protocol adaptors, may be implemented as MBeans. This allows such components to benefit from the dynamic management infrastructure.

MBean Server Factory

A JMX agent has a factory class for finding or creating an MBean server through the factory's static methods. This allows more flexible agent applications and possibly more than one MBean server in an agent.

The `MBeanServer` interface defines the operations available on a JMX agent. An implementation of the JMX agent specification provides a class that implements the `MBeanServer` interface. Throughout this document, we use the term *MBean server* to refer to the implementation of the `MBeanServer` interface which is available in an agent.

The `MBeanServerFactory` is a class whose static methods return instances of the implementation class. This object is cast as an instances of the `MBeanServer` interface, thereby isolating other objects from any dependency on the MBean server's actual implementation class. When creating an MBean server, the caller can also specify the name of the default domain that will be used in the JMX agent it represents.

An agent application uses these methods to create the single or multiple MBean servers that contain its MBeans. The JMX agent specification only defines the behavior of a single MBean server. The additional behavior required in a JMX agent containing multiple MBean servers is outside the scope of this specification.

The factory also defines static methods for finding an MBean server which has already been created. In this way, objects loaded into the JVM can access an existing MBean server without any prior knowledge of the agent application.

Registration of MBeans

The first responsibility of the MBean server is to be a *registry* for MBeans. MBeans may be registered either by the agent application, or by other MBeans. The interface of the `MBeanServer` class allows two different kinds of registration:

- Instantiation of a new MBean and registration of this MBean in a single operation. In this case, the loading of the java class of the MBean can be done either by using a default class loader, or by explicitly specifying the class loader to use.
- Registration of an already existing MBean instance.

An *object name* is assigned to an MBean when it is registered. The object name is a string whose structure is defined in detail in "`ObjectName` Class" on page 105. The object name allows an MBean to be identified uniquely in the context of the MBean server. This unicity is checked at registration time by the MBean server, which will refuse MBeans with duplicate names.

MBean Registration Control

The MBean developer may exercise some control upon the registering / unregistering of the MBean in the MBean server. This can be done in the MBean by implementing the `MBeanRegistration` interface. Before and after registering and deregistering

an MBean, the MBean server checks dynamically whether the MBean implements the `MBeanRegistration` interface. If this is the case, the appropriate callbacks are invoked.

The `MBeanRegistration` interface is actually an API element of the JMX instrumentation specification. It is described here because it is the implementation of the MBean server which defines the behavior of the registration control mechanism.

Implementing this interface is also the only means by which MBeans can get a reference to the `MBeanServer` with which they are registered. This means that they are aware of their management environment and become capable of performing management operations on other MBeans.

If the MBean developer chooses to implement the `MBeanRegistration` interface, the following methods must be provided:

- `preRegister` - This is a callback method that the MBean server will invoke before registering the MBean. The MBean will not be registered if any exception is raised by this method. This method may throw the `MBeanRegistrationException` which will be re-thrown as is by the MBean server. Any other exception will be caught by the MBean server, encapsulated in an `MBeanRegistrationException` and re-thrown.

  This method may be used to:

  - Allow an MBean to keep a reference on its MBean server.
  - Perform any initialization that needs to be done before the MBean is exposed to management operations.
  - Perform semantic checking on the object name, and possibly provide a name if the object was created without a name.
  - Get information about the environment, for instance, check on the existence of services the MBean depends upon. When such required services are not available, the MBean might either try to instantiate them, or raise a `ServiceNotFoundException` exception.
- `postRegister` - This is a callback method that the MBean server will invoke after registering the MBean. Its boolean parameter will be *true* if the registration was done successfully, and *false* if the MBean could not be registered. If the registration failed, this method can free resources allocated in preregistration.
- `preDeregister` - this is a callback method that the MBean server will invoke before de-registering an MBean.

  This method may throw an `MBeanRegistrationException`, which will be re-thrown as is by the MBean server. Any other exception will be caught by the MBean server, encapsulated in an `MBeanRegistrationException` and re-thrown.The MBean will not be de-registered if any exception is raised by this method.
- `postDeregister` - This is a callback method that the MBean server will invoke after de-registering the MBean.

FIGURE 7-1 describes the way the methods of the `MBeanRegistration` are called by the MBean server when an MBean registration or a de-registration is performed. The methods illustrated with a thick border are `MBeanServer` methods, the others are implemented in the MBean.

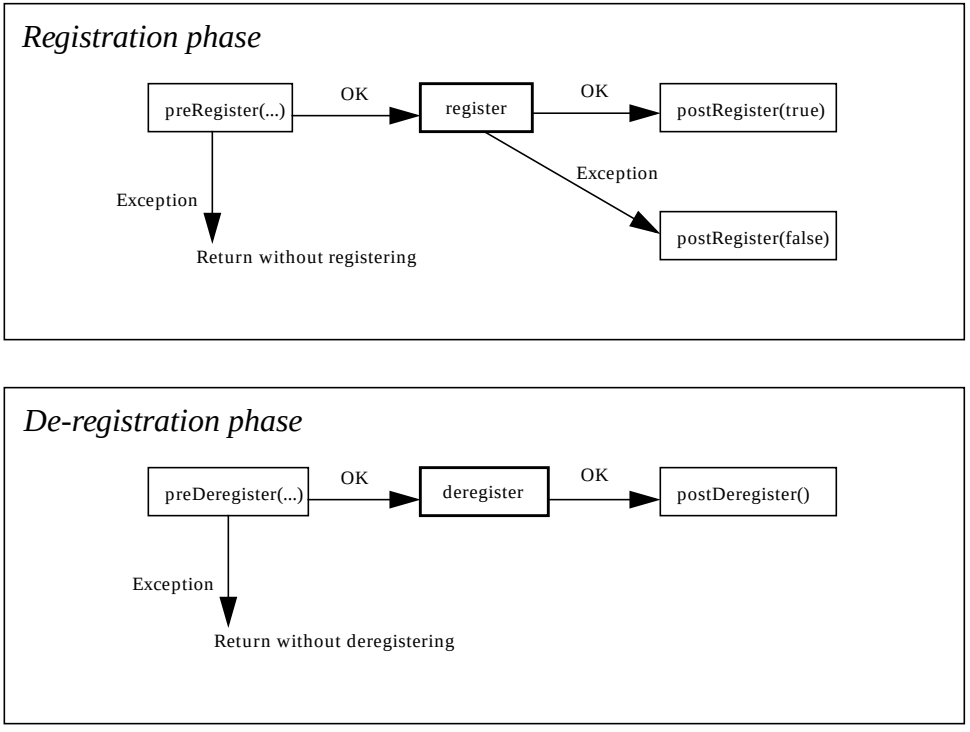


FIGURE 7-1 Calling Sequence for the MBean Registration Methods

Operations on MBeans

The methods of the `MBeanServer` interface define the following management operations to be performed on registered MBeans:

- Retrieve a specific MBean by its object name.
- Retrieve a collection of MBeans, by means of a pattern matching on their names, and optionally by means of a filter applied to their attribute values. Such a filter may be constructed by using the query expressions defined in “Queries” on page 122.
- Get one or several attribute value(s) of an MBean.
- Invoke an operation on an MBean.
- Discover the management interface of an MBean, that is, its attributes and operations. This is what is called the introspection of the MBean.

- Register interest in the notifications emitted by an MBean.

The methods of the MBean server are generic: they all take an object name which determines the MBean on which the operation is to be performed. The role of the MBean server is to resolve this object name reference, determine if the requested operation is allowed on the designated object, and if so, invoke the MBean method that performs the operation. If there is a result, the MBean server returns its value to the caller.

The detailed description of all MBean server operations is given in the Javadoc API.

MBean Server Delegate MBean

The MBean server defines a domain called "`JMImplementation`" in which one MBean of class `MBeanServerDelegate` is registered. This object identifies and describes the MBean server in which it is registered. It is also the broadcaster for notifications emitted by the MBean server. In other words, this MBean acts as a delegate for the MBean server which it represents.

The complete object name of this delegate object is specified by JMX, as follows: "`JMImplementation:type=MBeanServerDelegate`".

The delegate object provides the following information about the MBean server, all of which are exposed as read-only attributes of type `String`:

- The `MBeanServerId` identifies the agent. The format of this string is not specified, but it is intended to provide a unique id for the MBean server, for example, based on the hostname and a time stamp.
- The `SpecificationName` indicates the full name of the specification on which the MBean server implementation is based. The value of this attribute must be "`Java Management Extensions`".
- The `SpecificationVersion` indicates the version of the JMX specification on which the MBean server implementation is based. For this release, the value of this attribute must be "`1.0 Public Release 3`".
- The `SpecificationVendor` indicates the name of the vendor of the JMX specification on which the MBean server implementation is based. The value of this attribute must be "`Sun Microsystems`".
- The `ImplementationName` gives the implementation name of the MBean server. The format and contents of this string is given by the implementor.
- The `ImplementationVersion` gives the implementation version of the MBean server. The format and contents of this string is given by the implementor.
- The `ImplementationVendor` gives the vendor name of the MBean server implementation. The contents of this string is given by the implementor.

The `MBeanServerDelegate` class implements the `NotificationBroadcaster` interface and sends the `MBeanServerNotifications` that are emitted by the MBean server. Object wishing to receive these notifications must register with the delegate object (see "MBean Server Notifications" on page 121).

Note – The "`JMImplementation`" domain name is reserved for use by JMX Agent implementations. The `MBeanServerDelegate` MBean cannot be de-registered from the MBean server.

Remote Operations on MBeans

The remote interface for the MBean server is not specified in this phase of the JMX specification. It it presented here to provide a more complete view of the distributed services in the JMX architecture. The interface shown here is only a conceptual example.

Using an appropriate connector server in the agent, a remote management application will be able to perform operations on the MBeans through the corresponding connector client, once a connection is established.

FIGURE 7-2 shows how a management operation can be propagated from a remote management application to the MBean on the agent side. The example illustrates the propagation of a method for getting the "State" attribute of a standard MBean, in the following cases:

- The management application invokes a generic `getValue` method on the connector client which acts as a remote representation of the MBean server. This type of dynamic invocation is typically used in conjunction with the MBean introspection functionality which dynamically discovers the management interface of an MBean, even from a remote application.
- The management application invokes the `getState` method directly on a proxy object which is typically generated automatically from the MBean class (in the case of a Java application). The proxy object relies on the interface of the connector client to transmit the request to the agent and ultimately to the MBean. The response follows the inverse return path.

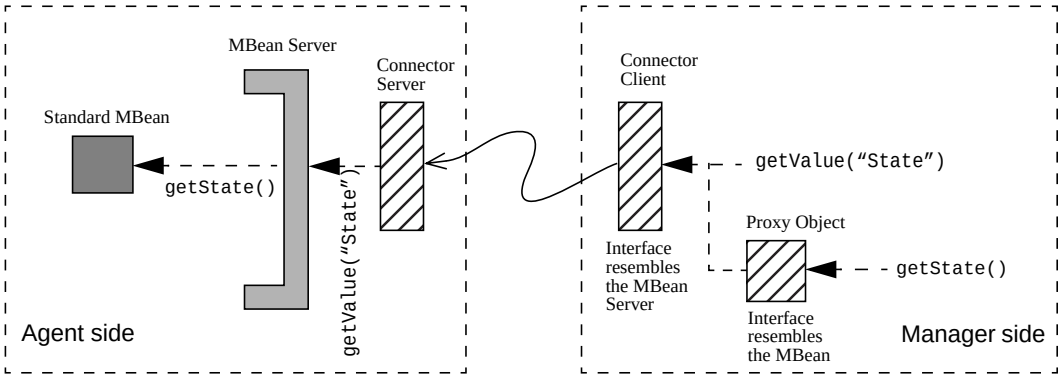


FIGURE 7-2 Propagation of a Remote Operation to an MBean.

