Java™ Management Extensions Instrumentation and Agent Specification, v1.2

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Maintenance Release, October 2002

Please
Recycle

Adobe PostScript™

Contents

Preface

This document provides an introduction to the *Java™ Management extensions* (JMX™) and then gives the JMX instrumentation and agent specifications that define these extensions. It is not intended to be a programming guide or 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 API documentation generated by the Javadoc™ tool, that 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 want to build devices, applications, or implementations that 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 Java Develoment Kit (JDK^{TM}) software and be familiar with system or network management. A working knowledge of the JavaBeans^{TM} 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, "**Introduction to the JMX Specification**" 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, "**MBean Instrumentation**" presents standard and dynamic MBeans, their characteristics and design patterns, their naming scheme, the notification model and the MBean metadata classes.

Chapter 3 "Open MBeans" presents the open MBean components and their Java classes.

Chapter 4, "**Model MBeans**" presents the model MBean concept and the Java classes on which it relies.

Part II "JMX Agent Specification"

Chapter 5, "**Agent Architecture**" presents the architecture of the JMX agent and its components.

Chapter 6, "**Foundation Classes**" defines the foundation classes used by the interfaces of the JMX agent components.

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

Chapter 8, "**Advanced Dynamic Loading**" defines advanced class-loading features, including the m-let (management applet) service that loads classes and libraries dynamically from a URL over the network.

Chapter 9, "**Monitoring**" defines the monitoring service that observes the value of an attribute in MBeans and signals when thresholds are reached.

Chapter 10, "Timer Service" defines the timer service that provides scheduling capabilities.

Chapter 11, "Relation Service" defines the relation service that creates relation types and maintains relations between MBeans based on these types.

Chapter 12, "Security" defines the permissions that are used to control access to MBeans.

Related Information

The model MBeans specification in Chapter 4, "Model MBeans", as well as the model MBean Reference Implementation and compatibility test cases, are based on an initial contribution from IBM.

The security specification in Chapter 12 was developed in conjunction with Hewlett Packard and IBM.

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

http://jcp.org/jsr/detail/3.jsp

A number of Java Specification Requests (JSRs) developed through the Java Community Process^{SM} make use of, or are related to the JMX specification:

- JSR 000018 - JAIN OAM API Specification
- JSR 000022 - JAIN SLEE API Specification
- JSR 000070 - IIOP Protocol Adaptor for JMX Specification
- JSR 000077 - J2EE^{TM} Management Specification
- JSR 000138 - Performance Metric Instrumentation
- JSR 000146 - WBEM Services: JMX Provider Protocol Adapter
- JSR 000151 - J2EE 1.4 Specification
- JSR 000160 - Java Management Extensions (JMX) Remoting
- JSR 000174 - Monitoring and Management Specification for the Java Virtual Machine
- JSR 000176 - J2SE 1.5 (Tiger) Release Content

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 and monitoring in the Java programming language. This chapter introduces all these elements, presenting the broad scope of these extensions. 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 and monitoring systems. In addition, the JMX specification is referenced by a number of Java APIs for existing standard management and monitoring technologies.

It should be noted that, throughout the rest of the present document, the concept of *management* refers to both management and *monitoring* services.

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.

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 managed beans (MBeans) 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 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.

- **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, that can leverage emerging technologies. For example, JMX solutions can use lookup and discovery services and protocols such as Jini™ network technology, Universal Plug'n'Play (Upnp), and the Service Location Protocol (SLP).

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

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

  The JMX API is 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 might 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 that 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., in its role as the JMX specification lead.

Because 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 the JavaTest™ software.

Each part of the JMX specification can identify mandatory and optional components. A JMX-compliant implementation must provide all mandatory services, and can 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 that 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 the JMX architecture 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 24.

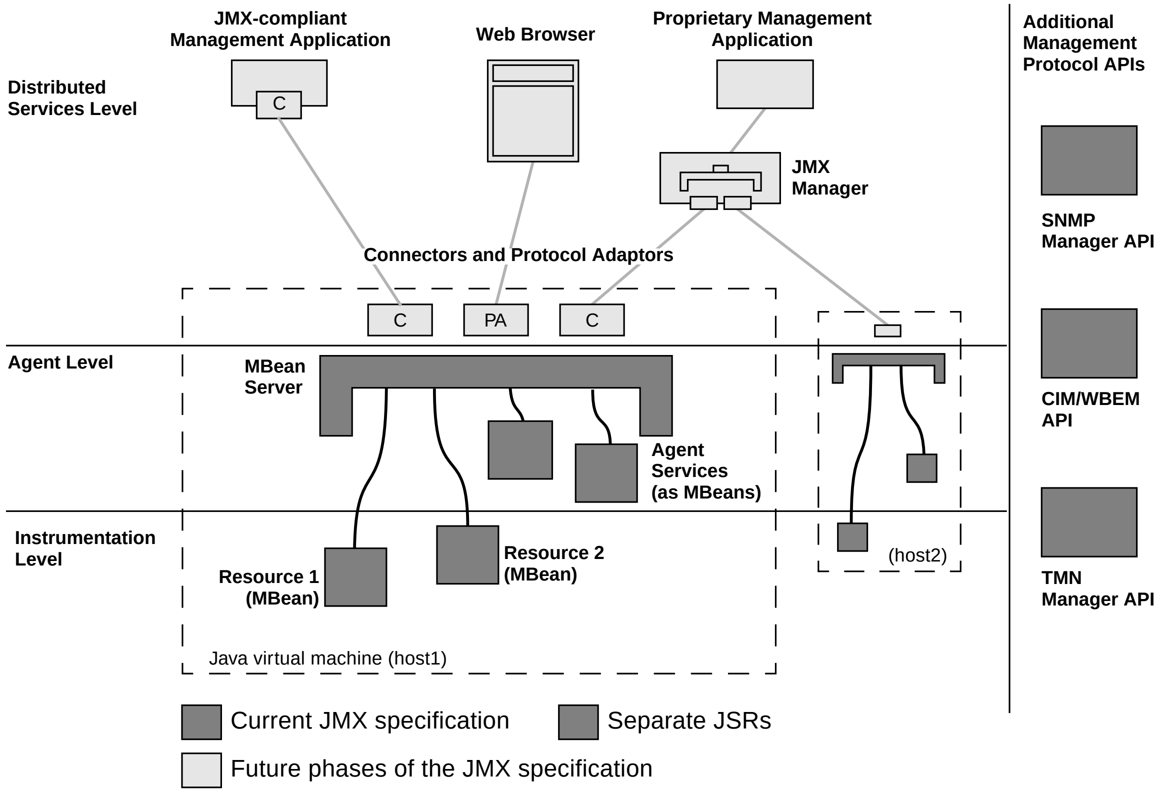

JMX-compliant Management Application
C
Web Browser
Proprietary Management Application
Additional Management Protocol APIs
Distributed Services Level
JMX Manager
SNMP Manager API
Connectors and Protocol Adaptors
C
PA
C
Agent Level
MBean Server
CIM/WBEM API
Agent Services (as MBeans)
Instrumentation Level
Resource 2 (MBean)
Resource 1 (MBean)
(host2)
TMN Manager API
Java virtual machine (host1)
Current JMX specification
Separate JSRs
Future phases of the JMX specification

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, that 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 that offers more flexibility at runtime. For further information, see “Managed Beans (MBeans)” on page 24.

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 that 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.

Because 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.

JMX manageable resources are automatically manageable by agents compliant with the JMX specification. They can also be managed by any system that is not compliant with the JMX specification 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, 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 (JVM) 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, a 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 that provide an MBean server and all 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 (the J2SE™ platform) version 1.3 or above, and on certain profiles of the Java 2 Platform, Micro Edition (the J2ME™ 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 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 the hypertext mark-up language (HTML) or the simple network management protocol (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 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, such as portability, on-demand deployment of management functionality, dynamic and mobility services, and security.

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.

Managed Beans (MBeans)

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 that can be accessed

- Operations that can be invoked
- Notifications that can be emitted (see “Notification Model” on page 25)
- 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 a *getter* and a *setter* methods implement read-write access.

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

- The resources your application manages
- 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 137).

The JMX specification defines four types of MBean: 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 runtime for greatest flexibility.
- **Open MBeans** are dynamic MBeans that rely on basic data types for universal manageability and that are self describing for user-friendliness.
- **Model MBeans** are also dynamic MBeans that are fully configurable and self described at runtime; 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 can provide services that allow distribution of this notification model, thus allowing a management application to listen to MBean events 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 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 MBean extend the metadata classes to provide additional information. Through this inheritance, the standard information will always be available and management applications that 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, a registry for objects in the instrumentation level, and the agent services that 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 that 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 (JVM) 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 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 that 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 is addressed by JSR 160, "JMX Remoting 1.2".**

Adaptors and connectors make all MBean server operations 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 management applet (m-let) service retrieves and instantiates new classes and native libraries from an arbitrary network location.
- **Monitors** observe the numerical or string value of an attribute of several MBeans 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 the agent services are mandatory in a JMX-compliant implementation.

Conformance

This section specifies the mandatory components of the instrumentation and agent levels regarding compliance of implementations to the JMX Instrumentation and Agent Specification, v1.2.

Instrumentation Level

Implementations compliant to the JMX Instrumentation Specification, v1.2, must provide all the components specified in Chapter 2 “MBean Instrumentation”, Chapter 3 “Open MBeans”, and Chapter 4 “Model MBeans” of this specification. This includes all associated classes as defined by their corresponding API documentation generated by the Javadoc tool. These components provide support for the instrumentation of standard and dynamic MBeans.

Agent Level

Implementations compliant with the JMX Agent Specification, v1.2, must 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 the corresponding API documentation generated by the Javadoc tool. Therefore, the implementation of all four agent services that are specified is mandatory.

What Has Changed

This section lists the major changes to the JMX specification since the previous version of the document (*JMX Instrumentation and Agent Specification, v1.1, March 2002*), namely:

- New interface `NotificationEmitter` extends `NotificationBroadcaster`
- Open MBeans made mandatory
- New interface `StandardMBean` generalizes standard MBean design patterns
- Model MBean descriptor fields updated
- `ObjectName` class updated
- `MBeanServerDelegate` updated
- New `MBeanServerConnection` interface
- Class loader repository redesigned and class loading behavior clarified
- New class `MBeanServerBuilder` allows alternative MBean server implementations
- Getters and setters cannot be invoked through the `invoke` method
- New `MBeanServerInvocationHandler` class facilitates construction of MBean proxies
- Timer service allows past times and supports fixed-rate execution
- Monitor service can monitor the same attribute in several MBeans
- New “Security” chapter added

All changes to the specification are marked in the text by change bars.

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 developed according to the rules defined in this chapter are said to be JMX manageable resources.

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

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

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

Developers of applications and devices are free to choose the granularity of objects that are 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 can typically be written as MBeans. MBeans can 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:

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

A class that 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 that implements the `DynamicMBean` interface specified in this chapter is known as a *dynamic* MBean, because 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 MBean are being managed in a JMX agent, management applications handle them in a similar manner.

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

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 instrument rapidly existing resources and other legacy code objects you want to manage.

Public Management Interface

It is not a requirement for an MBean of any type to be a public Java class. However, to be manageable, an MBean must have a public management interface. This public management interface is the MBean’s own interface in the case of a standard MBean, or the `DynamicMBean` interface, in the case of a dynamic MBean.

MBean Public Constructor

The Java class of an MBean, whether standard or dynamic, can optionally be of a public class and have one or more public constructors. An MBean must be of a public concrete class with a public constructor if it is to be instantiated by a JMX agent on demand from a management application.

An MBean can have any number of constructors, to allow an agent to perform an instantiation. An MBean can also optionally have any number of public constructors, all of which are available to a management application through a JMX agent that can instantiate this MBean class.

Public constructors of an MBean can have any number of arguments of any type. 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 can omit all constructors and rely on the *default constructor* which the Java compiler provides automatically in such a case. The default constructor takes no arguments and is public if the class is public. The Java compiler does not provide a default public constructor if any other constructor 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 subclasses
    //
    protected Simple() {
    }

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

Standard MBeans

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 must be observed when defining the interface of their Java object. These naming rules define the concepts of attributes and operations inspired by the JavaBeans™ component model. However, the actual design patterns for the JMX architecture 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 that 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 broadcasts

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. A management application can access these attributes and operations. A management application cannot access methods of the MBean's Java class that are not listed in this interface.

The name of an MBean's Java interface is formed by adding the `MBean` suffix to the MBean's fully-qualified Java class name. For example, the Java class `MyClass` would implement the `MyClassMBean` interface; the Java class `com.example.MyClass` would implement the `com.example.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 the methods in its MBean interface. How it implements these methods determines its response to management operations. An MBean can also define any other methods, public or otherwise, that do not appear in its MBean interface.

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

A standard MBean can 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. The same is true if `MySuperClass` is a dynamic MBean; `MyClass` would also be dynamic. If `MyClass` does not implement a `MyClassMBean` interface, then it will have the same management interface as `MySuperClass`. Otherwise, `MyClass` can redefine its management interface by implementing its own `MyClassMBean` interface.

In this case, `MyClassMBean` defines the management interface, and any management interface of `MySuperClass` is ignored. However, interfaces can also extend parent interfaces, and all methods in the inheritance tree are also considered. Therefore, `MyClassMBean` can extend `MySuperClassMBean`, allowing `MyClass` to extend the management interface of its parent. 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 for it to be a JMX manageable resource.

As of the JMX 1.2 specification, the `javax.management.StandardMBean` class can be used to define standard MBeans with an interface whose name is not necessarily related to the class name of the MBean.

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 that are in its management interface. Attributes are discrete, named characteristics of the MBean that 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.

Attribute names must begin with a character for which `Character.isJavaIdentifierStart` is `true`. The remaining characters in the name must also be `true` for `Character.isJavaIdentifierPart`.

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`*AttributeName*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 *AttributeName* cannot be overloaded, that is, there cannot be two setters or a getter and setter pair for the same name that operate on different types. An object with overloaded attribute names is not a compliant MBean. The *AttributeType* can be of any Java type, including array types, provided that this type is valid in the MBean's runtime 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. If access to individual elements of arrays is needed, it must be implemented through MBean operations.

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

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

To reduce redundancy, only one of the two getter methods for boolean types is allowed. An attribute can 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 that the resource exposes, 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 that does not fit an attribute design pattern is considered to define an operation.

A typical usage is shown in CODE EXAMPLE 2-2 where the MBean exposes the `reset` method to reinitialize 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 runtime. 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 predefined interface that exposes the attributes and operations only at runtime. Instead of exposing them directly through method names, dynamic MBeans implement a method that returns all attributes and operation signatures. In the same way as for standard MBeans, dynamic MBean attribute names and operation names must be valid Java identifiers.

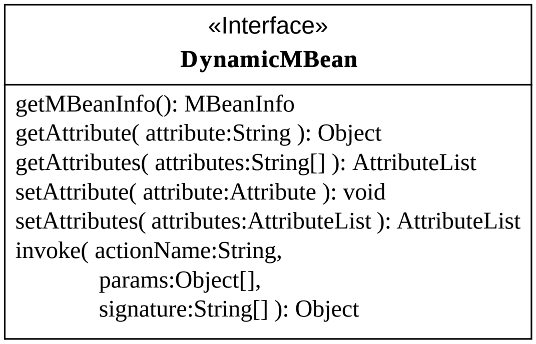
Because the names of the attributes and operations are determined dynamically, these MBeans provide great flexibility when instrumenting existing resources. An MBean that implements the `DynamicMBean` interface provides a mapping for existing resources that do not follow standard MBean design patterns.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

When managed through a JMX agent, dynamic MBeans offer all the same capabilities that are available through standard MBeans. Management applications that rely on JMX agents can manipulate all MBeans in exactly the same manner regardless of their type.

`DynamicMBean` Interface

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.