MBean Server Notifications

The MBean server will always emit notifications when MBeans are registered or deregistered. A specific subclass of the `Notification` class is defined for this purpose: the `MBeanServerNotification` class, which contains a list of object names involved in the operation.

The MBean server object doesn't broadcast notifications itself: its unique delegate MBean implements the `NotificationBroadcaster` interface to actually broadcast the notifications in its place.

To register for MBean server notifications, the listener will call the `addNotificationListener` method of the MBean server, as when registering for MBean notifications, but it will provide the standardized object name of the MBean server delegate object (see "MBean Server Delegate MBean" on page 119).

As when receiving MBean notifications, an object interested in receiving MBean server notifications must implement the `NotificationListener` interface.

Through its delegate, the MBean server emits the following two types of notifications:

- `jmx.mbean.created`
  This notifications informs of the registration of one or several MBeans. The notification will convey the list of object names of these MBeans.
- `jmx.mbean.deleted`
  This notifications informs of the de-registration of one or several MBeans. The notification will convey the list of object names of these MBeans.

Note – The MBean server does not send notifications when attributes of registered MBeans change values. When implemented, this type of notification is handled directly by the MBean, as described in "Attribute Change Notifications" on page 51.

Queries

Queries retrieve sets of MBeans from the MBean server, according to their object name, their current attribute values, or both. The JMX specification defines the classes that are used to build query expressions. These objects are then passed to methods of the `MBeanServer` interface to perform the query.

The query mechanism described in this chapter can be used both by agent-side objects and by manager applications. However, since managers rely on protocol adaptors or connectors that are not yet defined, only agent-side queries are covered in this version of the specification. As with most other functionality, it is intended that the remote MBean server interface be identical, relying on the same classes and allowing managers to perform queries in an identical fashion.

The methods of the `MBeanServer` interface that perform queries are:

- `queryMBeans(ObjectName name, QueryExp query)` - Returns a `Set` containing object instances (object name and class name pairs) for MBeans matching the name and query.
- `queryNames(ObjectName name, QueryExp query)` - Returns a `Set` containing object names for MBeans matching the name and query.

The meaning of the parameters is the same for both methods. The object name parameter defines a pattern: the *scope* of the query is the set of MBeans whose object name satisfies this pattern. The query expression is the user-defined criteria for filtering MBeans within the scope, based on their attribute values. If either query method finds no MBeans in the given scope and/or satisfying the given query expression, the returned `Set` will contain no elements.

When the object name pattern is `null`, the scope is equivalent to all MBeans in the MBean server. When the query expression is `null`, MBeans are not filtered and the result is equivalent to the scope. When both parameters are `null`, the result is the set of all MBeans registered in the MBean server.

The set of all MBeans registered in the MBean server always includes the delegate MBean, as does any count of the registered MBeans. Other queries may also return the delegate MBean if its object name is within the scope and/or if it satisfies the query expression (see "MBean Server Delegate MBean" on page 119).

Scope of a Query

The scope is defined by an object name pattern: see “Pattern Matching” on page 107. Only those MBeans whose object name match the pattern are considered in the query. The query expression then needs to be applied to each MBean in the scope to filter the final result of the query. If the query mechanism is properly implemented and the user gives a relevant object name pattern, the scope of the query can greatly reduce the execution time of the query.

It is possible for the pattern to be a complete object name, meaning that the scope of the query is a single MBean. In this case, the query is equivalent to testing the existence of a registered MBean with that name, or, if the query expression is not `null`, testing the attribute values of that MBean.

Query Expressions

A query expression is built up from constraints on attribute values (such as “equals” and “less-than” for numeric values and “matches” for strings). These constraints may then be associated by relational operators (“and”, “or”, and “not”) to form complex expressions involving several attributes of an MBean.

For example, the agent or the manager should be able to express a query such as: “Retrieve the MBeans for which the attribute `age` is at least `20` and the attribute `name` starts with `G` and ends with `ling`”.

A query expression is evaluated on a single MBean at a time, and if and only if the expression is true, that MBean is included the query result. The MBean server tests the expression individually for every MBean in the scope of the query. It is not possible for a query expression to apply to more than one MBean: there is no mechanism for defining cross-MBean constraints.

The following classes and interfaces are defined for developing query expressions:

- The `QueryExp` interface identifies objects that are complete query expressions. These objects can be used in a query or composed to form more complex queries.
- The `ValueExp` and `StringValueExp` interfaces identify objects that represent numeric and string values, respectively, for placing constraints on attribute values.
- The `AttributeValueExp` interface identifies objects that represent the attribute involved in a constraint.
- The `Query` class supports the construction of the query. It contains static methods that return the appropriate `QueryExp` and `ValueExp` objects.

In practice, users do not instantiate the `ValueExp` and `QueryExp` implementation classes directly. Instead, they should rely on the methods of the `Query` class to return the values and expressions, composing them together to form the final query expression.

Methods of the `Query` Class

The static methods of the `Query` class are used to construct the values, constraints, and subexpressions of a query expression.

The following methods return a `ValueExp` instance that may be used as part of a constraint, as described:

- `classattr` - The result represents the class name of the MBean and may only be used in a string constraint.
- `attr` - The result represents the value of the named attribute. This result may be used in boolean, numeric or string constraints, depending upon the type of the attribute. Attributes may also be constrained by the values of other attributes of an equivalent type. This method is overloaded to also take a class name: this is equivalent to also setting a constraint on the name of the MBean's class.
- `value` - The result represents the value of the method's argument, and it is used in a constraint. This method is overloaded to take any one of the following types:
  - `java.lang.String`
  - `java.lang.Number`
  - `int`
  - `long`
  - `float`
  - `double`
  - `boolean`

  In all of these cases, the resulting value must be used in a constraint on an equivalent attribute value.
- `plus`, `minus`, `times`, `div` - These methods each take two `ValueExp` arguments and return a `ValueExp` object that represents the result of the operation. These operations only apply to numeric values. These methods are useful for constructing constraints between two attributes of a same MBean.

The following methods represent a constraint on one or more values. They take `ValueExp` objects and return a `QueryExp` object that indicates if the constraint is satisfied at runtime. This return object can be used as the query expression, or it can be composed into a more complex expression using the logical operators.

- `gt`, `geq`, `lt`, `leq`, `eq` - These methods represent the standard relational operators between two numeric values, respectively: greater than, greater than or equals, less than, less than or equals, and equals. The constraint is satisfied if the relation is true with the arguments in the given order.

- `between` - This method represents the constraint where the first argument is strictly within the range defined by the other two arguments. All arguments must be numeric values.
- `in` - This method is equivalent to multiple “equals” constraints between a numeric value argument and an array of numeric values. The constraint is satisfied (`true`) if the numeric value is equal to any one of the array elements.
- `match` - This method represents the equality between an attribute’s value and a given string value or string pattern. The pattern admits wildcards (`*` and `?`), character sets (`[Aa]`), and character ranges (`[A-Z]`) with the standard meaning. The attribute must have a string value, and the constraint is satisfied if it matches the pattern.
- `initialSubString`, `finalSubString`, `anySubString` - These methods represent substring constraints between an attribute’s value and a given substring value. The constraint is satisfied if the substring is a prefix, suffix or any substring of the attribute string value, respectively.

A constraint can be seen as computing a boolean value and can be used as a subexpression to the following methods. They also return a `QueryExp` object that can either be used in a query or as a subexpression of an even more complex query using the same methods:

- `and` - The resulting expression is the logical AND of the two subexpression arguments.
- `or` - The resulting expression is the logical OR of the two subexpression arguments.
- `not` - The resulting expression is the logical negation of the single subexpression argument.

Query Expression Examples

Using these methods, the sample query mentioned at the beginning of this section will be built as follows. When constructing constraints on string values, the asterisk (*) is a wildcard character which can replace any number of characters, including zero. Alternatively, the programmer may use the substring matching methods of the `Query` class.

CODE EXAMPLE 7-1 Building a Query

```
QueryExp exp = Query.and(
    Query.geq(Query.attr("age"),
              Query.value(20)),
    Query.match(Query.attr("name"),
                Query.value("G*ling")));
```

Most queries follow the above pattern: the named attributes of an MBean are constrained by programmer-defined values and then composed into a query across several attributes. All exposed attributes can be used for filtering purposes, provided that they may be constrained by numeric, boolean or string values.

It is also possible to perform a query based on the name of the Java class that implements the MBean, using the `classattr` method of the Query class. CODE EXAMPLE 7-2 shows how to build a query for filtering all MBeans of the fictional class `managed.device.Printer`. This constraint may also be composed with constraints on the attribute values to form a more selective query expression.

CODE EXAMPLE 7-2 Building a Query Based on the MBean Class

```
QueryExp exp = Query.eq(
    Query.classattr(),
    Query.value("managed.device.Printer"));
```

Query Exceptions

Constructing or performing queries can result in some exceptions which are specific to the filtering methods.

`BadAttributeValueExpException` Class

The `BadAttributeValueExpException` is thrown when an invalid name for an MBean attribute is passed to a query constructing method.

`BadStringOperationException` Class

This exception is thrown when an invalid string operation is passed to a method for constructing a query.

`BadBinaryOpValueExpException` Class

This exception is thrown when an invalid expression is passed to a method for constructing a query.

`InvalidApplicationException` Class

This exception is thrown when an attempt is made to apply a constraint with class name to an MBean of the wrong class.

CHAPTER 8

Advanced Dynamic Loading

This chapter describes the dynamic loading services which provide the ability to retrieve and instantiate MBeans using new Java classes and possibly native libraries from a remote server.

Dynamic loading is performed by the *m-let* service which is used to instantiate MBeans obtained from a remote URL (Universal Resource Locator) on the network. M-let is an abbreviation for *management applet*.

This is a mandatory JMX component for all compliant JMX agents.

Overview

The *m-let service* allows you to instantiate and register in the MBean server, one or several MBeans coming from a remote URL. The m-let service does this by loading an *m-let text file*, which specifies information on the MBeans to be obtained. The information on each MBean is specified in an XML-like tag, called the `MLET` tag. The location of the m-let text file is specified by a URL. When an m-let text file is loaded, all classes specified in MLET tags are downloaded, and an instance of each MBean specified in the file is created and registered.

The m-let service is itself implemented as an MBean and registered in the MBean server, so it can be used either by other MBeans, by the agent application, or by remote management applications.

The operation of the m-let service is illustrated in FIGURE 8-1.

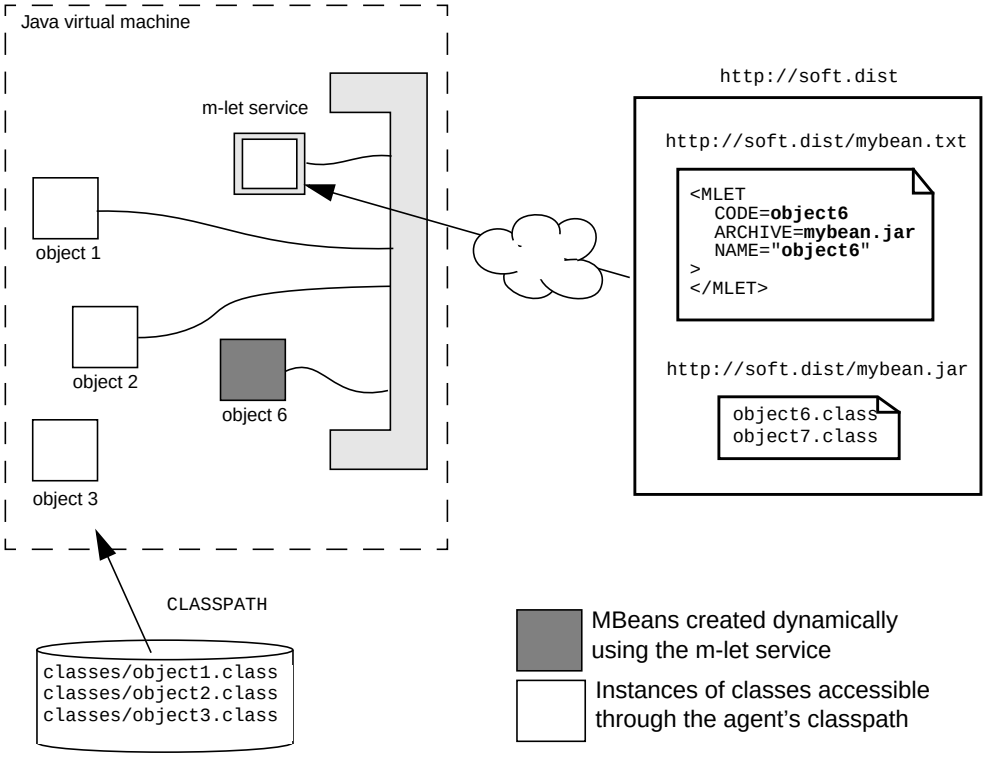


FIGURE 8-1 Operation of the M-Let Service

The MLET Tag

The m-let file may contain any number of MLET tags, each for instantiating a different MBean in a JMX agent. The MLET tag has the following syntax:

```
<MLET
    CODE = class | OBJECT = serfile
    ARCHIVE = "archivelist"
    [CODEBASE = codebaseURL]
    [NAME = MBeanName]
    [VERSION = version]
>
```

```
    [arglist]
</MLET>
```

The elements of this tag are explained below:

- `CODE = class`

  This attribute specifies the full Java class name, including package name, of the MBean to be obtained. The compiled `.class` file of the MBean must be contained in one of the JAR files specified by the `ARCHIVE` attribute. Either the `CODE` or the `OBJECT` attribute must be present.

- `OBJECT = serfile`

  This attribute specifies the `.ser` file that contains a serialized representation of the MBean to be obtained. This file must be contained in one of the JAR files specified by the `ARCHIVE` attribute. If the JAR file contains a directory hierarchy, this attribute must specify the path of the file within this hierarchy, otherwise a match will not be found.

- `ARCHIVE = archiveList`

  This mandatory attribute specifies one or more JAR files containing MBeans or other resources used by the MBean to be obtained. One of the JAR files must contain the file specified by the `CODE` or `OBJECT` attribute. If archive list contains more than one file:

  - Each file must be separated from the one that follows it by a comma (`,`)
  - The whole list must be enclosed in double quote marks (`""`)

  All JAR files in the archive list must be stored in the directory specified by the code base URL, or in the same directory as the m-let file which is the default code base when none is given.

- `CODEBASE = codebaseURL`

  This optional attribute specifies the code base URL of the MBean to be obtained. It identifies the directory that contains the JAR files specified by the ARCHIVE attribute. Specify this attribute only if the JAR files are not in the same directory as the m-let text file. If this attribute is not specified, the base URL of the m-let text file is used.

- `NAME = MBeanName`

  This optional attribute specifies the string format of an object name to be assigned to the MBean instance when the m-let service registers it in the MBean server.

- `VERSION = version`

  This optional attribute specifies the version number of the MBean and associated JAR files to be obtained.

  This version number can be used to specify whether or not the JAR files need to be loaded from the server to update those already loaded by the m-let service. The `version` must be a series of non-negative decimal integers each separated by a dot (`.`), for example `2.14`.

- `arglist`

  The optional contents of the MLET tag specify a list of one or more arguments to pass to the constructor of the MBean to be instantiated. The m-let service will look for a constructor with a signature that matches the types of the arguments specified in the `arglist`. Instantiating objects with a constructor other than the default constructor is limited to constructor arguments for which there is a string representation.

  Each item in the `arglist` corresponds to an argument in the constructor. Use the following syntax to specify the `argList`:

  ```
  <ARG TYPE=argumentType VALUE=argumentValue>
  ```

  where:

  - `argumentType` is the class of the argument (for example `Integer`)
  - `argumentValue` is the string representation of the value of the argument

The M-Let Service

The classes of the m-let service are members of the `javax.management.loading` package. The `MLet` class implements the `MLetMBean`, which contains the methods exposed for remote access. This implies that the m-let service is itself an MBean and may be managed as such.

The `MLet` class also extends the `java.net.URLClassLoader` object, meaning that it is itself a class loader. This allows several shortcuts for loading classes without requiring an m-let file.

Loading MBeans from a URL

The `getMBeansFromURL` methods of the m-let service perform the class loading based on the m-let text file on a remote server. The m-let file and the class files need to be available on the server as described in “The MLET Tag” on page 128. The two overloaded versions of this method take the URL argument as a string or as a `java.net.URL` object.

Each MLET tag in the m-let file describes one MBean to be downloaded and created in the MBean server. When the call to a `getMBeansFromURL` method is successful, the newly downloaded MBeans are instantiated in the JMX agent and registered with the MBean server. The methods return the object instance of the MBeans that were successfully created and a throwable object for those that weren’t.

Other methods of the MLet class manage the directory for native libraries downloaded in JAR files and used by certain MBeans. See the Javadoc API for more details.

Class Loader Functionality

The m-let service uses its class loader functionality to access the code base given in an m-let file or given by the URL itself. This code base is then available in the m-let service for downloading other MBeans from the same code base.

For example, an m-let file may specify a number of MLET tags in order to populate all of the MBeans in a JMX agent. Once the `getMBeansFromURL` method has been invoked to do this, the m-let service can be used to instantiate any one of those MBeans again, or any other class at the same code base.

This is done by passing the m-let service's object name as a class loader parameter to the `createMBean` method of the MBean server (see the corresponding Javadoc API). Since the code base has already been accessed by the m-let service, its class loader functionality can access the code base again. In this case, the information in the MLET tag is no longer taken into account, although the parameters of the `createMBean` method can be used to specify the parameters to the class constructor.

Since the `createMBean` methods of the `MBeanServer` interface take the object name of the class loader, this functionality is also available to remote management applications which do not have direct object references in the JMX agent.

The m-let service MBean also exposes the `addURL` methods for specifying a code base without needing to access any m-let file. These methods add the code base designated by the given URL to the class loader of the m-let service. MBean classes at this code base can be downloaded and created in the MBean server directly through the `createMBean` method, again with the m-let service given as the class loader object.

Note – Using the class loader of the m-let service to load create classes from arbitrary code bases or reload classes from m-let code bases implies that the agent application or the MBean developer has some prior knowledge of the code base contents at runtime.

CHAPTER 9

Monitoring

This chapter specifies the family of monitor MBeans which allow you to observe the variation over time of attribute values in other MBeans and emit notifications at threshold events. As a whole they are referred to as the monitoring services.

Monitoring services are a mandatory part of JMX-compliant agents, and they must be implemented in full.

Overview

Using a monitoring service, the *observed attribute* of another MBean (the *observed* MBean) is monitored at intervals specified by the *granularity period*. The type of the observed attribute is one of the types supported by the specific monitor subclass which is used. The monitor derives a value from this observation, called the *derived gauge*. This derived gauge is either the exact value of the observed attribute, or optionally, the difference between two consecutive observed values of a numeric attribute.

A specific notification type is sent by each of the monitoring services when the value of the derived gauge satisfies one of a set conditions. The conditions are specified when the monitor is initialized, or dynamically through the monitor MBean's management interface. Monitors may also send notifications when certain error cases are encountered during the observation.

Types of Monitors

Information on the value of an attribute within an MBean may be provided by three different types of monitors:

- `CounterMonitor` - Observes attributes with integer types (`Byte`, `Integer`, `Short`, `Long`) which behave like a counter; that is:
  - Their value is always positive or null
  - They can only be incremented
  - They may wrap, and in that case a *modulus* value is defined
- `GaugeMonitor` - Observes attributes with integer or floating point types (`Float`, `Double`) which behave as a gauge (arbitrarily increasing and decreasing).
- `StringMonitor` - Observes an attribute of type `String`.

All types of monitors extend the abstract `Monitor` class, which defines common attributes and operations.

Each of the monitors is also a standard MBean, allowing them to be created and configured dynamically by other MBeans or by management applications.

`MonitorNotification` Class

A specific subclass of the `Notification` class is defined for use by all monitoring services: the `MonitorNotification` class.

This notification is used to report one of the following cases:

- One of the conditions explicitly monitored is detected, for example, the high threshold of a gauge is reached.
- An error occurs during an observation of the attribute, for example, the observed object is no longer registered.

The notification type string within a `MonitorNotification` instance identifies the specific monitor event or error condition, as shown in FIGURE 9-1. The fields of a `MonitorNotification` instance contain the following information:

- The observed object name
- The observed attribute name
- The derived gauge, that is, the last value computed from the observation
- The threshold value or string which triggered this notification

The tree representation of all notification types which may be generated by the monitoring services is given in FIGURE 9-1. None of the monitor MBeans generate all of these types: the error types are common to all and described below, but each of the threshold events is particular to its monitor and described in the corresponding section.

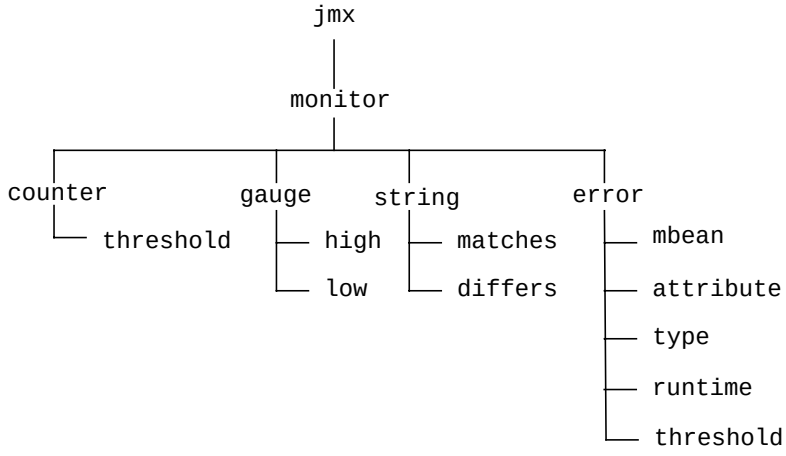


FIGURE 9-1 Tree Representation of Monitor Notification Types

Common Monitor Notification Types

The following notifications types are common to all monitors; they are emitted to reflect error cases:

- `jmx.monitor.error.mbean` - Sent when the observed MBean is not registered in the MBean server. The *observed object name* is provided in the notification.
- `jmx.monitor.error.attribute` - Sent when the observed attribute does not exist in the observed object. The *observed object name* and *observed attribute name* are provided in the notification.
- `jmx.monitor.error.type` - Sent when the observed attribute is not of the appropriate type for the given monitor. The *observed object name* and *observed attribute name* are provided in the notification.
- `jmx.monitor.error.runtime` - All exceptions (except the cases described above) that occur while trying to get the value of the observed attribute are caught by the monitor and will be reported in a notification of this type.