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.



```
getMBeanInfo(): MBeanInfo
getAttribute( attribute:String ): Object
getAttributes( attributes:String[] ): AttributeList
setAttribute( attribute:Attribute ): void
setAttributes( attributes:AttributeList ): AttributeList
invoke( actionName:String,
        params:Object[],
        signature:String[] ): Object
```

FIGURE 2-1 Definition of the `DynamicMBean` Interface

`getMBeanInfo` *Method*

This method returns an `MBeanInfo` object that 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 that 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 an attribute name and a list of attribute names, respectively, 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 map the exposed attribute names correctly to their values through these methods.

The classes that are used to represent attribute names and values and lists of names and values are described in "Attribute and AttributeList Classes" on page 113. 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 map the new values correctly 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 can verify that the mapping is consistent between the requested operation and that which is exposed. It can also choose between methods that have the same name but different signatures (overloaded methods), though this is not recommended.

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.

In the JMX 1.2 specification, it is forbidden to call getters and setters via `invoke`. Previously, an implementation was free to choose whether or not to allow this. However, if the property `jmx.invoke.getters` does not have an empty value, the code forbidding calling getters and setters through `invoke` is disabled.

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, 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 121). 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 MBean without distinction: 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 that it can gather off-line, or provide a wrapper for resources not written in the Java programming language.

Dynamics

Because the management interface of a dynamic MBean is returned at runtime by the `getMBeanInfo` method, the management interface itself can be dynamic. That is, subsequent calls to this method might not describe the same management interface. It should be noted that the `getMBeanInfo` method is allowed to vary.

Therefore, truly dynamic MBeans, in which you can change the MBean interface, are possible, though they can only be managed by proprietary management applications designed specifically to handle them.

Truly dynamic MBeans of this sort can only be used in limited circumstances, because in general there is no way for a management application to notice that the interface has changed. Developers implementing such systems should consider how they work when more than one management application is connected to the system. Race conditions also need to be considered: for instance, if the MBean sends a notification to say that its interface has changed, at the same time a management application might be performing an operation on the MBean based on its old interface.

Inheritance Patterns

The *introspection* of an MBean is the process that JMX agents use to determine its management interface. This algorithm is applied at runtime by a JMX compliant agent, but it is described here because 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. Because interfaces can 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. If `MyClass` is an instance of the `DynamicMBean` interface, then the return value of its `getMBeanInfo` method will list the attributes and operations of the resource. In other words, `MyClass` is a dynamic MBean.

2. If the `MyClass` MBean is an instance of a `MyClassMBean` interface, then only the methods listed in, or inherited by, the interface are considered among all the methods of, or inherited by, the MBean. The design patterns are then used to identify the attributes and operations from the method names in the `MyClassMBean` interface and its ancestors. In other words, `MyClass` is a standard MBean.

3. If `MyClass` is an instance of the `DynamicMBean` interface, then `MyClassMBean` is ignored. If `MyClassMBean` is not a public interface, it is not a JMX manageable resource. If the MBean is an instance of neither `MyClassMBean` nor `DynamicMBean`, the inheritance tree of `MyClass` is examined, looking for the nearest superclass that implements its own MBean interface.

a. If there is an ancestor called SuperClass that is an instance of 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. If SuperClassMBean is not a public interface, it is not a JMX manageable resource.

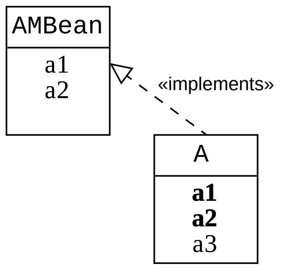
b. When there is no superclass with its own MBean interface, MyClass is not a JMX manageable resource.

As a general rule, the management interface is defined either by the DynamicMBean interface, if the MBean is an instance of DynamicMBean, or by the MBean class or the nearest ancestor that implements its own MBean interface. If neither the class nor any of its superclasses follows the rules for a standard or dynamic MBean, it is not a JMX manageable resource and the JMX agent will raise an MBean error (see “JMX Exceptions” on page 113). Similarly, an MBean interface implemented by a standard MBean must be public.

These rules do not exclude the rare case of a class that inherits from a standard MBean but is itself a dynamic MBean because it implements the DynamicMBean interface. On the other hand, a class that inherits from a dynamic MBean is always a dynamic MBean, even if it follows the rules for a standard MBean.

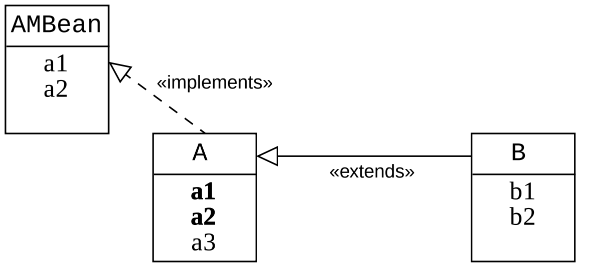
Standard MBean Inheritance

For standard MBeans, the management interface can 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, and so on, stand for attributes or operations recognized by the design patterns for standard MBeans. Various combinations of these example cases are also possible.



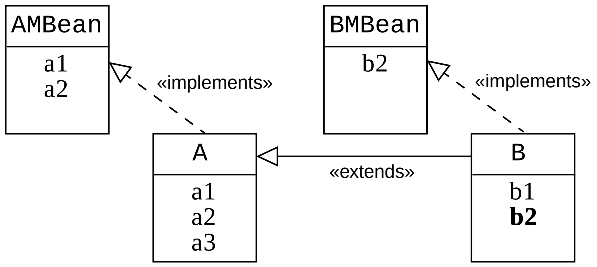
In the simplest case, class A implements class AMBean, which therefore defines the management interface for A: {a1, a2}.

FIGURE 2-2 Standard MBean Inheritance (Case 1)



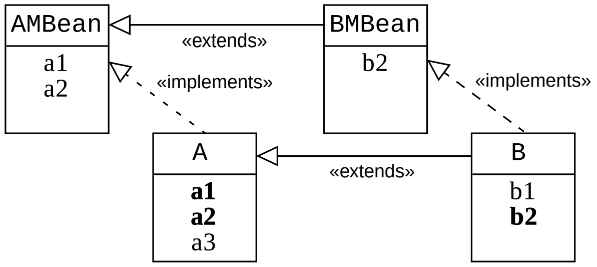
If class B extends A without defining its own MBean interface, then B is also an MBean. B has the same management interface as A: {a1, a2}

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}.

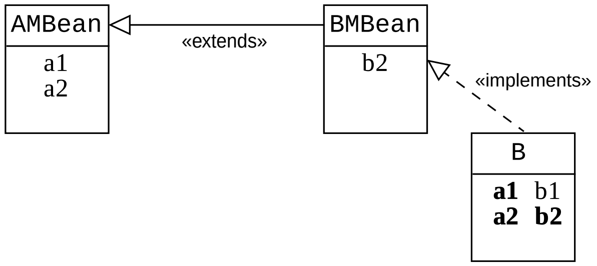
FIGURE 2-4 Standard MBean Inheritance (Case 3)



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)



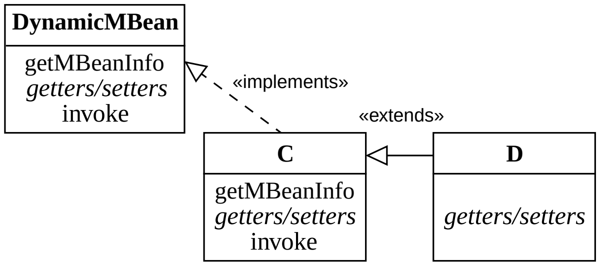
The class B must implement all methods defined in or inherited by the BMBean interface. If it does not inherit them, it must implement them explicitly: {a1, a2, b2}.

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 can also redefine getters, setters and the `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 that 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 that allows MBeans to broadcast such management events, 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 the different events that might 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`, that can signal any type of management event. The `Notification` event can be used directly, or can be subclassed, depending on the information that 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 that 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 that emits notifications. This interface allows listeners to register their interest in the notifications emitted by an MBean.
- The `NotificationEmitter` interface, that extends `NotificationBroadcaster` to allow more control when removing listeners.

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 that are needed. Using a filter, a listener only needs to register once to receive all selected events of an MBean.

Any type of MBean, standard or dynamic, can 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 the types they 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.

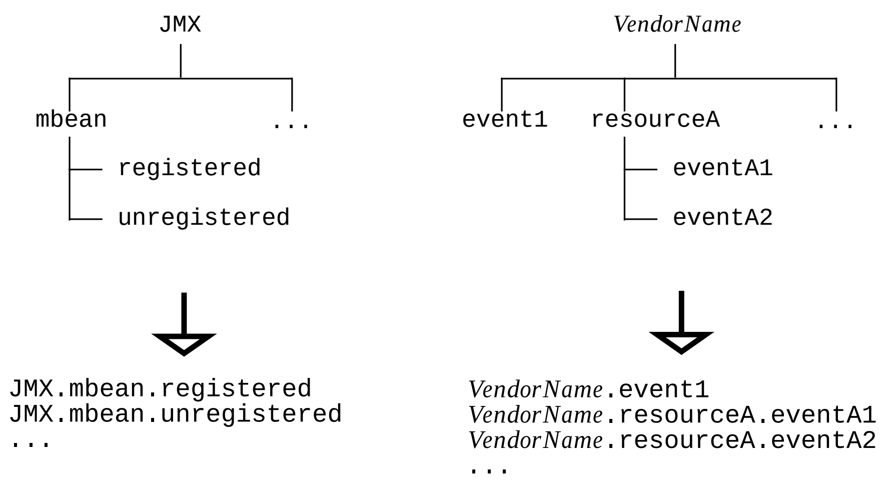


FIGURE 2-8 Structure of Notification Type Strings

`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*, a string expressed in a dot notation similar to Java properties, for example: *vendorName*`.resourceA.eventA1`
- A *sequence number*, 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 will communicate to its listeners

Notification broadcasters use the notification type to indicate the nature of the event to their listeners. Additional information that needs to be transmitted to listeners is 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 can be defined when additional semantics are required within the notification object.

`NotificationBroadcaster` and `NotificationEmitter` Interfaces

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

- `getNotificationInfo` - Gives a potential listener the description of all notifications this broadcaster can 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 59.
- `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 handback object.

  The handback 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 handback object can allow the listener to retrieve context information for use while processing the notification.

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

  The MBean has to maintain a table of listener, filter and handback triplets. When the MBean emits a notification, it invokes the `handleNotification` method of all the registered `NotificationListener` objects, with their respective handback 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. If a listener has been registered several times with this broadcaster, all entries corresponding to the listener will be removed.

Any type of MBean can implement the `NotificationBroadcaster` interface. This might lead to a special case of a standard MBean that has an empty management interface: its role as a manageable resource is to be a broadcaster of notifications. It must be a concrete class, 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 can be registered in a JMX agent, and its management interface only contains the list of notifications that it may send.

Instead of `NotificationBroadcaster`, an MBean can implement its subinterface `NotificationEmitter`, new in JMX 1.2. It is recommended that new code use `NotificationEmitter` rather than `NotificationBroadcaster`.

NotificationEmitter adds a second removeNotificationListener method that specifies the filter and handback values for the listener to be removed. If the listener is registered more than once with different filter and handback values, only a matching one is removed.

Instead of implementing `NotificationBroadcaster` or `NotificationEmitter`, an MBean can inherit from the standard JMX class `NotificationBroadcasterSupport`. This class manages a list of listeners, modified by the `addNotificationListener` and `removeNotificationListener` methods. Its `sendNotification` method sends a notification to all listeners in the list whose filters accept it.

`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 is invoked by a broadcaster MBean when it emits a notification.

Besides the `Notification` object, the listener's handback 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.

Because 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 then relies on the type string, other fields of the notification object and on the handback 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 will receive this notification.

The filter object is provided by the listener when it registers for notifications with the broadcaster, so each listener can 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 can 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*, that 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 need to do so within the JMX architecture, they 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 sending 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 that has changed
- The type of the attribute that 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 can also send other `Notification` objects for other events. The additional fields of this class provide the listener with information about the attribute that 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 that are currently enabled for receiving notifications

Notification listeners observing certain attributes for changes can 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 that 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 its MBeans, regardless of their type, through the MBean metadata classes. All clients, whether management applications or other local MBeans viewing the management interface of an MBean, need to be able to interpret these objects and their constructs. Certain MBeans might provide additional data by extending these classes (see “Open Type Descriptions” on page 65 and “Model MBean Metadata Classes” on page 75).

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 that 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
- `MBeanAttributeInfo` - describes an attribute
- `MBeanConstructorInfo` - describes the signature of a constructor
- `MBeanOperationInfo` - describes the signature of an operation
- `MBeanParameterInfo` - describes a parameter of an operation or constructor
- `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.

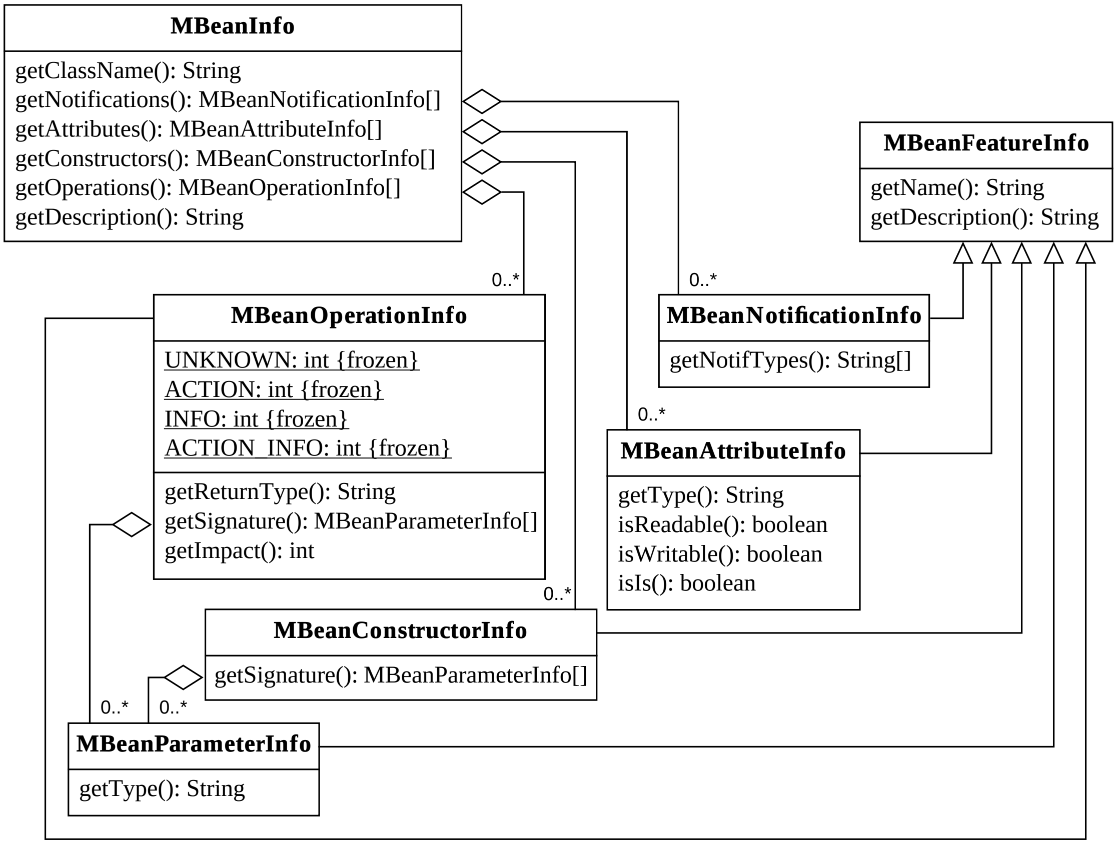


FIGURE 2-9 The MBean Metadata Classes

`MBeanInfo` Class

This class is used to fully describe an MBean: its attributes, operations, its constructors, and the notification types it can 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 that contains information that 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 is used to return a string describing the MBean that is suitable for displaying to a user in a GUI. It describes 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 its components. Therefore, all standard MBeans will have the same description. The `StandardMBean` class provides a way to add custom descriptions while keeping the convenience of the standard MBean design patterns.

For dynamic MBeans, it is the developer's responsibility to ensure that the description strings for the `MBeanInfo` object and all 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 the other component metadata classes. All 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 that provides a human readable explanation of a component. In the case of dynamic MBeans, this string must be provided by the developer. For example, this string must be suitable for displaying to an operator through the user interface of a management application.

`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 that is used to represent it when calling its getter or setter methods. The `getType` method returns a string containing the fully qualified name of this class. The format of this string is identical to that of the string returned by the `getName` method of the `java.lang.Class` class.

For a complete description of this format, please see the API documentation generated by the Javadoc tool for the `java.lang.Class` class in the Java 2 platform standard edition (J2SE) online documentation. As an example, an array of `java.util.Map` type is represented as the string "`[Ljava.util.Map;`", and a two-dimensional array of bytes is represented as the string "`[[B;`". Non-array objects are simply given as their full package name, such as "`java.util.Map`".

MBean access is either readable, writable or both. Read access implies that a manager can get the value of this attribute, and write access that it can 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 that 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`, that 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 58).

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 that 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 assign the impact of each method in its metadata object correctly and consistently. 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 fully qualified class name of the Java object returned by the operation being described. The format of this string is identical to that of the string returned by the getName method of the java.lang.Class class, as described by the API documentation generated by the Javadoc tool in the J2SE online documentation.

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 to provide a name and description.

As in the "MBeanAttributeInfo Class" on page 56, the getType method returns a string containing the fully qualified name of this class. The format of this string is identical to that of the string returned by the getName method of the java.lang.Class class, as described by the API documentation generated by the Javadoc tool in the J2SE online documentation.

`MBeanNotificationInfo` Class

The `MBeanNotificationInfo` class is used to describe the notifications that are sent by an MBean. This class extends the `MBeanFeatureInfo` class to provide a name and a description. The name must 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 can emit. The notification type is a string containing any number of elements in dot notation, not the name of the Java class that implements this notification. As described in “JMX Notification Model” on page 47, a single notification class can be used to send several notification types. All these types are returned in the string array returned by this method.

CHAPTER 3

Open MBeans

This chapter defines a way of instrumenting resources to which MBeans must conform if they are to be "open" to the widest range of management applications. These MBeans are called open MBeans.

In version 1.0 of the JMX specification, open MBeans were incompletely specified and could not be implemented. In version 1.1, open MBeans were completely specified but were optional in implementations. As of version 1.2, open MBeans are a mandatory part of any JMX implementation.

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 when 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 can 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.

Open MBeans are particularly useful where the management application does not necessarily have access to the Java classes of the agent. By using only standard, self describing types, agents and management applications can interoperate without having to share application-specific classes.

In addition, because the set of open MBean data-types is fixed, and does not include self referential types or subclassing, open MBeans are accessible even when the connection between the management application and the agent does not support Java serialization. An important case of this is when the management application is in a language other than Java.

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 the same functionality. It also has the responsibility of providing its own description. However, all 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” on page 62 below. It is the developer’s responsibility to implement the open MBean fully 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, a subclass of `MBeanInfo`. Other component metadata classes are also subclassed and it is the developer’s responsibility to describe the open MBean fully using the proper classes. If an MBean is marked as open in this manner, it is a guarantee that a management application compliant with the JMX specification can immediately make use of all attributes and operations without requiring additional classes.

Because 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 43). 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 make use immediately of MBeans without recompilation, reassembly, 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 `ObjectName`.

In addition, any array of the basic data can be used in open MBeans. A special class, `javax.management.openmbean.ArrayType` is used to represent the definition of single or multi-dimensional arrays in open MBeans.

The following list specifies all data types that are allowed as scalars or as any-dimensional arrays in open MBeans:

- `java.lang.Void`
- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.String`
- `java.math.BigDecimal`
- `java.util.Date`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `java.math.BigInteger`
- `javax.management.ObjectName`
- `javax.management.openmbean.CompositeData` (interface)
- `javax.management.openmbean.TabularData` (interface)

All the wrapper classes for the primitive types are defined and implemented in all Java virtual machines, as are the `BigDecimal`, `BigInteger`, and `Date` classes. The `ObjectName` class is provided by the implementation of the JMX specification. The `CompositeData` and `TabularData` interfaces are used to define aggregates of the basic data types and provide a mechanism for expressing complex data objects in a consistent manner.

Because `CompositeData` and `TabularData` objects are also basic data types, these structures can contain other composite or tabular structures and have arbitrary complexity. A `TabularData` object represents a homogeneous table of `CompositeData` objects, 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 objects, besides `ObjectName`, that are included in the set of basic data types: `CompositeData` and `TabularData`. Object names are described in “ObjectName Class” on page 109. These two objects are specified as interfaces and are supported by an implementation.

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

The two interfaces and implementations provide some semantic structure to build aggregates from the basic data types. An implementation of the `CompositeData` interface is equivalent to a hash table: values are retrieved by giving the name of the desired data item. An instance of a `TabularData` object contains an array of `CompositeData` instances that can be retrieved individually by giving a unique key. A `CompositeData` object is immutable once instantiated, you cannot add an item to it and you cannot change the value of an existing item. Tables are modifiable, and rows can be added or removed from existing instances.

`CompositeData` Interface and Support Class

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

When instantiating the `CompositeDataSupport` class, the user must provide the description of the composite data object in a `CompositeType` object (see “Open Type Descriptions” on page 65). Then, all the items provided through the constructor must match this description of the composite type. Because the composite object is immutable, all items must be provided at instantiation time, and therefore the constructor can verify that the items match the description. The `getOpenType` method will return this description so that other objects that interact with a `CompositeData` object can know its structure.

A `CompositeData` object associates string keys with the values of each data item. 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` Interface and Support Class

The `TabularDataSupport` class defines a table structure with an arbitrary number of rows that can be indexed by any number of columns. Each row is a `CompositeData` object, but all rows must have the same composite data description. The columns of the table are headed by the names of the data items that make up the uniform `CompositeData` rows. The constructor and the methods for adding rows verify that all rows are described by the same `CompositeData` instance.

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 that uniquely identifies each row of the table. When the table is instantiated, or when a row is added, the methods of this class must ensure that the index can uniquely identify all rows.

Both the description of composite object that makes up each row and the list of items that form the index are given by the table description returned by the `getOpenType` method. This method defined in the `TabularData` interface returns the `TabularType` object that describes the table (see "Open Type Descriptions" on page 65).

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

Open Type Descriptions

To manipulate the basic data types, management applications must be able to identify them. Primitive types are given by their wrapper class names and arrays can be represented in a standard way (see the API documentation generated by the Javadoc tool for the `getName` method of the `java.lang.Class` class). However, the complex data types need more structure than a flat string to represent their contents. Therefore, open MBeans rely on description classes for all the basic data types, including special structures for describing complex data.

These description classes are collectively known as the open types because they describe the open MBean basic data types. The abstract `OpenType` class is the superclass for the specialized open type classes for each category of basic data type. It defines common methods for providing a name for the type, giving it a description, and specifying the actual class that is being described. For simple types, this information can be redundant, that is the name is the same as the class name. For composite types, this information allows the user to name each of the items in a complex data structure. The user should also give items a meaningful description for other users who have to manipulate a composite data instance described by this type.

The `SimpleType`, `ArrayType`, `CompositeType`, and `TabularType` classes extend the `OpenType` class to accommodate the different sorts of basic data types.

The primitive types and the `ObjectName` class are described by instances of the `SimpleType` class when they are not used in arrays. This class does not define any more methods than its `OpenType` superclass, it only defines constant fields for each of the primitive types (and for the `ObjectName` class). These fields are themselves instances of the `SimpleType` class where the name, description and class name are predefined. These constants avoid having to instantiate and provide the information for the simple types every time their description is needed.

The `ArrayType` class provides a description of arrays of the basic data types. It inherits its description of the type from the `OpenType` class and adds the information about the number of its dimensions.

Finally, the open type classes for composite and tabular types provide the structure for describing these aggregate types that are specific to open MBeans. These structures are recursive, that is, they are built up from other open type instances. For complex structures, the name and description inherited from `OpenType` provide overall information about the structure.

A `CompositeType` instance gives the name, description and open type object for each item in the data structure. When associated with a `CompositeData` object (see "CompositeData Interface and Support Class" on page 64), the composite type describes the open type for each item of the composite data. This allows any manager that needs to handle the composite data instance to understand how to handle each of its constituent items.

Similarly, the `TabularType` gives the description needed to manipulate a `TabularData` object (see "TabularData Interface and Support Class" on page 64). This includes the open type instance that describes the composite structure of each row, and the list of item names in this structure that index the table.

Open MBean Metadata Classes

To distinguish open MBeans from other MBeans, JMX provides a set of metadata classes that are used specifically to describe open MBeans. These classes inherit from 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 that are particular to open MBeans.

The following interfaces in 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

For each of the above interfaces, a support class provides an implementation and directly extends the MBean metadata class, the name of which 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 59.

Through methods inherited from their superclasses, the open MBean metadata objects describe the management interface of an open MBean. Beyond this description, they provide new methods for returning the extra information required of open MBeans and to return the description of the new aggregate data types. This description is given by the appropriate subclass of the `OpenType` class.

Because open MBeans are a universal way of exchanging management functionality, their description must be rich enough for an operator to understand and use their functionality. All the open MBean metadata classes inherit the `getDescription` method that 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 must be suitable for displaying to a user in a GUI.

The extra information that the open MBean model allows 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 manipulate a new or unfamiliar open MBean intelligently.

`OpenMBeanInfo` Interface and Support Class

The `OpenMBeanInfoSupport` class provides the main information structure for describing an open MBean. It implements the `OpenMBeanInfo` interface and extends the `MBeanInfo` class. Thus, it 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 that 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. Because 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 following sections.

`OpenMBeanOperationInfo` and `OpenMBeanConstructorInfo` Interfaces and Support Classes

The `OpenMBeanOperationInfoSupport` and `OpenMBeanConstructorInfo-Support` classes implement their corresponding interface and extend the `MBeanOperationInfo` and `MBeanConstructorInfo` classes, respectively (see their definition on page 58). 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 an array of `OpenMBeanParameterInfo` instances whose class is described in the next section.

The `OpenMBeanOperationInfo` interface specifies the `getReturnOpenType` method. The open MBean metadata use this method to provide the description of the open type class that is actually returned by the method. For example, if the return type is actually a complex data object, this method returns either a `CompositeType` or `TabularType` instance that describes the data structure of the return type. When the return type is one of the Java primitive types, this information is redundant with the result of the `getReturnType` method. However, by returning the appropriate `SimpleType` instance, this method allows managers to treat all the open types homogeneously.

Only the `OpenMBeanOperationInfo` interface specifies 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 assign the impact of each operation correctly. The `getImpact` method provides information to the user about an operation's side effects, as a complement to its self description.

`OpenMBeanParameterInfo` and `OpenMBeanAttributeInfo` Interfaces and Support Classes

The `OpenMBeanParameterInfoSupport` and `OpenMBeanAttributeInfo-Support` classes implement their corresponding interface extend the `MBeanParameterInfo` and `MBeanAttributeInfo` classes, respectively (see their definition on page 58 and page 56). The former describes one parameter of an operation or constructor, and the latter describes an attribute of an open MBean.

Because these classes are specific to open MBeans, all parameter and attribute types returned by the inherited getType method are necessarily one of the basic data types. To describe the complex data types, both interfaces also specify the getOpenType method, that returns the OpenType subclass that describes the parameter or attribute. This allows a management application to handle all open types, including complex data structures that must be described by ArrayType, CompositeType or TabularType instances.

The open MBean attribute metadata 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 also define the getDefaultValue and getLegalValues methods to provide additional information about the parameter or attribute. These methods have exactly the same functionality in each class.

The getDefaultValue method is used to indicate an optional default value for a given parameter or attribute. At runtime, it returns an Object that 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. It can be null.

The getLegalValues method is used to return an optional list of permissible values for a given parameter or attribute. It returns an Object array, the elements of which must be 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 can be expected. If a set of legal values is supplied, then the methods that implement the DynamicMBean interface must verify that any value written to the attribute or used for this parameter is a member of this set. If getLegalValues returns null, then all assignment compatible values are legal.

Summary of Open MBean Requirements

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, described by an instance of the appropriate OpenType subclass.

- The implementation of the `getMBeanInfo` method must return an instance of a class that implements the `OpenMBeanInfo` interface. This object must fully describes the MBean components using the open MBean metadata objects.
- All 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`
  - `MBeanNotificationInfo.getDescription`
- Instances of `OpenMBeanOperationInfo.getImpact` must return one of the constant values `ACTION`, `INFO`, or `ACTION_INFO`. The value `UNKNOWN` cannot 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 ensure that all metadata for composite data and tabular data provide coherent default values, legal values and indexes.
The developer must also ensure 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 that anyone can use to instrument almost any resource rapidly. Model MBeans are dynamic MBeans that 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 according to the JMX specification in as little as three to five 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 that implement the Java Management extensions.

Overview

The model MBean specification is a 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. Managed resource developers do not have to supply their own implementation of this class. Instead, the resource is programmed to create and configure its model MBean at runtime, 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 can 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 delegates 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 can be other implementations of the model MBean available, depending on the needs of the environment in which the JMX agent is installed.

For example, a JMX agent running on a Java 2 Platform, Micro Edition (J2ME™) environment can provide a `RequiredModelMBean` with no persistence or remoteability. A JMX agent running in an application server's JVM supporting Java 2 Platform, Enterprise Edition (J2EE™) technologies can provide a `RequiredModelMBean` that handles persistence, transactions, remote access, location transparency, and security. In either case, the instrumentation programmer's task is the same. MBean developers do not have to provide different versions of their MBeans for different Java environments, nor do they 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, transactionality, location transparency, and locatability, as applicable in its environment. Instrumentation developers do not need to develop MBeans with their own transactional and persistence characteristics. They merely instantiate model MBeans in the JMX agent and trust that the model MBean implementation is appropriate for the environment in which the JMX agent currently exists.

Any implementation of the model MBean must implement the `ModelMBean` interface that extends the `DynamicMBean`, `PersistentMBean` and `ModelMBeanNotificationBroadcaster` interfaces. The model MBean must expose its metadata in a `ModelMBeanInfoSupport` object that extends `MBeanInfo` and implements the `ModelMBeanInfo` interface. A model MBean instance sends attribute change notifications and generic notifications for which management applications can listen. The model MBean has both a default constructor and a constructor that 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, and 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 can actually invoke the `myAppl.StatusChecker` method on another object that is part of the managed resource. The object `myAppl` can be in this JVM, or it can 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, dynamic, and configurable model MBeans are supported.

The `ModelMBean` interface extends the `DynamicMBean` interface. The implementation of the `DynamicMBean` methods uses the policy in the descriptors to guide how the requests are satisfied. How to do this is described in greater detail in “DynamicMBean Implementation” on page 87.

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 can be specified at both the MBean level and at the attribute level. These methods are called when appropriate by the model MBean implementation itself and not necessarily by the managed resource or a management application. The implementation can choose to not support any actual, direct persistence, in which case these methods will do nothing. However, if persistence is not implemented, an exception will be thrown.

Generic Notifications

The `ModelMBean` interface extends the `ModelMBeanNotificationBroadcaster` interface. This interface defines a `sendNotification` method that sends any `Notification` object to all registered listeners. It also overloads the `sendNotification` method to accept a text message and wraps it in a notification called `Generic` of type `jmx.modelmbean.generic`. 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 121). The managed resource will then create or find and reference one or more instances of the model MBean using the `instantiate`, `createMBean`, `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 needs 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 whenever necessary 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 can 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 to the model MBean. In turn, the model MBean returns these values to the MBean server, that returns them to the request originator, usually a management application. Because 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 if the JMX agent is recycled.

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 significant event defined by a managed resource or management system. 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 as 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 console 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 can then interact with the model MBean through the JMX agent. It finds the available attributes and operations through the `MBeanInfo` provided by the managed resource. For model MBeans, the manager finds 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. Because the model MBean is a notification broadcaster, management notification can be added as listeners for any notifications or attribute change notifications from the managed resource.