The following notification type is common to the counter and the gauge monitors; it is emitted to reflect specific error cases:

- `jmx.monitor.error.threshold` - Sent in case of any incoherence in the configuration of the monitor parameters:
  - Counter monitor: the threshold, the offset, or the modulus is not of the same type as the observed counter attribute.
  - Gauge monitor: the low threshold or high threshold is not of the same type as the observed gauge attribute.

`CounterMonitor` Class

A counter monitor sends a notification when the value of the observed counter attribute reaches or exceeds a *threshold* known as the comparison level. In addition, an *offset* mechanism enables particular counting intervals to be detected, as follows:

- If the offset value is not zero, whenever the threshold is triggered by the counter value reaching a comparison level, that comparison level is incremented by the offset value. This is regarded as taking place instantaneously, that is, before the count is incremented. Thus, for each level, the threshold triggers an event notification every time the count increases by an interval equal to the offset value.
- If the counter we are monitoring wraps around when it reaches its maximum value then the modulus value needs to be set to that maximum value. The modulus is the value at which the counter is reset to zero.
- If the counter difference option is used, then the value of the derived gauge is calculated as the difference between the observed counter values for two successive observations. If this difference is negative then the value of the derived gauge is incremented by the value of the modulus.

  The derived gauge value (`V[t]`) for the counter difference is calculated at time `t` using the following method, where `GP` is the granularity period:

  - if `(counter[t] - counter[t-GP])` is positive then
    `V[t] = counter[t] - counter[t-GP]`
  - if `(counter[t] - counter[t-GP])` is negative then
    `V[t] = counter[t] - counter[t-GP] + MODULUS`

The counter monitor has the following constraint:

- The threshold value, the offset value and the modulus value properties must be of the same type as the observed attribute.

The operation of a counter monitor example with an offset of 2 is illustrated in FIGURE 9-2.

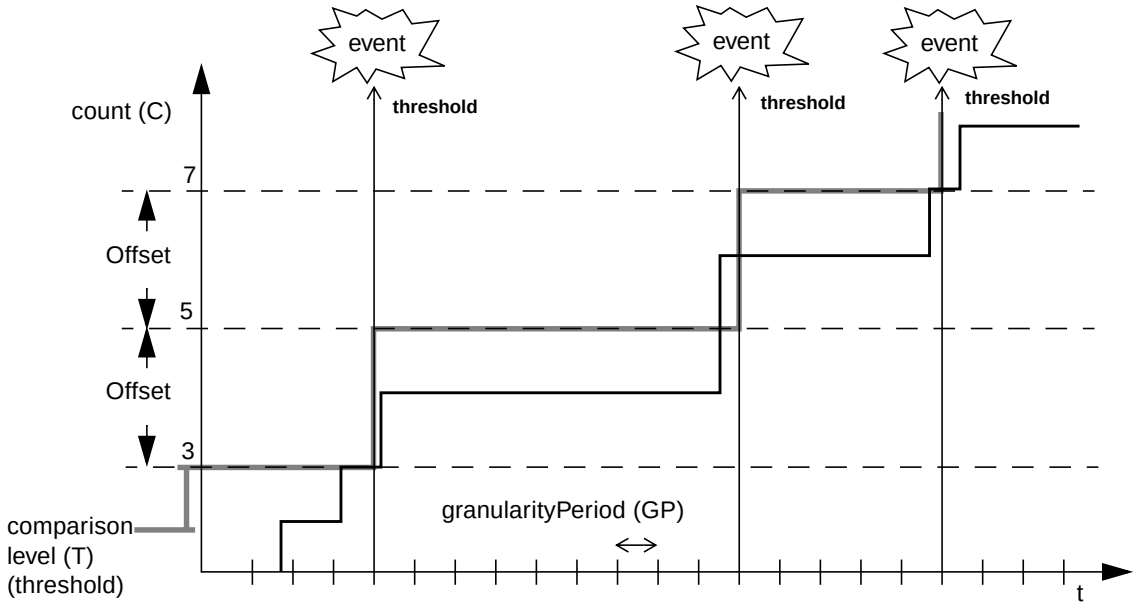


FIGURE 9-2 Operation of the `CounterMonitor`

The monitor observes a counter `C(t)` which varies with time `t`. The granularity period is `GP` and the comparison level is `T`. A `CounterMonitor` sends a notification when the value of the counter reaches or exceeds the comparison (threshold) level. After the notification has been sent, the comparison level is incremented by the offset value until the comparison level is greater than the current value of the counter.

Counter Monitor Notification Types

In addition to the monitor error notification types, a `CounterMonitor` MBean may broadcast the following notification type:

- `jmx.monitor.counter.threshold`
  This notification type is triggered when the derived gauge has reached or exceeded the threshold value.

GaugeMonitor Class

A gauge monitor observes the value of a numerical attribute which behaves as a gauge. A *hysteresis* mechanism is provided to avoid the repeated triggering of notifications when the gauge makes small oscillations around the threshold value. This capability is provided by specifying threshold values in pairs; one being a *high threshold* value and the other being a *low threshold* value. The difference between threshold values is the hysteresis interval.

The `GaugeMonitor` MBean has the following structure:

- The `HighThreshold` attribute defines the value that the gauge must reach or exceed in order to trigger a notification which will be broadcast only if the `NotifyHigh` boolean attribute is `true`.
- The `LowThreshold` attribute defines the value that the gauge must fall to or below in order to trigger a notification which will be broadcast only if the `NotifyLow` boolean attribute is set to `true`.

The gauge monitor has the following constraints:

- The threshold high value and the threshold low value properties are of the same type as the observed attribute.
- The threshold high value is greater than or equal to the threshold low value.

The gauge monitor has the following behavior:

- Initially, if `NotifyHigh` is `true` and the gauge value becomes equal to or greater than the `HighThreshold` value while the gauge is increasing, then the defined notification is triggered; subsequent crossings of the high threshold value will not trigger further notifications until the gauge value becomes equal to or less than the `LowThreshold` value.
- Initially, if `NotifyLow` is `true` and the gauge value becomes equal to or less than the `LowThreshold` value while the gauge is decreasing, then the defined notification is triggered; subsequent crossings of the low threshold value will not cause further notifications until the gauge value becomes equal to or greater than the `HighThreshold` value.

If the gauge difference option is used, then the value of the derived gauge is calculated as the difference between the observed gauge values for two successive observations.

In this case, the derived gauge value (`V[t]`) is calculated at time `t` using the following equation, where `GP` is the granularity period:

```
V[t] = gauge[t] - gauge[t-GP]
```

The operation of the `GaugeMonitor` is illustrated in FIGURE 9-3, assuming both notification switches are `true`.

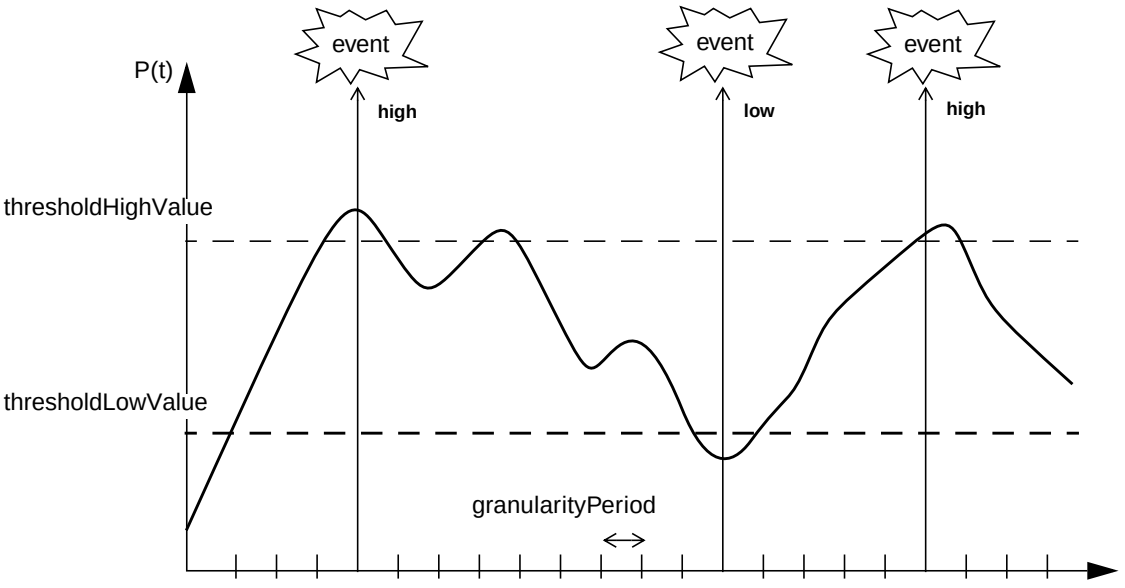


FIGURE 9-3 Operation of the `GaugeMonitor`

Gauge Monitor Notification Types

In addition to the monitor error notification types, a `GaugeMonitor` MBean may broadcast the following notification types:

- `jmx.monitor.gauge.high`
  This notification type is triggered when the derived gauge has reached or exceeded the high threshold value.
- `jmx.monitor.gauge.low`
  This notification type is triggered when the derived gauge has decreased to or below the low threshold value.

`StringMonitor` Class

A string monitor observes the value an attribute of type `String`. The derived gauge in this case is exactly the value of the observed attribute. The string monitor is configured with a value for the string called *string-to-compare*, and is able to detect the following two conditions:

- The derived gauge matches the string-to-compare. If the `NotifyMatch` attribute of the monitor is `true`, then a notification is sent. At the following observation times (defined by the granularity period), no other notification will be sent as long as the attribute value matches the string-to-compare.
- The value of the derived gauge differs from the string-to-compare. If the `NotifyDiffer` attribute of the monitor is `true`, then a notification is sent. At the following observation points, no other notification will be sent, for as long as the attribute value differs from the string-to-compare.

Assuming both notifications are selected, this mechanism ensures that matches and differs are strictly alternating, each occurring the first time the condition is observed.

The operation of the string monitor is illustrated in FIGURE 9-4. The granularity period is `GP`, and the string-to-compare is “XYZ”.