Model MBean Metadata Classes

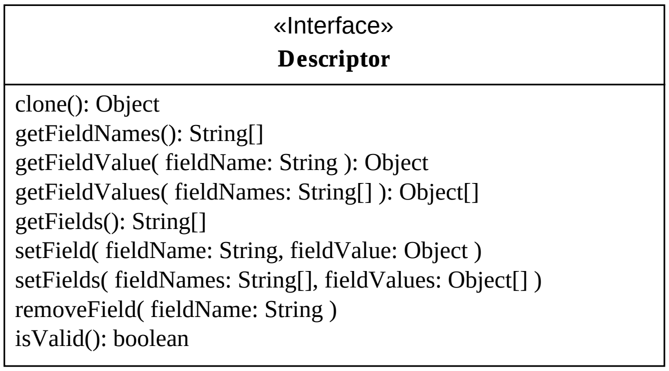
The management interface of a model MBean is described by its `ModelMBeanInfo` instance. The `getMBeanInfo` method of a model MBean (specified by the `DynamicMBean` interface) must return an extension of `MBeanInfo` that also supports the `ModelMBeanInfo` interface. The `ModelMBeanInfo` interface adds MBean *descriptor* components to the `MBeanInfo`. The `ModelMBeanInfo` interface returns arrays of `ModelMBeanAttributeInfo`, `ModelMBeanOperationInfo`, `ModelMBeanConstructorInfo`, and `ModelMBeanNotificationInfo` instances. These classes extend the MBean metadata classes of the same name without 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 runtime 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 96 for a discussion of the valid field names and values that must be supported.

The JMX specification includes a standard implementation of this interface called `DescriptorSupport`. Most applications will use this rather than implementing the `Descriptor` interface themselves.


«Interface»
Descriptor
clone(): Object
getFieldNames(): String[]
getFieldValue(fieldName: String): Object
getFieldValues(fieldNames: String[]): Object[]
getFields(): String[]
setField(fieldName: String, fieldValue: Object)
setFields(fieldNames: String[], fieldValues: Object[])
removeField(fieldName: String)
isValid(): boolean

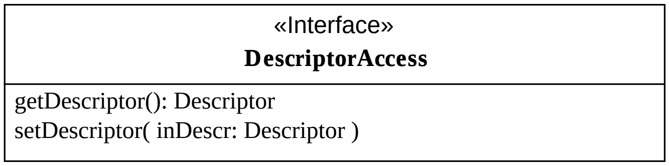
The meaning of these methods is explained in the API documentation generated by the Javadoc tool, that accompanies this specification. A brief summary is presented in the following table.

TABLE 4-1 `Descriptor` Interface Methods

Method	Description
`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.


«Interface»
DescriptorAccess
getDescriptor(): Descriptor
setDescriptor(inDescr: Descriptor)

The meaning of these methods is explained in the API documentation generated by the Javadoc tool, that accompanies this specification. A brief summary is presented in the following table.

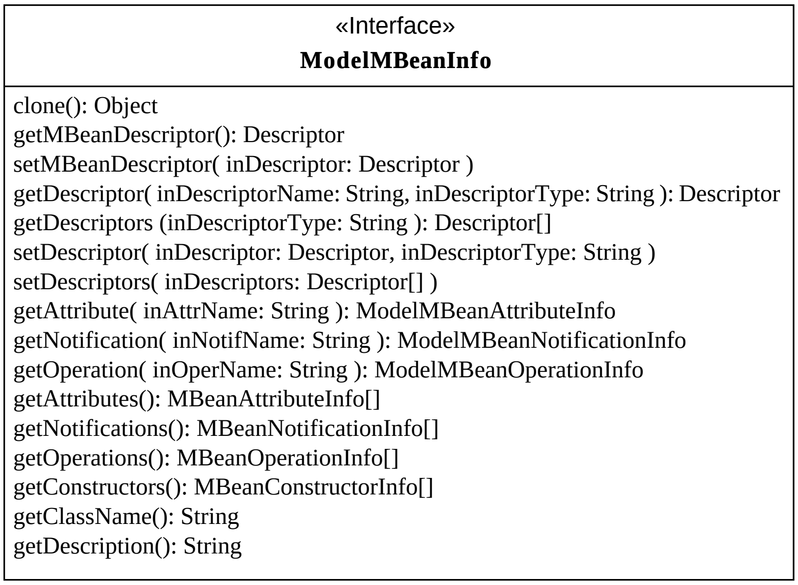
TABLE 4-2 `DescriptorAccess` Interface Methods

Method	Description
getDescriptor	Returns a copy of the descriptor associated with the metadata class
setDescriptor	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 that implements the `ModelMBeanInfo` interface.

The JMX specification includes a standard implementation of this interface called `ModelMBeanInfoSupport`. Most applications will use this rather than implementing the `ModelMBeanInfo` interface themselves.



«Interface»
ModelMBeanInfo

```
clone(): Object
getMBeanDescriptor(): Descriptor
setMBeanDescriptor( inDescriptor: Descriptor )
getDescriptor( inDescriptorName: String, inDescriptorType: String ): Descriptor
getDescriptors (inDescriptorType: String ): Descriptor[]
setDescriptor( inDescriptor: Descriptor, inDescriptorType: String )
setDescriptors( inDescriptors: Descriptor[] )
getAttribute( inAttrName: String ): ModelMBeanAttributeInfo
getNotification( inNotifName: String ): ModelMBeanNotificationInfo
getOperation( inOperName: String ): ModelMBeanOperationInfo
getAttributes(): MBeanAttributeInfo[]
getNotifications(): MBeanNotificationInfo[]
getOperations(): MBeanOperationInfo[]
getConstructors(): MBeanConstructorInfo[]
getClassName(): String
getDescription(): String
```

`ModelMBeanInfo` Implementation

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

- It must 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 is recommended that it implement the following constructors, though implementations of the JMX specification do not have to check this:

TABLE 4-3 `ModelMBeanInfo` Constructors

Constructor	Description
`ModelMBeanInfo`	The default constructor that constructs a `ModelMBeanInfo` with empty component arrays and a default MBean descriptor.
`ModelMBeanInfo` (with `ModelMBeanInfo`)	Constructs a `ModelMBeanInfo` that 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 one constructed by the `ModelMBeanInfo` implementation. The constructed MBean descriptor must not be null. It contains at least the `name` and `descriptorType` fields. The name should be the MBean class, as returned by the `getClassName` method inherited from `MBeanInfo`.
`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:

TABLE 4-4 ModelMBeanInfo Methods

Method	Description
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 55):

TABLE 4-5 ModelMBeanInfo Interface Method

Method	Description
getAttributes	Returns an array of all ModelMBeanAttributeInfo objects.
getNotifications	Returns an array of all ModelMBeanNotificationInfo objects.
getOperations	Returns an array of all ModeMBeanOperationInfo objects.

TABLE 4-5 ModelMBeanInfo Interface Method

Method	Description
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 that matches the name given by the getName method of the corresponding metadata object. It must have a descriptorType with the value "attribute". It can also contain the following defined fields: value, default, displayName, getMethod, setMethod, protocolMap, persistPolicy, persistPeriod, currencyTimeLimit, lastUpdatedTimeStamp, visibility, and presentationString. See "Attribute Descriptor Fields" on page 98 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 that 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 that has at least its name and descriptorType fields set.
- A constructor accepting a name, type, description, isReadable, isWritable, and isIs boolean parameters that 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 that 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 that 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 can also contain the defined fields displayName, visibility, and presentationString. See “Operation Descriptor Fields” on page 99 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 that 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 that has at least the name and descriptorType fields set.
- A constructor accepting a name, a description, and an MBeanParameterInfo array that 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 that 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 that 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 can also contain the defined fields displayName, targetObject, targetType, value, currencyTimeLimit, lastUpdatedTimeStamp, visibility, and presentationString. See “Operation Descriptor Fields” on page 99 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 that 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 that at least has its name and descriptorType fields set.
- A constructor accepting a name, description, MBeanParameterInfo array, type, and an impact that 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 that 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 that matches the name given by the getName method of the corresponding metadata object. It must have a descriptorType with the value "notification". It can also contain the defined fields displayName, severity, messageID, log, logfile, visibility, and presentationString. See "Notification Descriptor Fields" on page 100 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 that 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 that has at least its name and descriptorType fields set.
- A copy constructor accepting a ModelMBeanNotificationInfo.

Model MBean Specification

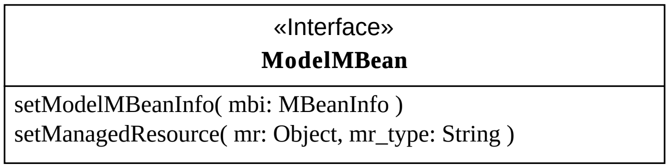
All JMX agents must have an implementation class of a model MBean called javax.management.modelmbean.RequiredModelMBean. The RequiredModelMBean and any other compliant model MBean must comply with the following requirements:

- Implement the `ModelMBean` interface that extends the following interfaces:
  - `DynamicMBean`
  - `PersistentMBean`
  - `ModelMBeanNotificationBroadcaster`
- Return an object from the `getMBeanInfo` method of the `DynamicMBean` interface that:
  - 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 that need to be manageable instantiate the `RequiredModelMBean` or another compliant model MBean using the MBean server's `createMBean` method, passing as a parameter 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.


«Interface»
ModelMBean
setModelMBeanInfo(mbi: MBeanInfo)
setManagedResource(mr: Object, mr_type: String)

`ModelMBean` Implementation

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

`setModelMBeanInfo` *(with* `ModelMBeanInfo`*)*

This method 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 the `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`)*

This method 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`. An implementation must support `ObjectReference`, but need not support the other types. It can also define implementation-specific types.

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

This method returns the ModelMBeanInfo object that 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 that 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

These methods are invoked to get attribute information from this instance of the ModelMBean implementation synchronously. Model MBeans that support attribute value caching will perform 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. Otherwise, if a value field is defined in the descriptor and it is not “stale” as described below, its contents are returned. Otherwise, if no value or getMethod descriptor fields are defined the default 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 never be stale. If `currencyTimeLimit` is `-1`, then the value will always 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. The `lastUpdatedTime` will be reset to the current time. If a `getMethod` is not defined and the value is stale, then the `default` from the `Descriptor` for the attribute will be returned.

`setAttribute` *and* *`setAttributes`*

These methods are 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 `-1`. The `lastUpdatedTime` field will be set whenever the `value` field is set.

`invoke`

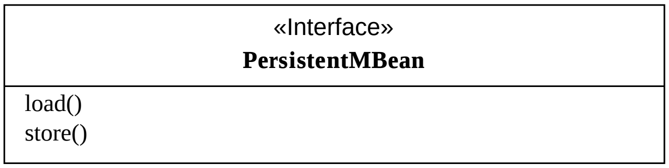
The `invoke` method will execute the operation name passed in with the parameters passed in, according to the `DynamicMBean` interface. The method will be invoked on the model MBean's managed resource (as set by the `setManagedResource` method). If the `targetObject` field of the descriptor is set and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

If operation caching is supported, the response from the operation will be cached in the `value` and `lastUpdatedTimeStamp` fields of the operation's descriptor if the `currencyTimeLimit` field in the operation's descriptor is not `-1`. If `invoke` is executed for a method and the `value` field does not contain a stale value then it will be returned and the associated method will not actually be executed. This is true even if the `invoke` parameters are not the same as the parameters that produced the cached value. If this is not appropriate for the operation, caching must not be used.

The `RequiredModelMBean` class extends these semantics. If the method name and signature supplied to `invoke` correspond to a public method of the `RequiredModelMBean` class itself, then that method is invoked. Otherwise, the behaviour is as explained above.

`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 might do nothing. If the model MBean implementation does not support persistence, then these methods will throw an exception. 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. This should only be called by an implementation of the `ModelMBean` interface.

- `store`

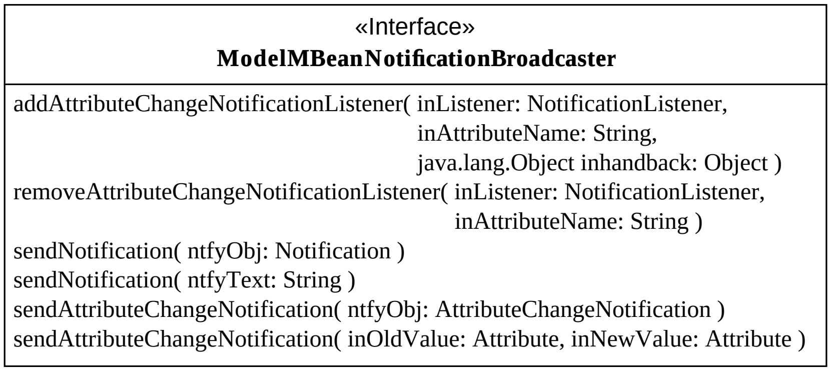
  Writes the MBean in a persistent store. It is only called by an implementation of the `ModelMBean` interface to store itself according to persistence policy for the MBean. When used, it can be called with every invocation of `setAttribute`, or on a periodic basis.

`ModelMBeanNotificationBroadcaster` Interface

This interface extends the `NotificationBroadcaster` interface and must be implemented by any MBean needing 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 different set of listeners to those to which other notifications would go. All other notifications go to listeners who registered using the methods defined in the `NotificationBroadcaster` interface. `AttributeChangeNotifications` are also sent to those listeners, but in addition they are sent to listeners added using `addAttributeChangeNotificationListener`.

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 that implement the intended effect; namely, they change the attribute value on the model MBean.


«Interface»
ModelMBeanNotificationBroadcaster
addAttributeChangeNotificationListener(inListener: NotificationListener, inAttributeName: String, java.lang.Object inhandback: Object)
removeAttributeChangeNotificationListener(inListener: NotificationListener, inAttributeName: String)
sendNotification(ntfyObj: Notification)
sendNotification(ntfyText: String)
sendAttributeChangeNotification(ntfyObj: AttributeChangeNotification)
sendAttributeChangeNotification(inOldValue: Attribute, inNewValue: Attribute)

`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 that 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 called "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 that 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 that 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 an `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 includes policy for managing its persistence, caching, and protocol mapping. For operations, the method signature must be defined. For notifications, the type, identity, severity, and logging policy are defined optionally.

Notification Logging Policy

The model MBean will log notifications if the `log` field 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 policies can be implemented as an option. 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 can have different levels of persistence. When there is no persistence, objects will be completely transient in nature. In a simple implementation, the `ModelMBeanInfo` can be serialized into a flat file. In a more complex environment, persistence can 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 can 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, for the resource to be accessible.

A `ModelMBean` implementation that supports persistence will attempt to prime itself when it is registered in the MBean server, by calling the `ModelMBean.load` method. This method must determine where the persistent representation of the MBean is located, retrieve it, and initialize the model MBean. For simpler representations, the directory and filename to be used for persistence can be defined directly 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 the 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 service characteristics can have an impact upon performance and security.

Because 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 checkpoint model MBeans periodically, it will do so 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 will apply to all 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.

Behavior of Cached Values

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, 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 that are expressed in units of seconds. If the current time is past `lastUpdateTimeStamp` + `currencyTimeLimit`, then the attribute value is *stale*. If the `currencyTimeLimit` is `-1`, then the attribute value is always stale. If the `currencyTimeLimit` is `0`, then the attribute value is never stale.

If a `getAttribute` is received for an attribute with a stale value (or no value) in the descriptor, then:

- If there is a `getMethod` for the attribute, it will be invoked and the returned value will be recorded in the `value` field in the descriptor for the attribute. The `lastUpdatedTimeStamp` will be reset, and the caller will be handed the new value.
- If there is no `getMethod` defined, then the default value from the `default` field in the descriptor for the attribute will be returned.

Protocol Map Support

The model MBean's default behavior and simple APIs satisfy 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, for example, 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 that 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 operation 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 might also need to register MBeans that need to be locatable from other JMX agents without advance knowledge about which JMX agent the MBean is currently registered with. MBeans that need to be locatable in this type of environment define an `export` field in the MBean descriptor in its `ModelMBeanInfo` object.

The value of the `export` field is the external name or object required to export the MBean appropriately. If the JMX agent does not support interoperation 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 or null, 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 is available from these types of managers can be significantly different. Enterprise managers might 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 or 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 that is nearly always visible. The smallest grain is `4`, for an MBean or a component that 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 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 can be defined in a management solution to store custom information as needed.

Field names are not case sensitive. The field `descriptorType` can also be referred to as `DescriptorType` or `DESCRIPTORTYPE`. The case used when a descriptor is created or updated is preserved. It is recommended that the form shown here be used consistently.

Certain field values are also case insensitive. This is true for the values of the `descriptorType`, `persistPolicy`, and `log` fields.

MBean Descriptor Fields

These are the predefined fields for the MBean descriptor. These values are valid for the entire model MBean. These values can 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 that always contains the value “`MBean`”.

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

`persistPolicy` - Defines the default persistence policy for attributes in this MBean that do not define their own `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 affecting performance.

- `Always` - This is a synonym of `OnUpdate`, which is recognized for compatibility reasons. It is recommended that applications use `OnUpdate` instead. An implementation of the `Descriptor` interface, such as `DescriptorSupport`, can choose to replace a value of "`Always`" for `persistPolicy` by a value of "`OnUpdate`".

`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` has not elapsed since the previous storage. The value of this field is a number of seconds, specified as a decimal integer string.

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

`persistName` - The filename in which this MBean is stored. This should be the same as the MBean's name (for this reference implementation). For other implementations, this value can be a keyword or value to assist the appropriate persistence mechanism.

`log` - A boolean where `true` indicates that all sent notifications are logged to a file, and `false` indicates that no notification logging will 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 are 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 `-1`, then the value must be retrieved on every request. If `currencyTimeLimit` is 0, then the value is never stale. The value of this field is a number of seconds, specified as a decimal integer string.

`export` - Its value can be any object that is serializable and contains the information necessary to make the MBean locatable. A value of `null`, or a String value of `F` or `false`, 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`, indicating a level of granularity for the MBean. A value of `1` is for the large grain and most frequently viewed MBeans. A value of 4 is the smallest grain and possibly the least frequently viewed MBeans. This value can be used by adaptors or management applications.

presentationString - XML-encoded string that describes how the attribute will 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 that 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 that 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 from the managed resource. 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 is set by the managed resource and used by the adaptors as hints for representing this attribute to management applications.

persistPolicy - Defines the persistence policy for this attribute. If defined, this overrides a persistPolicy in the MBean descriptor. The possible values and their meanings are the same as for the persistPolicy in the MBean descriptor, described on page 96.

persistPeriod - The meaning of this field is the same as for the persistPeriod in the MBean descriptor, described on page 97.

currencyTimeLimit - The meaning of this field is the same as for the currencyTimeLimit in the MBean descriptor, described on page 97.

`lastUpdatedTimeStamp` - Time stamp from when the `value` field was last updated. The value of this field is a string created by code equivalent to `Long.toString(System.currentTimeMillis())`.

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

`presentationString` - XML-encoded string that describes how the attribute is 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 that always contains the value "`operation`".

`displayName` - Display name of the operation.

`value` - 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 `-1`.

`currencyTimeLimit` - The period of time in seconds that the `value` is current and not stale. If the `value` 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 `-1`, then the value is always stale and is not cached. If the `currencyTimeLimit` is `0`, then the value is never stale. The value of this field is a number of seconds, specified as a decimal integer string.

`lastUpdatedTimeStamp` - The time stamp of when the `value` field was updated. The value of this field is a string created by code equivalent to `Long.toString(System.currentTimeMillis())`.

`visibility` - Integer set from `1` to `4` indicating a level of granularity for the MBean operation. A value of `1` is for the large grain and most frequently viewed MBean operations. A value of `4` is the smallest grain and the least frequently viewed MBean operations. This value can 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.

`targetObject` - a resource to which invocations of this method are directed. This overrides the managed resource specified by the `ModelMBean.setManagedResource` method for the MBean as a whole.

`targetType` - the type of the resource defined by the `targetObject`. Every implementation must recognize the type `ObjectReference`, where calling the MBean operation results in calling a method with the same name and parameter types on the `targetObject`. Implementations can also recognize the predefined types `ObjectReference`, `Handle`, `IOR`, `EJBHandle`, and `RMIReference`, as well as implementation-defined types.

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 that 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 is 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 are 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 that 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 Java Management extensions (JMX) agent architecture and its basic concepts. It serves as an introduction to the JMX agent specification.

Overview

A JMX agent is a management entity that runs in a Java Virtual Machine (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 that represent managed resources, as specified in Part I, “JMX Instrumentation Specification” on page 31.
- The MBean server, 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 that 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 that 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 predetermined intervals and act as a scheduler

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

Remote management applications can 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.

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

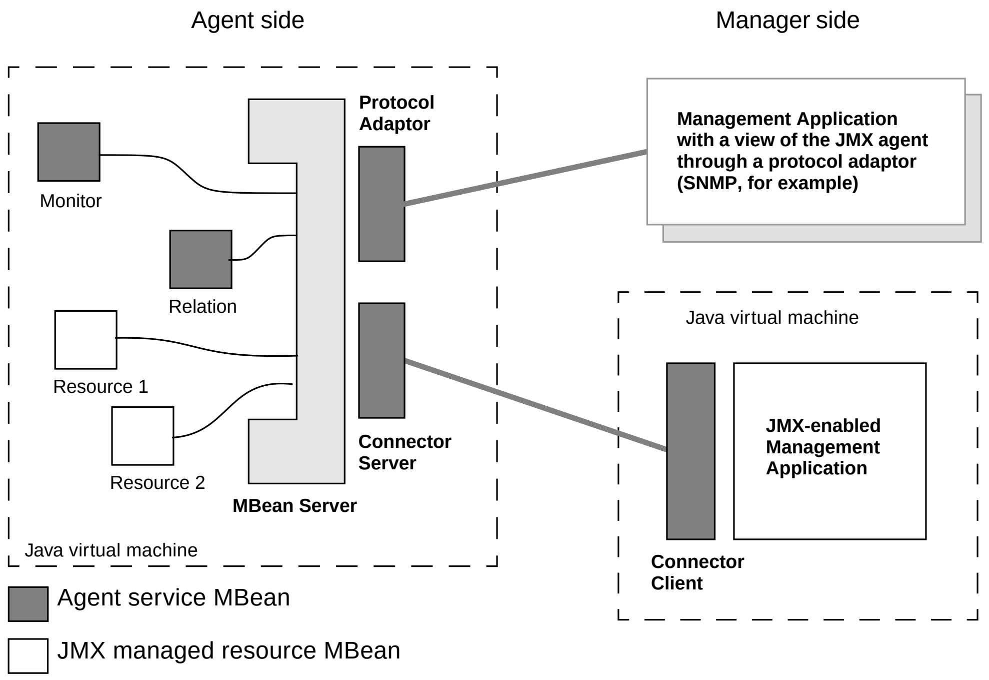


FIGURE 5-1 Key Concepts of the JMX Agent Architecture

The JMX architecture allows objects to perform the following operations on a JMX agent. These objects can 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 these operations are performed, either directly or indirectly, through the MBean server of the JMX agent.

JMX Compliant Agent

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

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

All these components are specified in this document and in the associated API documentation generated by the Javadoc tool. 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, namely, a management application developed using the distributed services of the JMX specification. 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 because 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 to apply the management operation they receive to the MBeans, and 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 an implementation of the JMX specification 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 that are used as argument types or returned values in methods of various Java Management extensions (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 can 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 that 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 can 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 111), 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 can contain any characters except the colon (`:`) that terminates the domain name, and the asterisk (`*`) and question mark (`?`), that are wildcard characters. JMX always handles the domain name as a whole, therefore any semantic subdefinitions within the string are opaque to a JMX implementation.

To avoid collisions between MBeans supplied by different vendors, a useful convention is to begin the domain name with the reverse DNS name of the organization that specifies the MBeans, followed by a period and a string whose interpretation is determined by that organization. For example, MBeans specified by Sun Microsystems Inc., DNS name `sun.com`, would have domains such as `com.sun.MyDomain`. This is essentially the same convention as for Java-language package names.

Key Property List

The key property list allows 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 can contain any number of key properties, the order of which is not significant.

The value in a key property is an arbitrary string, except that it cannot contain any of these characters:

```
:",=*?
```

If strings with these special characters are required, a quoting mechanism exists. Use `ObjectName.quote` to convert any string into a quoted form that is usable as a key property value, and `ObjectName.unquote` to convert back to the original string.

A useful convention is to include a `type` property in every object name. Thus, the set of all MBeans of type `user` can be matched with the pattern "`*:type=user,*`".

String Representation of Names

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

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

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

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

Pattern Matching

Most of the basic MBean operations (for example, `create`, `get` and `set` attributes) need to identify one MBean uniquely 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 described in the following sections.

Domain Name

The matching syntax is consistent with standard file globbing, namely:

- An asterisk `(*)` matches any character sequence, including an empty one
- A question mark `(?)` matches any one single character

Key Property List

There is no wildcard matching performed on property names or on property values. Only complete property-value pairs are used in pattern matching. While key properties are atomic, the list of key properties can be incomplete and used as a pattern.

The `*` is the wildcard for key properties, it replaces any number of key properties that can 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 can 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

If the example 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 to convey 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 API documentation generated by the Javadoc tool.

`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.

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` also wraps the actual exception defined by the user and thrown by an MBean method.

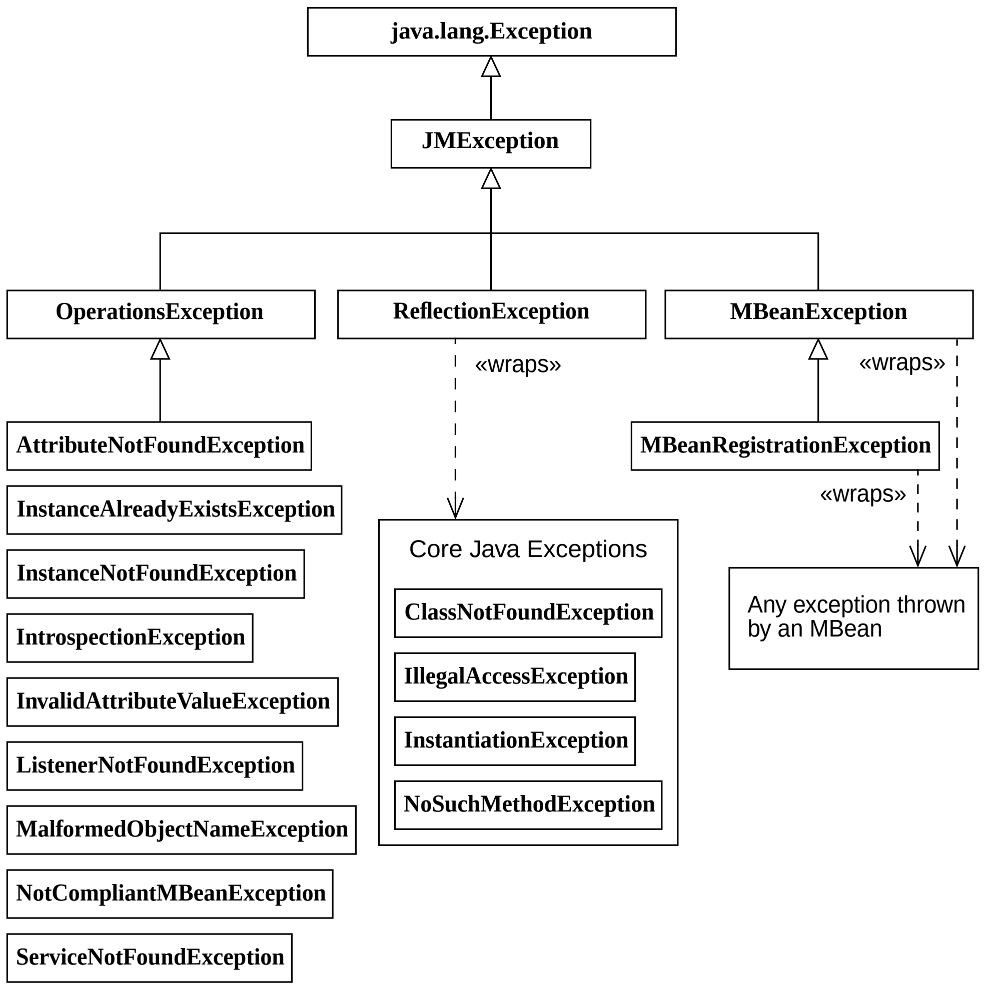


FIGURE 6-1 The JMX Exceptions Object Model

`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` has three subclasses to wrap different sorts of exceptions. The `RuntimeOperationsException` class wraps runtime exceptions thrown while performing operations in the agent. The `RuntimeMBeanException` class wraps runtime exceptions thrown by an MBean. Finally, the `RuntimeErrorException` class is used by the MBean server to wrap `Error` objects thrown by an MBean.

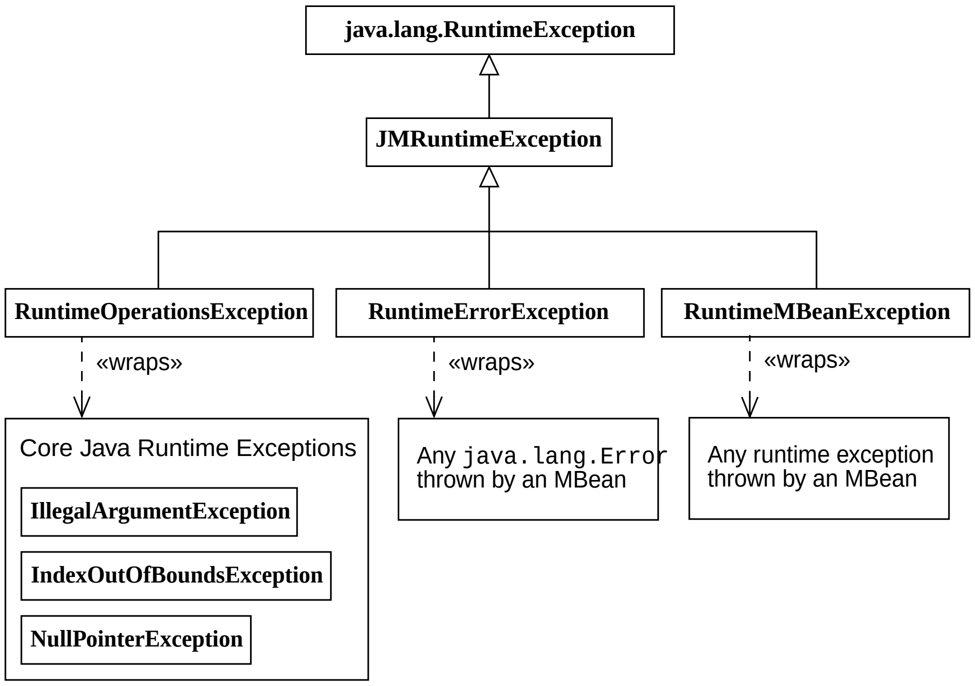

java.lang.RuntimeException
JMRuntimeException
RuntimeOperationsException
RuntimeErrorException
RuntimeMBeanException
«wraps»
«wraps»
«wraps»
Core Java Runtime Exceptions
IllegalArgumentException
IndexOutOfBoundsException
NullPointerException
Any java.lang.Error thrown by an MBean
Any runtime exception thrown by an MBean

FIGURE 6-2 The JMX Runtime Exceptions Object Model

Description of JMX Exceptions

The following sections describe the exceptions thrown in the JMX specification.

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 exception classes that can be “wrapped” in a `ReflectionException` include the following:

- `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 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 in the MBean server that is not an MBean that is compliant with the JMX specification.

`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.

The exception classes that can be "wrapped" in a `RuntimeOperationsException` include the following:

- `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 is 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 must be caught and thrown again as a `RuntimeErrorException`.