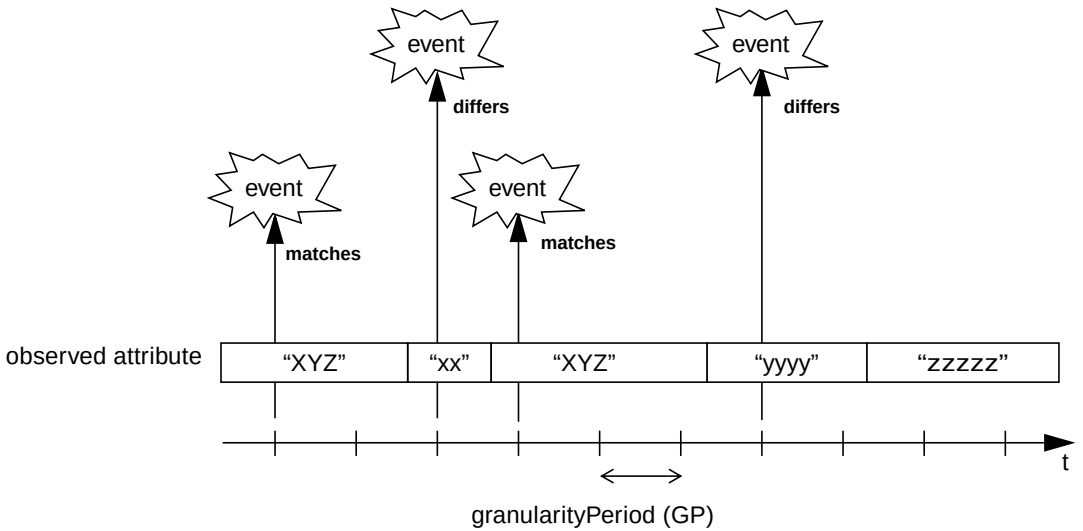


FIGURE 9-4 Operation of the `StringMonitor`

String Monitor Notification Types

In addition to the monitor error notification types, a `StringMonitor` MBean may broadcast the following notification types:

- `jmx.monitor.string.matches`
  This notification type is triggered when the derived gauge first matches the string to compare.
- `jmx.monitor.string.differs`
  This notification type is triggered when the derived gauge first differs from the string to compare.

Implementation of the Monitor MBeans

FIGURE 9-5 provides the class diagram of the various monitor MBeans, with the interfaces they implement. The Javadoc API provides the complete description of all monitoring service interfaces and classes.

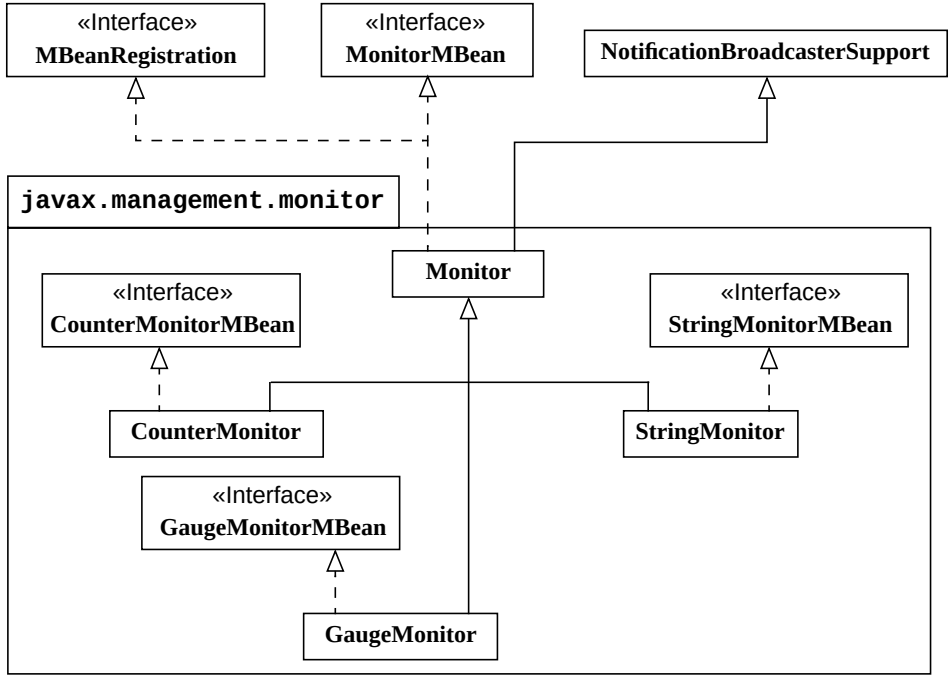


FIGURE 9-5 The Package and Class Diagram of the Monitor MBeans

CHAPTER 10

Timer Service

The timer service triggers notifications at specific dates and times; it may also trigger notifications repeatedly at a constant interval. The notifications are sent to all objects registered to receive notifications emitted by the timer. The timer service is an MBean that can be managed, allowing applications to set up a configurable scheduler.

Conceptually, the `Timer` class manages a list of dated notifications that are sent when their date and time arrives. Methods of this class are provided to add and remove notifications from the list. In fact, the notification *type* is provided by the user, along with the date and optionally a period and the number of repetitions. The timer service always sends the notification instances of its specific `TimerNotification` class.

Timer Notifications

The timer service can manage notifications in two different ways:

- Notifications that are triggered only once
- Notifications that are repeated with a defined period and/or number of occurrences

This behavior is defined by the parameters passed to the timer when the notification is added into the list of notifications. Each of the notifications that is added to the timer service is assigned a unique identifier number. Only one identifier number is assigned to a notification, no matter how many times it is triggered.

`TimerNotification` Class

A specific subclass of the `Notification` class is defined for use by the timer MBeans: the `TimerNotification` class. The notification type contained in instances of the `TimerNotification` class is particular: it is defined by the user when the notification is added to the timer.

The `TimerNotification` class has a notification identifier field that uniquely identifies the timer notification which triggered this notification instance.

Adding Notifications to the Timer

The timer service maintains an internal list of the dated notifications that it has been asked to send. Notifications are added to this list using the `Timer` class' `addNotification` methods. This method can take the following parameters. These parameters are used by the timer to create a `TimerNotification` object and then add it to the list:

- `type` - The notification type string.
- `message` - The notification's detailed message string.
- `usedData` - The notification's user data object.
- `date` - The date when the notification will occur.

The `addNotification` method is overloaded and, in addition to the notification's parameters and date, it can take the following optional parameters:

- `period` - The interval in milliseconds between notification occurrences. Repeating notifications are not enabled if this parameter is zero or `null`.
- `nbOccurences` - The total number of times that the notification will occur. If the value of *nbOccurences* is zero or is not defined (`null`), and if the `period` is not zero or `null`, then the notification will repeat indefinitely.

If the notification to be inserted has a date that is before the current date, the `addNotification` method attempts to update this entry as follows:

- If the notification is added with a non-null period, the method updates the notification date by adding the defined period interval until the notification date is later than the current date. When the notification date is later than the current date, the method inserts the notification into the list of notifications.
- If a number of occurrences is specified with a periodicity, the method updates the notification date as explained above. The number of times that the method adds the period interval is limited by the specified number of occurrences which is decreased in proportion. If the notification date remains earlier than the current date, a standard Java `IllegalArgumentException` is thrown.

- If the notification is added with a null or zero-length period, the notification date cannot be updated to a valid date, and a standard Java `IllegalArgumentException` is thrown.

The `addNotification` method returns the identifier of the new timer notification. This identifier can be used to retrieve information about the notification from the timer or to remove the notification from the timer's list of notifications.

After a notification has been added to the list of notifications, its associated parameters cannot be updated.

Removing Notifications From the Timer

Timer notifications are removed from the list of notifications using the one of the following methods of the `Timer` class:

- `removeNotification` - Takes the notification identifier as parameter and removes the corresponding notification from the list.
- `removeNotifications` - Takes the notification type as parameter and removes all notifications from the list that were added with that type.

  If the specified notification identifier or type does not correspond to any notifications in the list, the methods throw an `InstanceNotFoundException`.
- `removeAllNotifications` - Empties the timer's list of notifications.

Starting and Stopping the Timer

The timer is started using the timer `start` method and stopped using the `stop` method. If the list of notifications is empty when the timer is started, the timer waits for a notification to be added. No timer notifications will be triggered before the timer is started or after it is stopped.

You can determine whether the timer is running or not by invoking the timer method `isActive`. The `isActive` method returns `true` if the timer is running.

If any of the notifications in the timer's list have associated dates that have passed when the timer is started, the timer attempts to update them. The dates of periodic notifications are incremented by their interval period until their dated is greater than the current date. The number of increments may be limited by their defined number of occurrences. Notifications with fixed dates preceding the start date and limited occurrence notifications which cannot be updated to exceed the start date are removed from the timer's list of notifications.

When a notification is updated or removed during timer start-up, its notification is either triggered or ignored, depending upon the `sendPastNotifications` attribute of the `Timer` class:

- `sendPastNotifications` = `true` - All notifications with a date before the start date are triggered; if the notification is periodic, its date is updated and the notification is sent.
- `sendPastNotifications` = `false` - Notifications with a date before the start date are ignored; if the notification is periodic, the notification date is updated but no notification is sent.

Setting the `sendPastNotifications` flag to `false` can be used to suppress a flood of notifications being sent out when the timer is started. The default value for this flag is `false`. Setting this flag to `true` insures that notification dates which occur when the timer is stopped are not lost. The user can choose to receive them when the timer is started again, even though they no longer correspond to their set dates.

CHAPTER 11

Relation Service

As part of the agent specification, the Java Management extensions also define a model for relations between MBeans. A relation is a user defined, n-ary association between MBeans in named roles. The JMX specification defines the classes that are used to construct an object representing a relation, and it defines the relation service which centralizes all operations on relations in an agent.

All relations are defined by a relation type that provides information about the roles it contains, such as their multiplicity, and the class name of MBeans which fulfill the role. Through the relation service, users create new types and then create, update, or remove relations that fulfill these types. The relation service also performs queries among all relations to find related MBeans.

The relation service maintains the consistency of relations, checking all operations and all MBean deregistrations to ensure that a relation always conforms to its relation type. If a relation is no longer valid, it is removed from the relation service, though its member MBeans continue to exist otherwise.

The Relation Model

A relation is composed of named *roles*, each of which has a value consisting of the list of MBeans in that role. This list must comply with the role information which defines the multiplicity and class of MBeans in the corresponding role. A set of one or more role information definitions constitutes 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.

Terminology

The JMX relation model relies on the following terms. Here we only define the concepts represented by a term, not the corresponding Java class.

role information Describes one of the roles in a relation. The role information gives the name of the role, its multiplicity expressed as a single range, the name of the class that fulfills this role, read-write permissions, and a description string.

relation type The metadata for a relation, composed of a set of role information. It describes the roles that a relation must fulfill, and it serves as a template for creating and maintaining relations.

relation A current association between the MBeans that fulfill a given relation type. A relation can only be created and modified such that the roles of its defined type are always respected. A relation may also have properties and methods that operate on its MBeans.

role value The list of MBeans that currently fulfill a given role in a relation. The role value must at all times conform to its corresponding role information.

unresolved role An unresolved role is the result of an illegal access operation on a role, compared to its role information. Instead of the resulting role value, the unresolved role contains the reason for the refused operation. For example, setting a role with the wrong class of MBean, providing a list with an illegal cardinality, or attempting to write a read-only role will all return an unresolved role.

support classes Internal classes used to represent relation types and relation instances. The support classes are also exposed to simplify user implementations of relation classes. The user's external implementation must still rely on the relation service to maintain the consistency of the relation model.

relation service An MBean which can access and maintain the consistency of all relation types and all relation instances within a JMX agent. It provides query operations to find related MBeans and their role in a relation. It is also the sole source of notifications concerning relations.

Example of a Relation

Throughout this chapter we will use the example of a relation between books and their owner.

In order to represent this relation in the JMX model, we say that `Books` and `Owner` are roles. `Books` represents any number of owned books of a given MBean class, and `Owner` is a 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.