CHAPTER 7

MBean Server

This chapter describes the Managed Bean server, or MBean server, that is the core component of the Java Management extensions (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 that 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 are registered in an MBean server:

- MBeans that represent managed resources for management purposes. These resources can be of any kind: application, system, or network resources that provide a Java interface or a Java wrapper.
- MBeans that add management functionality to the agent. This functionality can be fully generic, for example, providing a logging or a monitoring capability, or it can 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, can 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 that 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 instance 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 used in the JMX agent it represents.

An agent application uses these methods to create the single or multiple MBean servers containing 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 that 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.

MBean Server Permission Checking

Access to the `MBeanServerFactory` class's static methods is controlled by the `MBeanServerPermission` class. `MBeanServerPermission` extends basic Java permissions, and grants access to the following MBean server operations:

- `createMBeanServer`
- `findMBeanServer`
- `newMBeanServer`
- `releaseMBeanServer`

Permission checking is covered further in Chapter 12, "Security".

Registration of MBeans

The first responsibility of the MBean server is to be a *registry* for MBeans. MBeans can 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 109. The object name allows an MBean to be identified uniquely in the context of the MBean server. This uniqueness is checked at registration time by the MBean server, that will refuse MBeans with duplicate names.

MBean Registration Control

The MBean developer can exercise some control over the registery and unregistery 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 that 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 have information about 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 invokes before registering the MBean. The MBean is not registered if any exception is raised by this method. This method might throw the `MBeanRegistrationException` that will be thrown again unchanged by the MBean server. Any other exception will be caught by the MBean server, encapsulated in an `MBeanRegistrationException` and thrown again.

  This method can 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 example, check on the existence of services upon which the MBean depends. 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 MBean was registered successfully, and *false* if the MBean could not be registered. If registration failed, this method can free resources allocated in preregistration.
- `preDeregister` - this is a callback method that the MBean server invokes before unregistering an MBean.

  This method might throw an `MBeanRegistrationException`, that is thrown again unchanged by the MBean server. Any other exception is caught by the MBean server, encapsulated in an `MBeanRegistrationException` and thrown again. The MBean is not unregistered if any exception is raised by this method.
- `postDeregister` - This is a callback method that the MBean server invokes after unregistering 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 unregistration is performed. The methods illustrated with a thick border are `MBeanServer` methods, the others are implemented in the MBean.

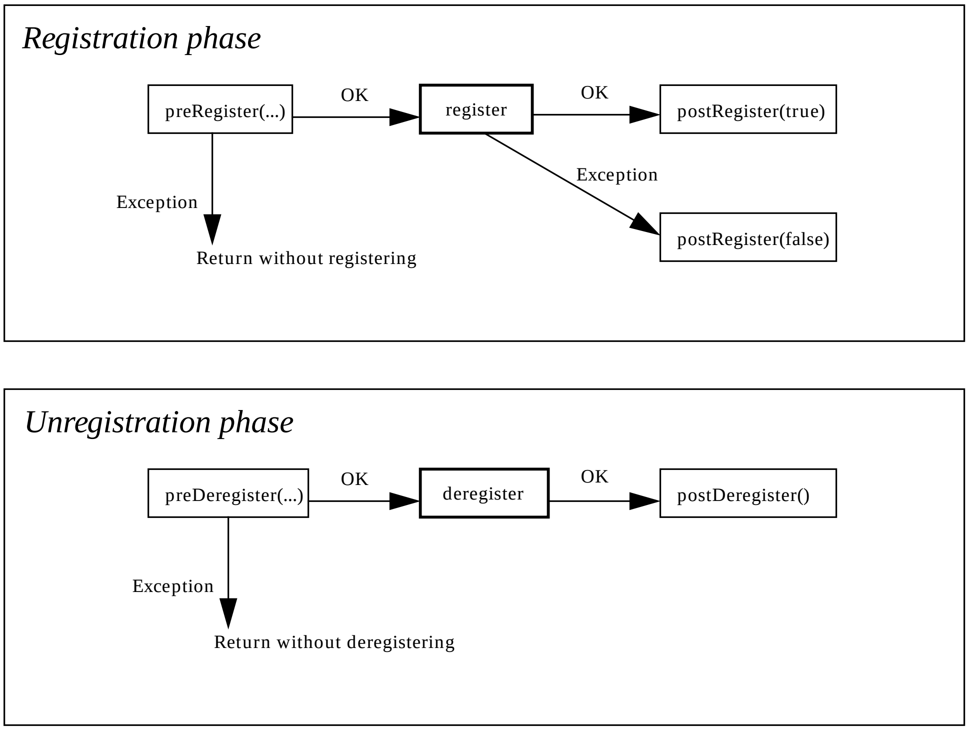


FIGURE 7-1 Calling Sequence for the MBean Registration Methods

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 pattern matching on their names, and optionally by means of a filter applied to their attribute values. Such a filter can be constructed by using the query expressions defined in “Queries” on page 129.
- 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 that determines the MBean on which the operation is 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.

Calling a method in the MBean server requires an appropriate permission. Permissions are described in Chapter 12, “Security”.

The detailed description of all MBean server operations is given in the API documentation generated by the Javadoc tool.

MBean Proxies

As an alternative to calling the generic methods of the `MBeanServer` interface directly, code that accesses a specific MBean can construct a *proxy* for it. A proxy is a Java object that implements the same interface as the MBean itself. A method in this interface on the proxy is routed through the MBean server to the MBean.

It is simpler and less error-prone to use proxies where possible rather than calling the methods of the MBean server directly.

Proxies are constructed using the class `javax.management.MBeanServerInvocationHandler`. The Javadoc for that class explains in detail how to construct and use them.

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 that it represents.

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

The delegate object provides the following information about the MBean server, all of which is 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 identifier for the MBean server, for example, based on the host name 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 "`1.2 Maintenance Release`".
- 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 are given by the implementor.
- The `ImplementationVersion` gives the implementation version of the MBean server. The format and contents of this string are given by the implementor.
- The `ImplementationVendor` gives the vendor name of the MBean server implementation. The contents of this string are given by the implementor.

The `MBeanServerDelegate` class implements the `NotificationBroadcaster` interface and sends the `MBeanServerNotifications` that are emitted by the MBean server. For objects to receive these notifications, they must register with the delegate object (see "MBean Server Notifications" on page 128).

Note – The "`JMImplementation`" domain name is reserved for use by JMX Agent implementations. The `MBeanServerDelegate` MBean cannot be unregistered 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 is 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 is able to perform operations on the MBeans through the corresponding connector client, once a connection is established.

Typically, a remote client is able to perform a subset of the operations in the `MBeanServer` interface, through that interface's parent `MBeanServerConnection`. Because remote connections can fail, each method in `MBeanServerConnection` declares `IOException` in its throws clause. See "The MBeanServerConnection interface" on page 134.

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, that acts as a remote representation of the MBean server. This type of dynamic invocation is typically used in conjunction with the MBean introspection functionality that 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 that 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.

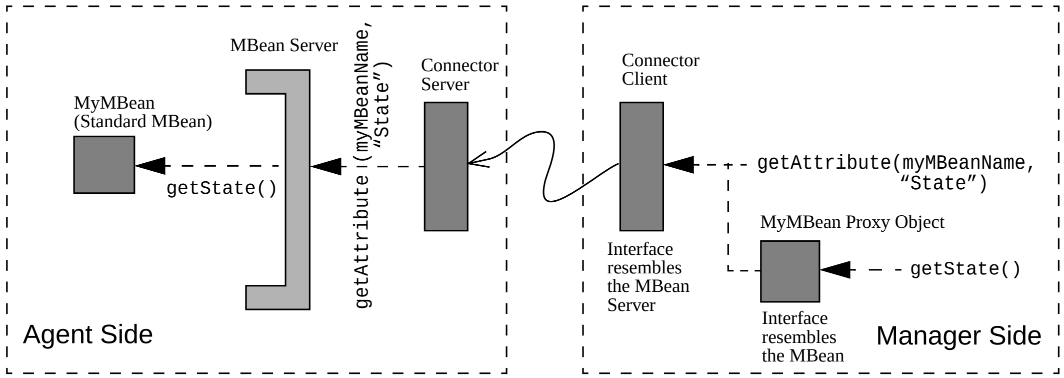

MBean Server
Connector Server
Connector Client
MyMBean (Standard MBean)
getState()
getAttribute(myMBeanName, "State")
getAttribute(myMBeanName, "State")
MyMBean Proxy Object
getState()
Interface resembles the MBean Server
Interface resembles the MBean
Agent Side
Manager Side

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, that contains a list of object names involved in the operation.

The MBean server object does not broadcast notifications itself: its unique delegate MBean implements the `NotificationBroadcaster` interface to 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 126).

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

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

- `JMX.mbean.registered` - This notification indicates that one or more MBeans have been registered. The notification will convey the list of object names of these MBeans.
- `JMX.mbean.unregistered` - This notification indicates that one or more MBeans have been unregistered. The notification conveys the list of these MBeans' object names .

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 52.

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 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 that are in the given scope, or that satisfy the given query expression, or both, 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 can also return the delegate MBean if its object name is within the scope and if it satisfies the query expression, if any (see “MBean Server Delegate MBean” on page 126).

Scope of a Query

The scope is defined by an object name pattern: see “Pattern Matching” on page 111. Only those MBeans whose object name matches the pattern are considered in the query. The query expression must then 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 can then be associated by relational operators (`and`, `or`, and `not`) to form complex expressions involving several MBean attributes.

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.

If the evaluation of the query expression for a given MBean in the scope results in an exception, that MBean is omitted from the query result. The exception is not propagated to the caller of `queryMBeans` or `queryNames`. Errors (subclasses of `java.lang.Error`) can be propagated, however.

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.

- The `ObjectName` class (see “ObjectName Class” on page 109) implements the `QueryExp` interface and can be used within queries. Usually, an `ObjectName` in a query will be a pattern. When an `ObjectName` query expression is being evaluated for a given MBean, the expression is true if the MBean’s name matches the `ObjectName` pattern.

In practice, users do not instantiate the `ValueExp` and `QueryExp` implementation classes directly. Instead, they 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 can be used as part of a constraint, as described:

- `classattr` - The result represents the class name of the MBean and can only be used in a string constraint.
- `attr` - The result represents the value of the named attribute. This result can be used in boolean, numeric or string constraints, depending upon the type of the attribute. Attributes can also be constrained by the values of other attributes of an equivalent type. This method is overloaded to take a class name too: this is equivalent to setting a constraint on the name of the MBean’s class.

  If an MBean in the scope does not have an attribute with the given name, or if there is a class name parameter and it does not match the MBean’s class, then the MBean is omitted from the query result.
- `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 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 the 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. Constraints 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. The second subexpression is not evaluated if the first one is false.
- `or` - The resulting expression is the logical OR of the two subexpression arguments. The second subexpression is not evaluated if the first one is true.
- `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 is built as follows. When constructing constraints on string values, the asterisk (`*`) is a wildcard character that can replace any number of characters, including zero. Alternatively, the programmer can 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 can 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 can 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

Performing queries can result in some exceptions that are specific to the filtering methods. If the evaluation of a query for a given MBean produces one of these exceptions, the MBean is omitted from the query result. Application code will not see these exceptions in usual circumstances. Only if the application itself throws the exception, or if it calls `QueryExp.apply`, will it see these exceptions.

`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 a class name to an MBean of the wrong class.

`MBeanServerConnection` Interface

The JMX 1.2 specification introduces a new interface `MBeanServerConnection`, the parent interface of `MBeanServer`. The purpose of this interface is to provide a common type to be used for access to an MBean server regardless of whether it is remote, namely, accessed through a connector, or local, and accessed directly as a Java object.

The `MBeanServerConnection` interface is similar to `MBeanServer`, but with two key differences:

- It omits the following methods that are only appropriate for local access to the MBean server:
  - `instantiate`. This method is useful to create instances of parameters to MBean methods or constructors when they are of classes unknown to the caller's class loader. But a remote client's class loader must know the classes to be able to deserialize them.
  - `registerMBean`. This method registers a local object as an MBean within the MBean server. It does not make sense to register a remote object in this way.
  - `getClassLoader`, `getClassLoaderFor`, `getClassLoaderRepository`. These methods are useful for the server end of a connector. (See "Using the Correct Class Loader for Parameters" on page 146.) Class loaders and the Class Loader Repository are not serializable in general, so they could not be transmitted to a remote client. In any case, it is not appropriate for a remote client to be able to access this information.
  - `deserialize`. This method returns an `ObjectInputStream`, that is not a serializable class, so it could not be transmitted to a remote client. The method is only of use to the server end of a connector, and even there it is superseded by the `getClassLoader` (etc) methods.
- Each remaining method includes `java.io.IOException` in its `throws` clause.

Application code that interacts with MBeanServerConnection works with a local MBean server or with the client end of a connector, regardless of whether it is connected to the server or the connector.

Because all the methods of MBeanServerConnection can throw IOException, application code that calls them must be prepared to deal with this exception and handle it appropriately. In the case of a local MBean server, these exceptions cannot happen, but the code must handle them anyway. This is the price to pay for operating the same way in both the local and remote cases.

Changing the MBean Server Implementation

As of version 1.2 of the JMX specification, the system property javax.management.builder.initial can be set to replace the default implementation of the MBeanServer interface with a different implementation. When the createMBeanServer or newMBeanServer method of the MBeanServerFactory class is called, it consults this property. If a value exists, then it must name a public class that is a subclass of javax.management.MBeanServerBuilder. The class is instantiated and used to create an MBeanServer instance.

An MBeanServerBuilder must be able to create an instance of MBeanServerDelegate and an instance of MBeanServer. The MBeanServerDelegate can be the standard javax.management.MBeanServerDelegate, or a custom subclass, for example, to override the ImplementationName attribute. The MBeanServer can be a complete reimplementation of the MBeanServer interface, or it can build on the standard implementation by instantiating javax.management.MBeanServerBuilder, calling its newMBeanServer method, and wrapping the resulting object in another MBeanServer object.

CHAPTER 8

Advanced Dynamic Loading

This chapter describes the dynamic loading services that build on Java's class loader functionality to provide the ability to retrieve and instantiate MBeans using new Java classes and possibly native libraries. The origin of these classes and libraries is not necessarily known when the MBean server is deployed, and can include code loaded from a remote server.

Dynamic loading is usually performed by the management applet (m-let) service that is used to instantiate MBeans obtained from a remote URL (Universal Resource Locator) on the network.

The Java Management extensions (JMX) specification also defines lower-level mechanisms for class loading, that allow developers to extend the functionality of the m-let service or to load classes without it.

This chapter describes mandatory functionality for all compliant JMX agents.

Overview of M-Lets

The m-let service allows you to instantiate and register one or more MBeans from a remote URL, in the MBean server. The m-let service does this by loading an m-let text file, that specifies information on the MBeans to be obtained. The information on each MBean is specified in a tag similar to those used in XML, 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 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.

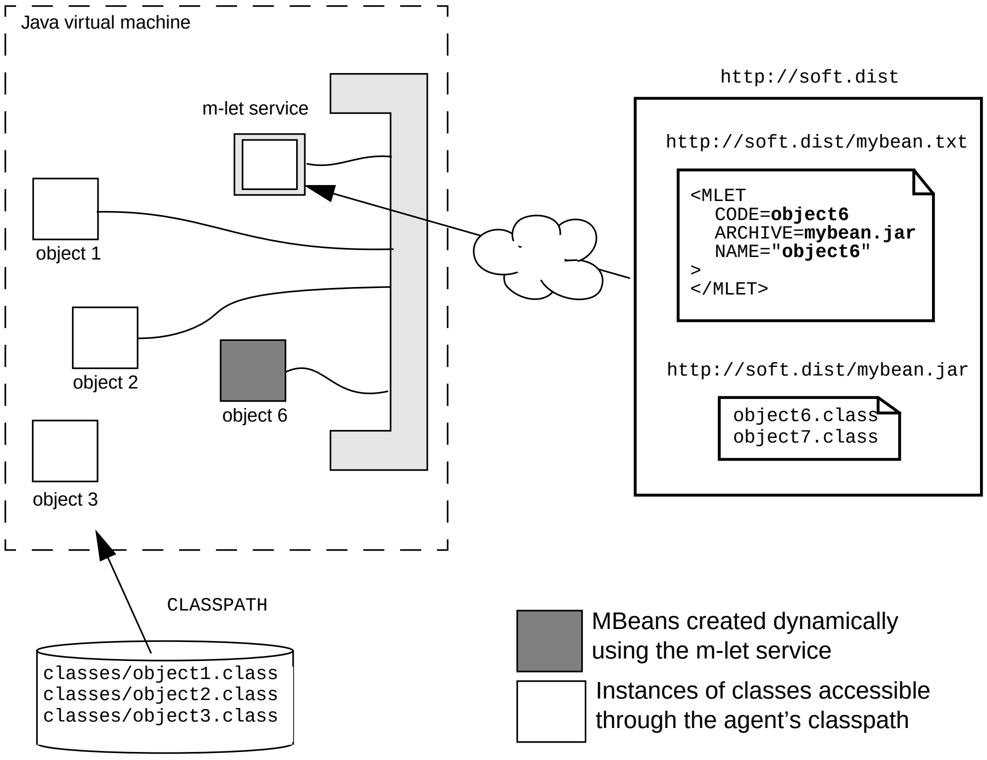


FIGURE 8-1 Operation of the M-Let Service

The `MLET` Tag

The m-let file can 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 next one 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, that 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. This attribute is used when 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 taken as the code base URL.

- `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 decimal point (`.`), 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 looks 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`, that contains the methods exposed for remote access. This implies that the m-let service is itself an MBean and can 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 138. 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 were not.

Other methods of the `MLet` class manage the directory for native libraries downloaded in JAR files and used by certain MBeans. See the API documentation generated by the Javadoc tool 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 can specify a number of `MLET` tags to populate all 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 API documentation generated by the Javadoc tool). Because 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.

Because the `createMBean` methods of the `MBeanServer` interface take the object name of the class loader, this functionality is also available to remote management applications that 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 to 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.

The Class Loader Repository

The MBean server maintains a list of class loaders in the *class loader repository*. The class loader repository is also sometimes referred to as the default loader Repository.

The class loader repository is used in the following circumstances:

- In the `createMBean` and `instantiate` methods from the MBeanServer interface. The class loader repository is used to find and load the class named by the *`className`* parameter to these methods.

  These methods exist in several overloaded forms. Some of the forms have an `ObjectName` parameter specifying an MBean that is a class loader. The class loader repository is not used by those forms.
- When an m-let does not find a class in its URLs, it tries to load the class through the class loader repository. This behavior can be disabled when the m-let is created.
- The method `getClassLoaderRepository` from the `MBeanServer` interface provides a way for clients of the MBean server to access its class loader repository directly.

If several MBean servers are created within the same Java Virtual Machine, for example by a program that calls `MBeanServerFactory.createMBeanServer` several times, each one has its own class loader repository, independently of the others.

How to Add Loaders to the Class Loader Repository

When an MBean server is created, its class loader repository contains the class loader that was used to load the `MBeanServer` implementation class. Thereafter, a class loader is added to the repository if it is registered as an MBean. If the MBean is subsequently unregistered, it is removed from the repository.

Put another way, if an MBean is registered that is a descendant of `java.lang.ClassLoader`, it is added to the class loader repository.

If an MBean is a descendant of `java.lang.ClassLoader` but implements the interface `javax.management.loading.PrivateClassLoader`, then it is never added to the class loader repository.

Because the class `javax.management.loading.MLet` is a descendant of `java.lang.ClassLoader`, m-lets are added to the class loader repository when they are registered in the MBean server. The JMX specification includes a class `PrivateMLet` that subclasses `MLet` and implements `PrivateMLet`. A `PrivateMLet` behaves like an `MLet` in every way except that it is never added to the class loader repository.

Order of Loaders in the Class Loader Repository

The order of class loaders in the repository is significant. When a class is loaded using the repository, each class loader in turn is asked to load the class. If a loader successully loads the class, the search stops. If a loader throws `ClassNotFoundException`, the search continues with the next loader in the list. If no loader succeeds in loading the class, the attempt results in a `ClassNotFoundException`.

The first loader in the class loader repository is the one that was used to load the `MBeanServer` implementation class. Thereafter, each entry is an MBean that is a descendant of `java.lang.ClassLoader`. The order of these loaders is the order in which the MBeans were registered.

More formally, an MBean *m1* appears before an MBean *m2* if the `createMBean` or `registerMBean` operation that registered *m1* completed before the operation that registered *m2* started. If neither operation completed before the other started, both MBeans were registered at the same time in different threads, and the order between *m1* and *m2* is indeterminate.

M-Let Delegation to the Class Loader Repository

An m-let is a class loader, and as such follows the standard behaviour for a class loader when it loads a class using its `loadClass` method:

- First, it sends a request to its *parent class loader* to load the class. The parent class loader is specified when the m-let is created. By default, it is the *system class loader*.
- If the parent class loader is unable to load the class, the m-let attempts to load it itself through its list of URLs. The `MLet` class is a subclass of `java.net.URLClassLoader`, and the behavior for loading a class through a list of URLs is inherited from `URLClassLoader`.

If neither of these two attempts finds the class, the m-let attempts to load the class through the class loader repository. We say that it *delegates* to the class loader repository. Only if that attempt also fails does the m-let throw a `ClassNotFoundException`.

When an m-let delegates to the class loader repository, each loader in the repository is asked in turn to load the class. However, if the m-let is itself in the class loader repository, the search stops as soon as the m-let is reached. That is, the m-let only delegates to loaders that precede it in the repository.

The class loader repository can be used as a way to make common classes available to MBeans from different sources. The common classes are placed in the repository, and m-lets that delegate to the repository can find them. Because m-lets only delegate to loaders that precede them in the repository, the order in which loaders are registered is important. If m-let *m1* defines classes that are used by classes in m-let *m2*, then *m1* must be registered before *m2*.

When an m-let is created, it is possible to control both whether it delegates to other loaders via the repository, and whether other loaders delegate to it:

- If the m-let is constructed with the Boolean *delegateToCLR* parameter false, then it will not delegate to the class loader repository.
- If the m-let is an instance of `PrivateMLet`, then it will not be added to the class loader repository, so other loaders will not delegate to it.

New Semantics in the JMX 1.2 Specification

In versions of the JMX specification prior to 1.2, m-lets delegated to the complete list of loaders in the class loader repository. That is, if an m-let did not find a class itself, every other loader in the repository was consulted. Loaders were consulted regardless of whether they were before or after the m-let in the repository.

This behaviour is open to a subtle problem with certain Java Virtual Machines. Note that a class is not necessarily loaded by an explicit call to the `loadClass` method of some class loader. More often, a class is loaded because it is referred to by another class, for instance because it is the type of a field in that class, or of a parameter or return value in a method, or it is a superclass or superinterface of the class, or one of the methods in the class constructs an instance of it or refers to a static field or method in it. Simplifying slightly, we can say that the exact moment when a class is loaded cannot be predicted.

When a class *c1* refers to another class `A` for the first time, `A` is loaded using *c1*'s class loader. If *c1* was loaded by the m-let *m1*, then `A` will also be loaded using *m1*.

If *m1* does not find the class `A` through its parent class loader, or through its list of URLs, it will delegate to the class loader repository.

Referring to FIGURE 8-2, imagine that at the same time in another thread, a class *c2*, loaded by the m-let *m2*, refers to the class `B` for the first time. Again, `B` will be loaded using *m2*, and if *m2* does not find the class itself, it will delegate to the class loader repository.

If *m1* searches through all the loaders in the repository besides itself, and *m2* does likewise, then *m1* will end up sending a request to *m2* to load `A` and *m2* will end up sending a request to *m1* to load `B`.

A problem arises with certain Java Virtual Machine implementations that do not allow more than one thread at a time to load a class through a given class loader. Because thread 1 is loading class `A` through *m1*, thread 2 cannot simultaneously load class `B` through *m1*. Because thread 2 is loading class `B` through *m2*, thread 1 cannot simultaneously load class `A` through *m2*. Each thread must wait for the other to finish before it can proceed, creating a classical deadlock situation.

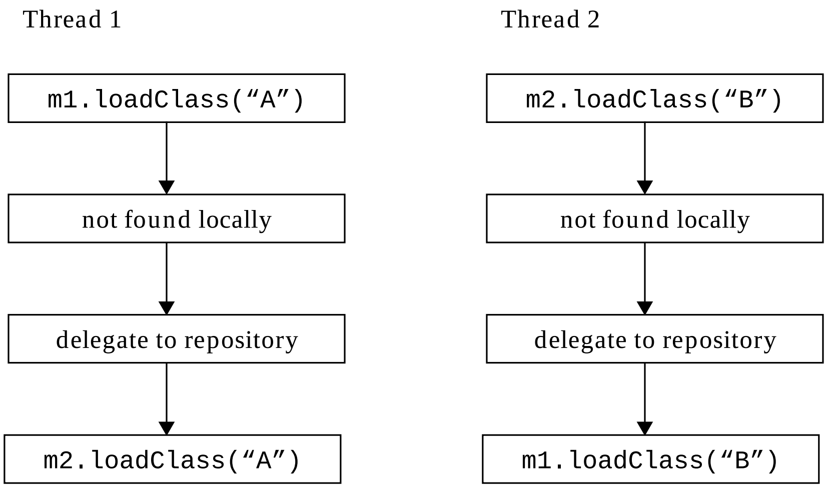


FIGURE 8-2 Deadlock scenario for m-let delegation

The change in semantics avoids this scenario because one of the two m-lets must appear after the other in the class loader repository. If *m2* appears after *m1*, *m1* will never attempt to load a class using *m2*, because it only delegates to loaders that appear earlier than it in the repository. So the deadlock cannot happen.

If you are prepared to run the risk of deadlock, or you are sure that a scenario such as the above cannot happen, it is straightforward to subclass `MLet` and override its `loadClass` method to restore the previous semantics of delegating to all the loaders in the repository, whether before or after the m-let. However, you should remember that if a class loaded by such an `MLet` subclass refers to another class that does not exist, *all* the m-lets in the class loader repository are consulted before `ClassNotFoundException` is thrown.

Using the Correct Class Loader for Parameters

A subtle pitfall of class loading is that the class `a.b.C` created by the class loader *cl1* is not the same as the class `a.b.C` created by the class loader *cl2*. Here, "created" refers to the class loader that actually creates the class with its `defineClass` method. If *cl1* and *cl2* both find `a.b.C` by delegating to another class loader *cl3*, it is the same class.

A value of type "`a.b.C` created by *cl1*" cannot be assigned to a variable or parameter of type "`a.b.C` created by *cl2*". An attempt to do so will result in an exception such as `ClassCastException`.

For the JMX specification, this can pose problems for the parameters of the methods `createMBean`, `invoke`, `setAttribute`, and `setAttributes` of the MBean server.

Suppose you have an MBean with the following interface:

CODE EXAMPLE 8-1 Simple MBean interface

```
public interface AnMBean {
    public void m(SomeClass x);
}
```

If the MBean server contains an instance of this MBean that was created by the class loader *cl1*. At some stage, either during loading or the first time it is referenced, *cl1* will load the class `SomeClass`.

Suppose now that you are writing a connector. At the receiving (server) end of the connector, you get a request to invoke `m` on the MBean. The sender will have sent an instance of `SomeClass` that you must recreate, for example, by deserializing it. If you recreate it with any class loader other than *cl1*, you will get an exception when you try to pass it to the method `m`.

This means that your connector server must have a way of instantiating received objects using the correct class loader.

To this end, there are three methods in the `MBeanServer` interface related to class loading:

- `getClassLoaderFor`
- `getClassLoaderRepository`
- `getClassLoader`

These methods are described in the following sections.

`getClassLoaderFor`

The appropriate method for CODE EXAMPLE 8-1 is:

```
public ClassLoader getClassLoaderFor(ObjectName name);
```

By calling this method with the `ObjectName` of an MBean, you can obtain the class loader that created the MBean's class, *cl1* in our example. Then you can get the correct class, `SomeClass`, using *cl1*.

The `getClassLoaderFor` method is appropriate for the `invoke`, `setAttribute`, and `setAttributes` operations, because the appropriate class loader for the parameters to these operations is the class loader of the target MBean.

The `getClassLoaderFor` method was introduced in version 1.2 of the JMX specification. Previously, connector servers had to use one of the `deserialize` methods in the `MBeanServer` interface. These methods are now deprecated. Consequently, use `getClassLoaderFor` instead of

`deserialize(ObjectName name, byte[] data)`.

`getClassLoader` and `getClassLoaderRepository`

For the `createMBean` operation, the target MBean does not exist, because the whole purpose of the operation is to create it. There are two classes of `createMBean` operation, depending on whether there is a *loaderName* parameter that specifies the name of an MBean that is a class loader to be used to load the MBean class.

- For a `createMBean` operation that does not include a *loaderName*, the MBean class is loaded using the class loader repository. If the constructor invoked to create the MBean has parameters, then these should also be loaded using the class loader repository, to avoid the problem that was explained above for `invoke`. Thus, the `MBeanServer` interface contains a method:

  ```
  public ClassLoaderRepository getClassLoaderRepository();
  ```

  The method `loadClass` in the `ClassLoaderRepository` interface can be used to load classes through the class loader repository.

- For a `createMBean` operation that includes a *loaderName*, the MBean class is loaded using the class loader that is registered as an MBean with the given name. If the constructor has parameters, their classes should be loaded by the same class loader. Thus, the `MBeanServer` interface contains a method:

  ```
  public ClassLoader getClassLoader(ObjectName name);
  ```

Similar considerations apply to the `instantiate` methods of the `MBeanServer` interface. However, these methods are not usually exposed through connectors.

The methods `getClassLoader`, and `getClassLoaderRepository` were introduced in the JMX specification version 1.2. Previously, connector servers had to use one of the three `deserialize` methods in the `MBeanServer` interface. These methods are now deprecated.

Consequently, use `getClassLoaderRepository` instead of

```
deserialize(String className, byte[] data)
```

Also use `getClassLoader` instead of

```
deserialize(String className, ObjectName loaderName, byte[] data)
```

CHAPTER 9

Monitoring

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

Monitoring services are a mandatory part of agents that comply with the JMX specification, and they must be implemented in full.

Overview

Using a monitoring service, the *observed attribute* of one or more other MBeans (the *observed* MBeans) is monitored at intervals specified by the *granularity period*. For each observed MBean, 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 of conditions. The conditions are specified when the monitor is initialized, or dynamically through the monitor MBean's management interface. Monitors can also send notifications when certain error cases are encountered while monitoring an attribute value.

Types of Monitors

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

- `CounterMonitor` - Observes attributes with Java integer types (`Byte`, `Integer`, `Short`, `Long`) that behave like a counter, namely:

    - Their value is always greater than or equal to zero.
    - They can only be incremented.
    - They can roll over, and in that case a *modulus* value is defined.
- `GaugeMonitor` - Observes attributes with Java integer or floating point types (`Float`, `Double`) that behave like a gauge (arbitrarily increasing and decreasing).
- `StringMonitor` - Observes an attribute of type `String`.

All types of monitors extend the abstract `Monitor` class, that defines common attributes and operations. The type of the observed attribute must be supported by the specific monitor subclass used.

However, monitors verify the type of the object instance that is returned as the attribute's value, not the attribute type declared in the observed MBean's metadata. For example, this allows a string monitor to observe an attribute declared as an `Object` in its metadata, as long as actual values of the attributes are `String` instances.

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 trigger conditions of a monitor 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 MBean 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 MBean's object name
- The observed attribute name
- The derived gauge, namely, the last value computed from the observation
- The threshold value or string that triggered this notification

The tree representation of all notification types that can be generated by the monitoring services is given in FIGURE 9-1. The error types are common to all monitors and are described below. Each of the threshold events is particular to its monitor and is described in the corresponding section.