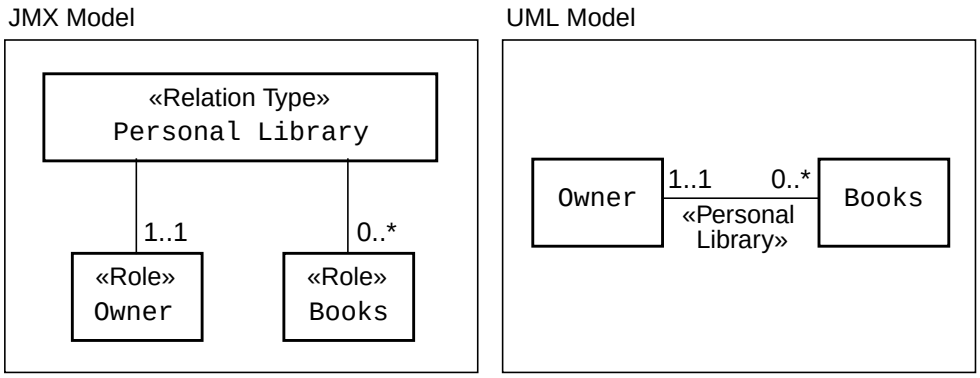


FIGURE 11-1 Comparison of the Relation Models

In the JMX model, the relation type is a static set of roles. Relation types may be defined at run-time, but once defined, their roles and the role information cannot be modified. The relation instance of a given type defines the MBeans in each role and provides operations on them, if necessary.

Maintaining Consistency

MBeans are related through relation instances defined by relation types in the relation service, but the MBeans remain completely accessible through the MBean server. Only registered MBeans, identified by their object name, may be members of a relation. The relation service never operates on member MBeans, it only provides their object names in response to queries.

The relation service blocks the creation of invalid relation types, for example if the role information is inconsistent. In the same way, invalid relations may not be created, either because the relation type is not respected or because the object name of a member MBean does not exist in the MBean server. The modification of a role value is also subject to the same consistency checks.

When a relation is removed from the relation service, its member MBeans are no longer related through the removed instance, but are otherwise unaffected. When a relation type is removed, all existing relations of that type are first removed. The caller is responsible for being aware of the consequences of removing a relation type.

Because relations are defined only between registered MBeans, deregistering a member MBean modifies the relation. The relation service listens for all MBean server notifications that indicate when a member of any relation is deregistered. The

corresponding MBean is then removed from any role value where it appears. If the new cardinality of the role is not consistent with the corresponding relation type, that relation is removed from the relation service.

The relation service sends a notification after all operations that modify a relation instance, either creation, update, or removal. This notification provides information about the modification, for example new role values, or the identifier of the relation. The notification also indicates whether the relation was internally or externally defined (see "External Relations" on page 152).

There is a slight difference between the two models presented in FIGURE 11-1 on page 149. The UML association implies that each one of the `Books` can only have one owner. Our relation type only models a set of roles, indicating that a relation instance has one `Owner` MBean and any number of MBeans in the `Books` role.

The JMX relation model only guarantees that an MBean fulfills its designated role, it does not allow a user to define how many relations in which an MBean may appear. Therefore, the relation service does not do inter-relation consistency checks, they are the responsibility of the management application when creating relation instances.

If it is needed, the designer of a management solution must implement this level of consistency in the applications that call upon the relation service. 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.

Implementation

The JMX specification defines the Java classes whose behavior implements this relation model. Each of the concepts defined in "Terminology" on page 148 has a corresponding Java class (see FIGURE 11-2 on page 151). Along with the behavior of the relation service object itself, these classes determine how the relation service is used in management solutions.

This section will explain the interaction between the relation service and the support classes. The operations and other details of all classes will be covered in further sections. The exception classes are all subclasses of the `RelationException` class and provide only a message string. The Javadoc API for the other classes indicates which exceptions are raised by specific operations.

In practice, role description structures are handled outside of the relation service, and their objects are instantiated directly by the user (see "Role Description Classes" on page 161). Role information objects are grouped into arrays to define a relation type. Role objects and role lists are instantiated to pass to setters of role values. Role results are returned by getters of role values, and their role lists and unresolved role lists can be extracted for processing.

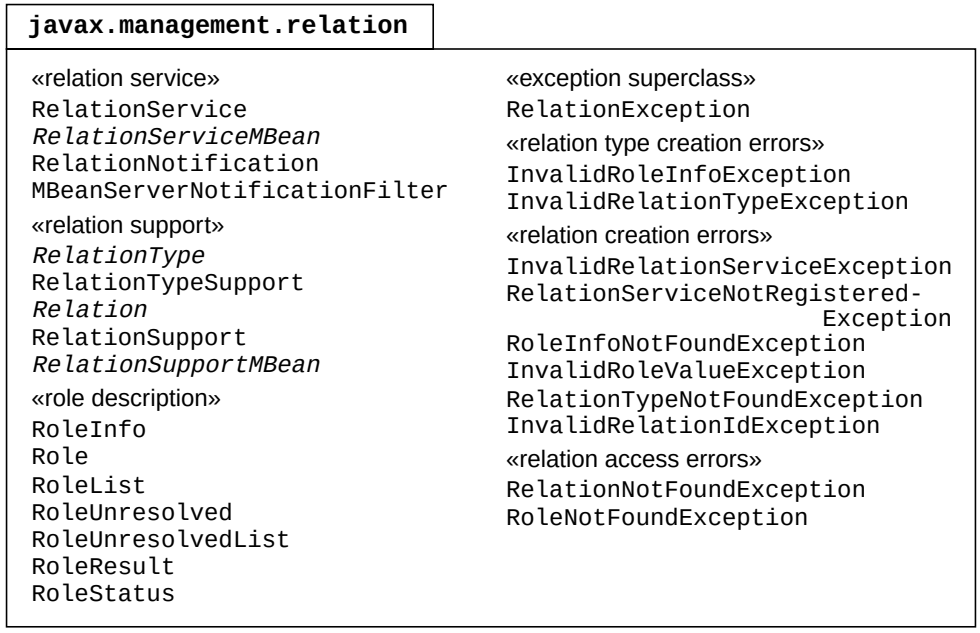


FIGURE 11-2 Classes of the `javax.management.relation` package

On the other hand, relation types and relation instances are controlled by the relations service in order to maintain the consistency of the relation model. The implementation of the JMX relation model provides a flexible design whereby relation types and instances can be either internal or external to the relation service.

Internal relation types and instances are created by the relation service and can only be accessed through its operations. The objects representing types and relations internally are not accessible to the user. External relation types and instances are objects instantiated outside of the relation service and added under its control. Users can access these object in any manner that has been designed into them, including as registered MBeans.

External Relation Types

The relation service maintains a list of relation types that are available for defining new relations. A relation type must be created internally or instantiated externally and added to the relation service before it can be used to define a relation.

Objects representing external relation types must implement the `RelationType` interface. The relation service relies on its methods to access the role information for each of the roles defined by the external object. See “RelationTypeSupport Class” on page 158 for the description of a class used to define external relation types.

Relation types are immutable, meaning that once they are added to the relation service, their role definitions cannot be modified. If an external relation type exposes methods for modifying the set of role information, they should not be invoked by its users after the instance has been added under the control of the relation service. The result of doing so is undefined, and consistency within the relation service is no longer guaranteed.

The benefit of using an external relation type class is that the role information may be defined statically in a class constructor, for example. This allows predefined types to be rapidly instantiated and then added to the relation service.

Once it has been added to the relation service, an external relation type can be used indifferently to create internal or external relations. An external relation type is also removed from the relation service in the same way as an internal relation type, with the same consequences (see “RelationService Class” on page 154)

External Relations

The relation service also maintains a list of the relations that it controls. Internal relations are created through the relation service and are only accessible through its methods. External relations are MBeans instantiated by the user and added under the control of the relation service. They must be registered in the MBean server before they can be added to the relation service. They are accessible both through the relation service and through the MBean server.

An external relation object must implement the `Relation` interface which defines the methods that the relation service will use to access its role values. An external relation is also responsible for maintaining its own consistency, by only allowing access to its role values as described by its relation type. Finally, an external relation must inform the relation service when any role values are modified.

The relation service object exposes methods for checking role information and updating its internal role values. The external relation object must be designed to call these when appropriate. Failure to do so will result in an inconsistent relation service whose behavior is thereafter undefined.

The major benefit of external relations is the ability to provide methods that return information about the relation’s members or even operate on the role values. Since the external relation is also an MBean, it may choose to expose these methods as attributes and operations.

For example, the book ownership relation may be represented by a unary relation type containing only the role `Books`. The relation would be implemented by instances of an `Owner` MBean that are external to the relation service. This MBean could have an attribute such as `bookCount` and operations such as `buy` and `sell` which all apply to the current members of the relation.

See “RelationSupport Class” on page 161 for an example of an external relation.

Relation Service Classes

The relation service is implemented in the `RelationService` object, a standard MBean defined by the `RelationServiceMBean` interface. It can therefore be accessed and managed remotely from a management application.

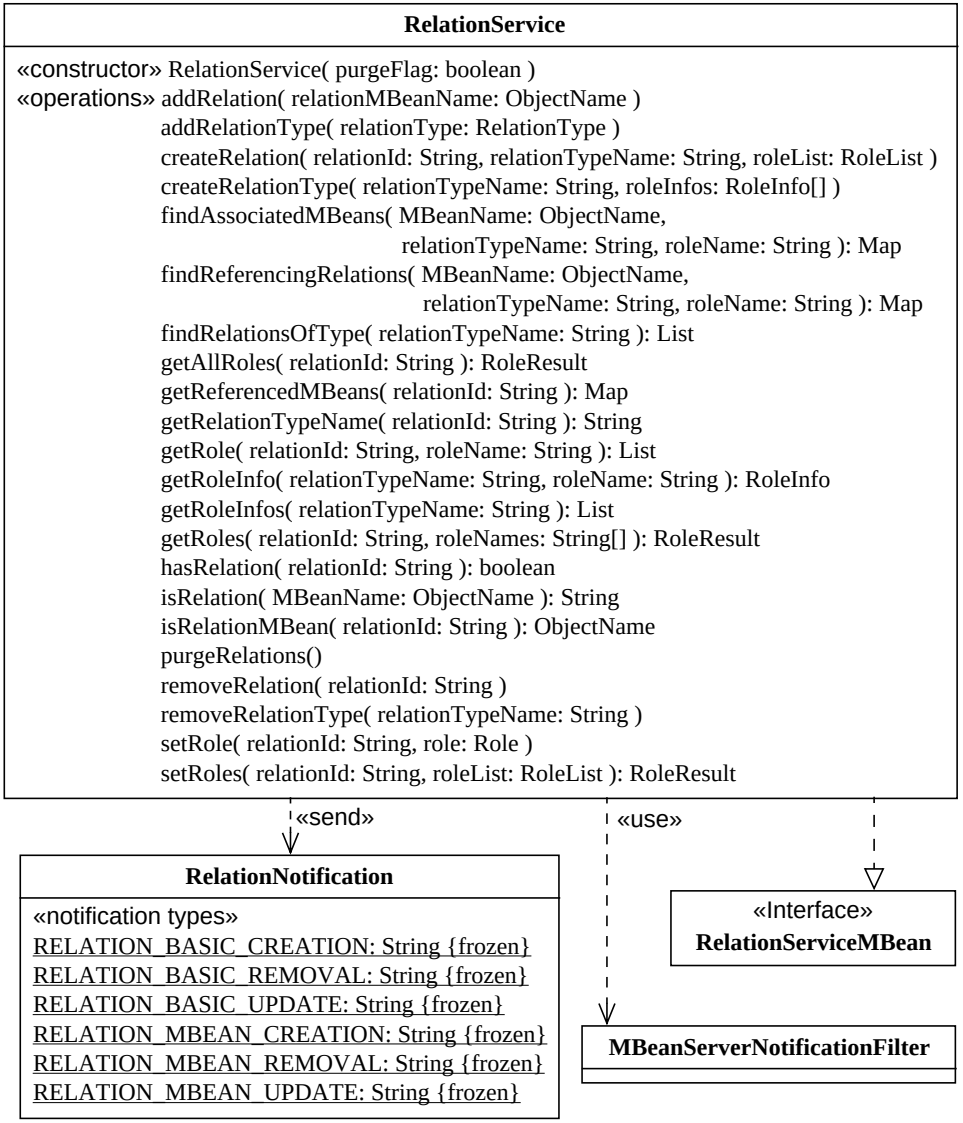


FIGURE 11-3 Relation Service Classes

The relation service MBean is a notification broadcaster and the only object to send `RelationNotification` objects. In order to maintain consistency, it also listens for MBean server notifications through an `MBeanServerNotificationFilter` object.

`RelationService` Class

The relation service exposes methods for creating and removing relation types and relation instances, and for accessing roles in relations. It also exposes methods for querying the relations and their members in order to find related MBeans.

There are two methods to define a relation type:

- `createRelationType` - Creates an internal relation type from an array of role information objects; the relation type will be identified by a name passed as a parameter and which must be unique among all relation type names.
- `addRelationType` - Makes an externally defined relation type available through the relation service (see "RelationType Interface" on page 158).

There are also two similar methods for defining a relation. Every new relation triggers a `RelationNotification`:

- `createRelation` - Creates an internal relation using the given list of role values; the relation will be identified by an identifier passed as a parameter and which must be unique among all relation identifiers.
- `addRelation` - Places an external relation represented by a MBean under the control of the relation service; the MBean must have been previously instantiated and registered in the MBean server.

The method `removeRelationType` removes both internal or external relation types. All relations of that type will be removed with the `removeRelation` method (see "Maintaining Consistency" on page 149).

The `removeRelation` method removes a relation from the relation service, meaning that it can no longer be accessed. Member MBeans in the roles of the relation continue to exist. When an external relation is removed, the MBean that implements it will still be available in the MBean server. Removing a relation triggers a relation notification.

The relation service provides methods to access a relation type, identified by its unique name: `getRoleInfo` and `getRoleInfos`.

It provides methods to access the relation and its role values. All access to roles is subject to the access mode defined in the relation type and to consistency checks, especially for setting role values: `getRelationTypeName`, `getRole`, `getRoles`, `getAllRoles`, `getReferencedMBeans`, `setRole` and `setRoles`. Setting roles will trigger a relation update notification.

There are also methods for identifying internal and external relations:

- `hasRelation` - Indicates if a given relation identifier is defined.
- `isRelation` - Takes an object name and indicates if it has been added as an external relation to the service.
- `isRelationMBean` - Returns the object name of an externally defined relation.

The following query methods retrieve relations where a given MBean is involved:

- `findReferencingRelations` - Retrieves the relations where a given MBean is referenced.
  It is possible to restrict the scope of the search by specifying the type of the relations to look for and/or the role where the MBean is expected to be found in the relation.
  In the result, relation identifiers are mapped to a list of role names where the MBean is referenced (as an MBean can be referenced in several roles of the same relation).
- `findAssociatedMBeans` - Retrieves the MBeans associated to a given MBean in the relation service.
  It is possible to restrict the scope of the search by specifying the type of the relations to look for and/or the role where the MBean is expected to be found in the relation.
  In the result, the object names of related MBeans are mapped to a list of relation identifiers where the two are associated.

The method `findRelationsOfType` returns the relation identifiers of all the relations of the given relation type.

To maintain the consistency, the relation service listens to the deregistration notifications from the MBean server delegate. It will be informed when an external relation's MBean is unregistered, in which case the relation is removed, or when an MBean that is a member of a relation is unregistered (see "Maintaining Consistency" on page 149). The `purgeRelations` method will check all relation data for consistency and remove all relations that are no longer valid.

Every time a relevant deregistration notification is received, the relation service behavior depends upon the purge flag attribute:

- If the purge flag is true, the `purgeRelations` method will be called automatically.
- When the purge flag is false, no action is taken and the relation service might be in an inconsistent state until the `purgeRelations` method is called by the user.

The relation service also exposes methods that allow external relation MBeans to implement the expected behavior, or to inform the relation service so that it can maintain the consistency:

- `checkRoleReading` and `checkRoleWriting` - Check if a given role can be read and updated by comparing the new value to the role information.

- `sendRelationRemovalNotification`, `sendRoleUpdateNotification`, and `sendRelationCreationNotification` - Trigger a notification for the given event.
- `updateRoleMap` - Informs the relation service that a role value has been modified, so that it may update its internal data.

`RelationNotification` Class

An instance of that class is created and sent as a notification when a relation is created or added, updated, and removed. It defines two separate notification type for each of these events, depending upon whether the event concerns an internal or external relation. The static fields of this class describe all notification type strings that the relation service may send (see FIGURE 11-3 on page 153).

The methods of this class allow the listener to retrieve information about the event:

- `getRelationId` - Returns the identifier of the relation affected by this event.
- `getRelationTypeName` - Returns the relation type identifier of the relation affected by this event.
- `getObjectName` - Returns the object name only if the involved relation was an externally defined MBean.
- `getRoleName`, `getOldRoleValue`, `getNewRoleValue` - Give additional information about a role update event.
- `getMBeansToUnregister` - Returns the list of object names for MBeans expected to be unregistered due to a relation removal.

`MBeanServerNotificationFilter` Class

This class is used by the relation service to receive only those notifications concerning MBeans that are role members or external relation instances. It filters MBeans based on their object name, ensuring that the relation service will only receive the deregistration notifications for MBeans of interest.

Its methods allow the relation service to update the filter when it must add or remove MBeans in relations or representing external relations.

The filter instance used by the relation service is not exposed for management by the relation service. This class is described here because it is available as part of the `javax.management.relation` package and may be reused elsewhere.

Interfaces and Support Classes

External relation types and relation instances rely on the interfaces defined in the following figure and may choose to extend the support classes for convenience. Implementations of the JMX specification may also rely on these classes internally.

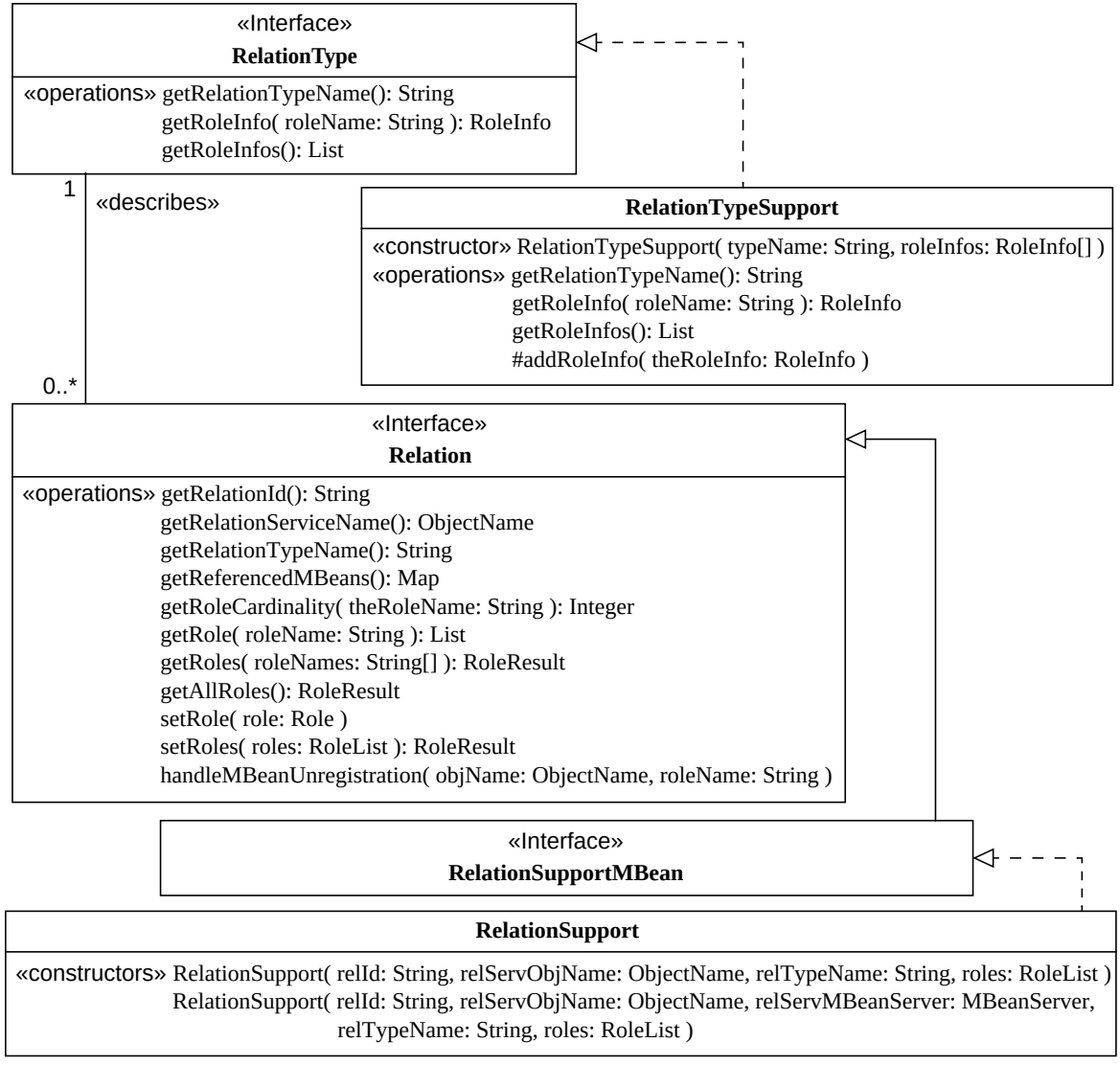


FIGURE 11-4 Interfaces and Support Classes

`RelationType` Interface

Any external representation of a relation type must implement the `RelationType` interface in order to be recognized by the relation service. The methods of this interface expose the name of the relation type and its role information (see “RoleInfo Class” on page 162).

The relation service will invoke the methods of this interface to access the relation type name or the role information. Because a relation type is immutable, the returned values should never change while the relation type is registered with the relation service.

An instance of an object that implements this interface can be added as an external relation type, using the `addRelationType` method of the relation service. Providing its implementation is coherent, it can be accessed through the relation service in the same manner as an internal relation type. In fact, internal relation types are usually represented by an object that also implements this interface

`RelationTypeSupport` Class

This class implements the `RelationType` interface and provides a generic mechanism for representing any relation type. The name of the relation type is passed as a parameter to the class constructor.

There are two ways which may be used to define a specific relation type through an instance of the `RelationTypeSupport` class:

- Its constructor takes an array of `RoleInfo` objects.
- The `addRoleInfo` method takes a single `RoleInfo` object at a time.

Role information may not be added after an instance of this class has been used to define an external relation type in the relation service.

Users may also extend this class to create custom relation types without needing to rewrite the role information access methods. For example, the constructor of the subclass may determine the `RoleInfo` objects to be passed to the superclass constructor. This effectively encapsulates a relation type definition in a class which may be downloaded and instantiated dynamically.

The implementation of the relation service will usually instantiate the `RelationTypeSupport` class to define internal relation types, but these objects are not accessible externally.

`Relation` Interface

The `Relation` interface describes the operations to be supported by a class whose instances are expected to represent relations. Through the methods of this interface, the implementing class exposes all the functionality needed to access the relation.

The class that implements the `Relation` interface to represent an external relation must be instrumented as an MBean. The object must be instantiated and registered in the MBean server before it can be added to the relation service. Then, it can be accessed either through the relation service or whatever management interface it exposes in the MBean Server.

Specified Methods

Each relation is identified in the relation service by a unique relation identifier that is exposed through the `getRelationId` method. The string that it returns must be unique among all relations in the service at the time it is registered. The relation service will refuse to add an external relation with a duplicate or null identifier.

In the same way, the `getRelationTypeName` method must return a valid relation type name, in this case, one that has already been defined in the relation service. An external relation instance must also know about the relation service object where it will be controlled: this can be verified through the `getRelationServiceName` method. This method returns an object name which is assumed to be valid in the same MBean server as the external relation implementation.

The other methods of the Relation interface are used by the relation service to access the roles of a relation under its control. Role values can be read or written either individually or in bulk (see “Role Description Classes” on page 161). Individual roles which cannot be accessed cause an exception whose class indicates the nature of the error (see the exception classes in FIGURE 11-2 on page 151).

The methods for bulk role access follow a “best effort” policy: access to all indicated roles is attempted and roles which cannot be accessed do not block the operation. Those which cannot be accessed, either due to error in the input or due to the access rights of the role, will return an unresolved role object indicating the nature of the error (see “RoleUnresolved Class” on page 164).

The `getReferencedMBeans` method returns a list of object names for all MBeans referenced in the relation, with each object name mapped to the list of roles where the MBean is a member.

Maintaining Consistency

The relation service delegates the responsibility of maintaining the role consistency to the relation object. In that way, consistency checks can be performed when the roles are accessed through methods of the external relation. However, the relation service must be informed of any role modifications, so that it can update its internal data structures and send notifications.

When accessing a role, either getting or setting its value, the relation instance must verify that:

- The named role has a corresponding role information in the relation type.
- The role has the appropriate access rights according to its role information.
- The role value provided for setting a role is consistent with that role's information regarding cardinality and MBean class.

An implementation of the `Relation` interface may rely on the methods of the relation service MBean that are provided to simplify these verifications: `checkRoleReading` and `checkRoleWriting`.

After setting a role, an external relation must call the `updateRoleMap` operation of the relation service, providing the old and new role values. This allows the relation service to update its internal data for maintaining consistency.

The relation service must be informed of all new role values so that it may listen for a deregistration notification concerning any of the member MBeans. When a member MBean of an external relation is deregistered from the is MBean server, the relation service checks the new cardinality of the role it fulfilled.

If the cardinality is no longer valid, the relation service will remove this relation instance (see "RelationService Class" on page 154). If the external relation is still valid, the relation service will call its `handleMBeanUnregistration` method.

When called, this method should remove the MBean from the role where it was referenced (since all role members must be registered MBeans). The guarantee that the relation service will call this method when necessary frees the external relation from having to listen for MBean deregistrations itself. It also allows the relation implementation to define how the corresponding role will be updated. For example, the deregistration of an MBean in a given role could update other roles.

In this case, and in any other case where an exposed method modifies a role value, the implementation should use its own `setRole` method or call the appropriate relation service methods, such as `updateRoleMap`. It is the responsibility of all implementations of the `Relation` interface to maintain the consistency of their relation instance, as well as that of the relation service concerning their role values.

`RelationSupport` Class

This class is a complete implementation of the `Relation` interface that provides a generic relation mechanism. This class must be instantiated with a valid role list that defines the relation instance it will represent. The constructor also requires a unique relation identifier, and the name of an existing relation type that is fulfilled by the given role list.

In fact, the `RelationSupport` class implements the `RelationSupportMBean` which extends the `Relation` interface. This implies that it is also a standard MBean whose management interface exposes all of the relation access methods. Since an external relation must first be registered in the MBean server, external instances of the relation support class can be managed by remote applications.

Users may also extend the `RelationSupport` class in order to take advantage of its implementation when developing a customized external relation. Users may also choose to extend its MBean interface in order to expose other attributes or operations that access the relation. This customization must still maintain the consistency of role access and role updating, but it can use the consistency mechanism built into the methods of the `RelationSupport` class.

The relation service will usually instantiate the `RelationSupport` class to define internal relation instances, but these objects are not accessible externally.

Role Description Classes

The relation service accesses the roles of a relation for both reading and writing values. The JMX specification defines the classes that are used to pass role values as parameters and receive them as results. These classes are also used by external relation MBeans that implement the behavior of a relation.

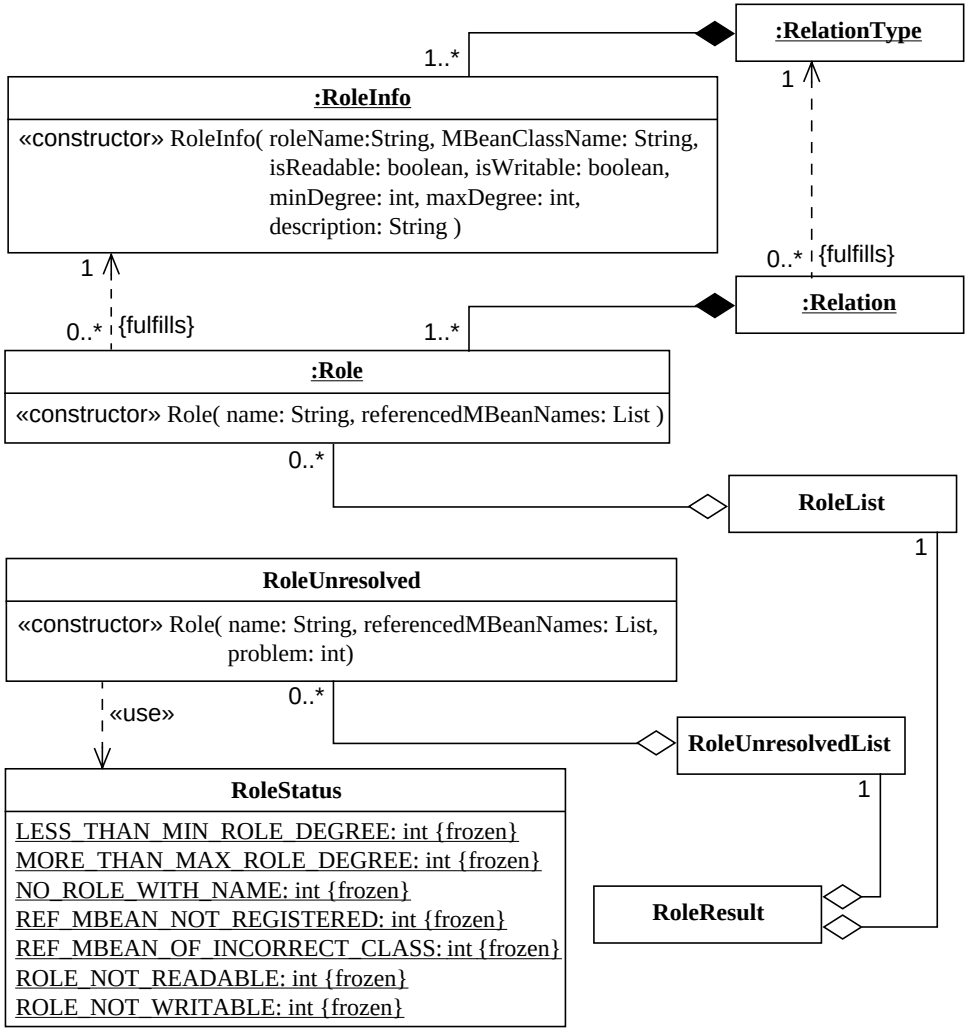


FIGURE 11-5 Role Description Classes

`RoleInfo` Class

The role information provides a metadata description of a role. It specifies:

- The name of the role.
- The multiplicity of the role, expressed as a single closed interval between the minimum and maximum number of MBeans that can be referenced in that role.
- The name of the MBean class of which all members must be instances.

- The role access mode, that is whether the role is readable, writable, or both

When role information is used as a parameter for a new relation type, it is the defining information for a role. When that relation type is declared in the relation service, the service will verify that:

- The role information object is not null.
- The role name is unique among all roles of the given relation type; the relation service does not guarantee that roles with the same name across relation types are identical, this is the user's responsibility.
- The minimum and maximum cardinality define a proper, non-null interval.

`Role` Class

An instance of the `Role` class represents the value of a role in a relation. It contains the role name and the list of object names that reference existing MBeans.

A role value must always fulfill the role information of its relation's type. The role name is the key for associating the role value with its role definition.

The `Role` class is used as a parameter to the `setRole` method of both the relation service and the `Relation` interface. It is also a component of the lists that are used in bulk setter methods and for defining an initial role value. For each role being initialized or updated, the relation service will verify that:

- A role with the given name is defined in the relation type.
- The number of referenced MBeans is greater than or equal to the minimum cardinality and less or equal to the maximum cardinality.
- Each object name references a registered MBean of the expected MBean class.

`RoleList` Class

This class extends `java.util.ArrayList` to represents a set of `Role` objects.

Instances of the `RoleList` class are use to define initial values for a relation. When calling the `createRelation` method of the relation service, roles which admit a cardinality of 0 may be omitted from the role list. All other roles of the relation type must have a well-formed role value in the initial role list.

Role list objects are also use as parameters to the `setRoles` method of both the relation service and the `Relation` interface. These methods will set only the roles for which a valid role value appears in the role list.

Finally, all bulk access methods return a result containing a `RoleList` object representing the roles that were successfully accessed.

`RoleUnresolved` Class

An instance of this class represents an unsuccessful read or write access to a given role in a relation. It is used only in the return values of role access methods of either the relation service or of an object implementing the `Relation` interface.

The object contains:

- The name of the role that could not be accessed
- The value provided for an unsuccessful write access
- The reason why the attempt has failed, encoded as an integer value; the constants for decoding the problem are given in the FIGURE 11-5 on page 162.

`RoleUnresolvedList` Class

This class extends `java.util.ArrayList` to represents a set of `RoleUnresolved` objects. All bulk access methods returns a result containing a `RoleUnresolvedList` object representing the roles that could not be accessed.

`RoleResult` Class

The `RoleResult` class is the return object for all bulk access methods of both the relation service and implementations of the `Relation` interface. A role result contains a list of roles and their values, and a list of unresolved roles and the reason each could not be accessed.

As the result of a getter, the role values contain the current value of the requested roles. The unresolved list contains the roles which could not be read, either because the role name is not valid or because the role does not permit reading.

As the result of a setter, the role values contain the new value for those roles where the operation was successful. The unresolved list contains the roles which could not be written, for any access or consistency reason.

`RoleStatus` Class

This class contains static fields giving the possible error codes of an unresolved role. The error codes are either related to access permissions or consistency checking. The names of the fields identify the nature of the problem; they are given in FIGURE 11-5 on page 162.