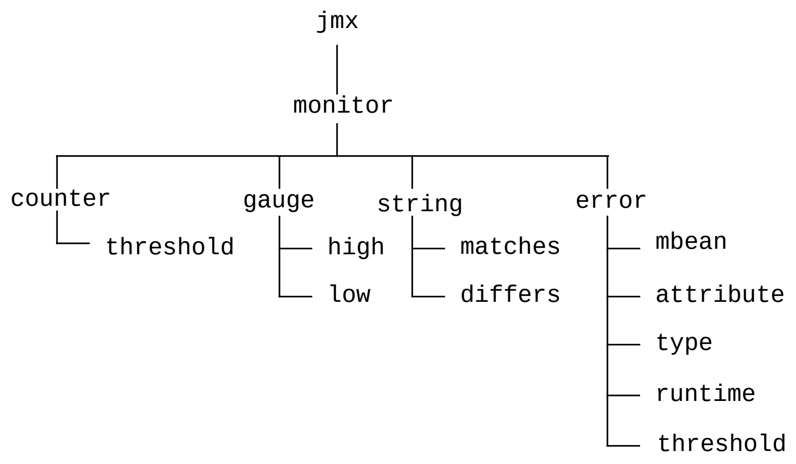


FIGURE 9-1 Tree Representation of Monitor Notification Types

Common Monitor Notification Types

The following notification types are common to all monitors and are emitted to reflect error cases. The first measurement is made when the monitor is started:

- `jmx.monitor.error.mbean` - Sent when one of the observed MBeans 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 one of the observed objects. The *observed object name* and *observed attribute name* are provided in the notification.
- `jmx.monitor.error.type` - Sent when the object instance of the observed attribute value is `null` or 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 comparison level known as the *threshold*.

The counter can *roll over* (also known as *wrapping around*) when it reaches a maximum value. In this case, the notification is triggered every time the counter reaches or exceeds the threshold, provided it has been observed below the threshold because the previous notification. A counter does not necessarily roll over to zero, but this does not affect the monitor that handles the general case.

In addition, an *offset* mechanism can detect counting intervals, as follows:

- The offset mechanism is enabled whenever the monitor's offset value is non-zero.
- Whenever the monitor detects that the counter reaches or exceeds the threshold, a notification is triggered and the threshold is incremented by the offset value. The threshold is incremented by the offset value as many times as necessary for the threshold to exceed the counter value again, but still only one notification is sent.
- If the counter that is monitored rolls over when it reaches a maximum value, then the *modulus* value needs to be set to that maximum value. The threshold will then also "roll over" whenever it strictly exceeds the modulus value. When the threshold "rolls over", it is reset to the value that was specified through the latest call to the monitor's `setInitThreshold` method, before any offsets were applied.
- The `getThreshold` method of the `CounterMonitor` class always returns the current value of the threshold, that includes any offset increments.
- All incrementing or rolling over of the threshold is considered to take place instantaneously, namely, before the count is incremented. Thus, if the granularity period is set appropriately, the monitor triggers a threshold notification every time the count increases by an interval equal to the offset value.

If the counter difference option is used, then the value of the derived gauge is computed as the difference between the observed counter values for two consecutive observations. If the counter will roll over, then the modulus must be defined when counter difference is active. When the counter does roll over, the difference between the observations will be negative and value of the modulus needs to be added.

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

- While `t` is less than `StartDate+2GP`, `V[t] = (Integer)0`

- If `(counter[t] - counter[t-GP])` is greater than or equal to zero, 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 integer type as the observed attribute.

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

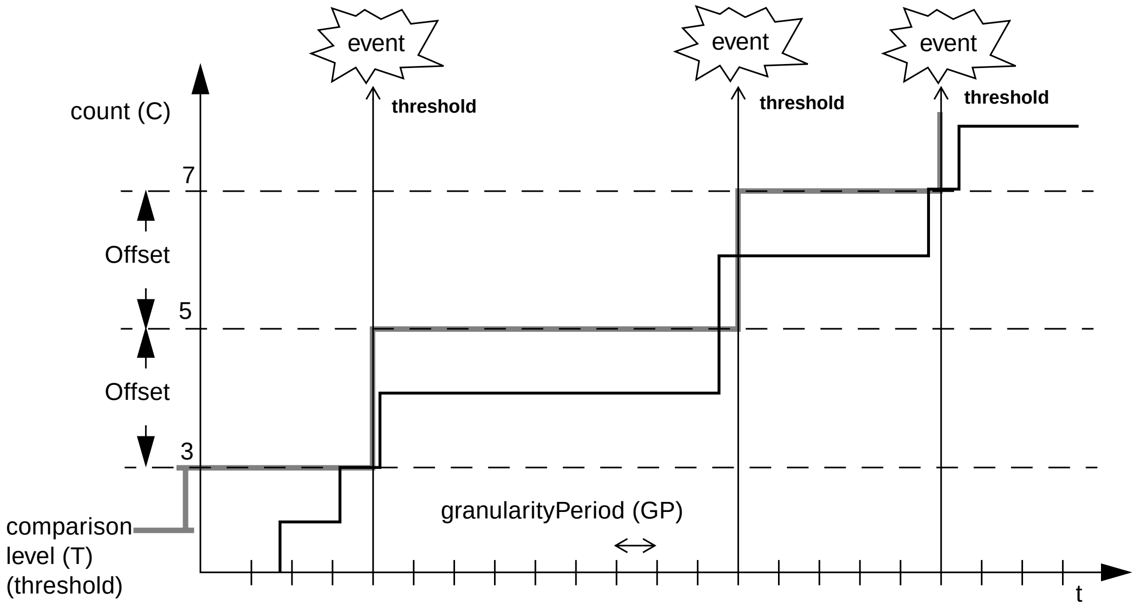


FIGURE 9-2 Operation of the `CounterMonitor`

The monitor observes a counter `C(t)` that 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 threshold 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 can 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 that 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 to trigger a notification that will be broadcast only if the `NotifyHigh` boolean attribute is `true`.
- The `LowThreshold` attribute defines the value that the gauge must fall to or fall below to trigger a notification that 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.

- Upon creation of the gauge, if `NotifyHigh` is `true` and the gauge value is already equal to or greater than the `HighThreshold` value, then the defined notification is triggered. Similarly, if `NotifyLow` is `true` and the gauge value is already equal to or less than the `LowThreshold` value, then the defined notification is also triggered.

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 consecutive observations.

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

- While `t` is less than `StartDate+2GP`, `V[t] = (Integer)0`
- Otherwise, `V[t] = gauge[t] - gauge[t-GP]`

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

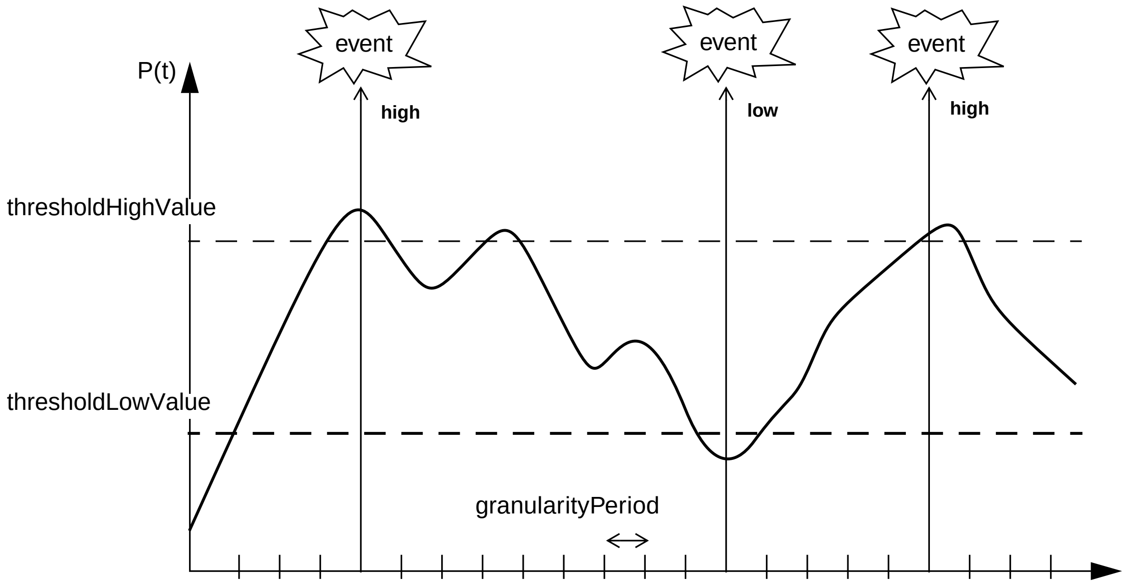


FIGURE 9-3 Operation of the `GaugeMonitor`

Gauge Monitor Notification Types

In addition to the monitor error notification types, a `GaugeMonitor` MBean can 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 of an attribute of type `String`. The derived gauge in this case is always 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 subsequent observation times (defined by the granularity period), no other notification will be sent for as long as the attribute value still 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 subsequent observation times, 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”.

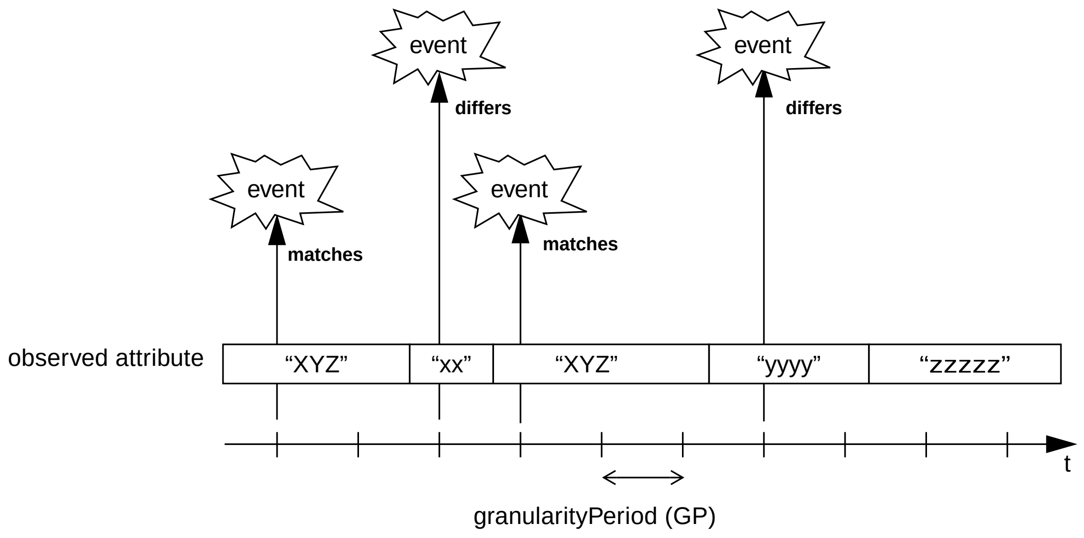


FIGURE 9-4 Operation of the `StringMonitor`

String Monitor Notification Types

In addition to the monitor error notification types, a `StringMonitor` MBean can 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 package diagram of the various monitor MBean classes, with the interfaces they implement. The API documentation generated by the Javadoc tool provides the complete description of all monitoring service interfaces and classes.

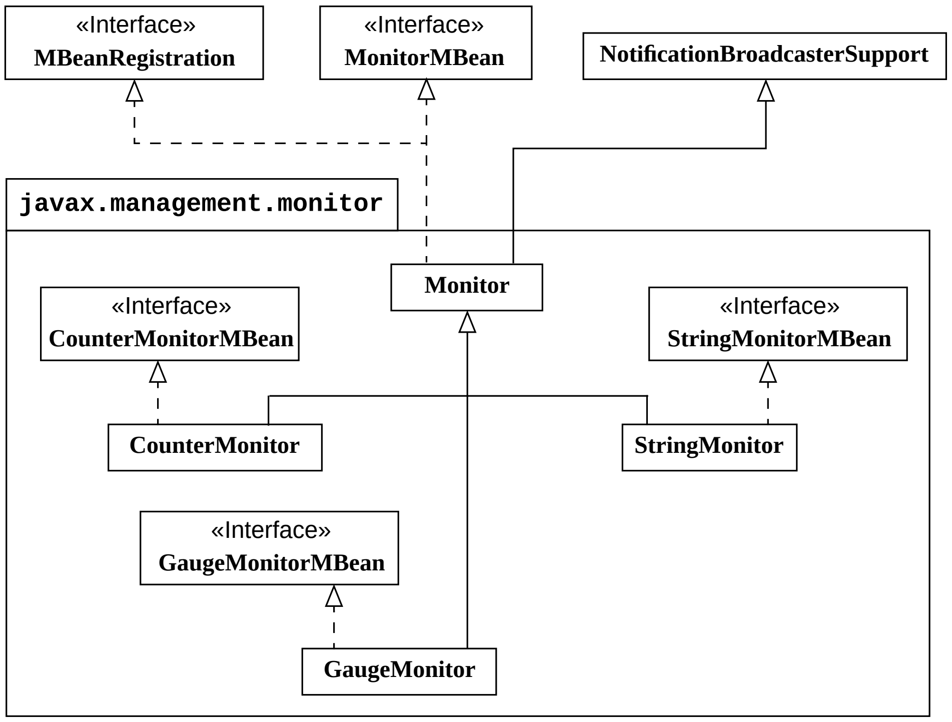


FIGURE 9-5 The Package and Class Diagram of the Monitor MBeans

CHAPTER 10

Timer Service

The timer service triggers notifications at specific dates and times. It can 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 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 service: 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. All notifications broadcast by the timer service are instances of the TimerNotification class.

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

Adding Notifications to the Timer

The timer service maintains an internal list of the dated notifications that it has been requested to send. Notifications are added to this list using the Timer class' addNotification methods. The methods take the following parameters, 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 Timer class includes integer constants for expressing durations in milliseconds, that can then be used to create java.util.Date objects.

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 [sic] - The total number of times that the notification will occur. If the value of this parameter 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 behaves as if the current date had been specified. Updating the date of a timer notification that is being added *does not* generate any notification events, as opposed to the sendPastNotifications mechanism that applies when the timer is started (see "Starting and Stopping the Timer" on page 162).

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

When a one-off notification (period is zero or null) occurs, it is removed from the timer's list of notifications.

Receiving Timer Notifications

The timer service MBean is a standard notification broadcaster with notification types and times defined by the list of notifications built up through the `addNotification` method. All listeners of a given timer MBean will receive all its timer notifications. Listeners configured to listen for a specific timer notification should specify the appropriate filter object when registering as a listener (see "NotificationFilter Interface" on page 51).

When the timer is active and the date of a timer notification comes due, the timer service broadcasts this notification with the given type, message, and user data, along with the notification's identifier within the timer. If a periodic notification has a specified number of occurrences, that number is decremented by one. Accessing the occurrence parameter of a timer notification always returns the remaining number of occurrences at the time of access.

When a notification is not repeating or when it has exhausted its number of occurrences, it is removed from the timer's list of notifications. The methods of the `Timer` class for accessing notification parameters will raise an exception if called with the identifier of a timer notification that has been sent and removed.

Removing Notifications From the Timer

Timer notifications can also be removed from the list of notifications using one of the following methods of the `Timer` class:

- `removeNotification` - Takes a notification identifier as a parameter and removes the corresponding notification from the list. If the specified identifier does not exist in the list, this method throws an `InstanceNotFoundException`.
- `removeNotifications` - Takes a notification type as a parameter and removes all notifications from the list that were added with that type. If the specified notification type does not correspond to any notifications in the list, this method throws an `InstanceNotFoundException`.
- `removeAllNotifications` - Empties the timer's list of notifications. This method also resets the notification identifiers, meaning that all existing identifiers for this timer are invalid and will erroneously refer to new notifications.

Starting and Stopping the Timer

The timer service, represented by an instance of the `Timer` class, is activated using the `start` method and deactivated 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 are 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 date is greater than the current date. The number of increments can be limited by their defined number of occurrences. Notifications with one-time dates preceding the start date and notifications with a limited number of occurrences that 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 on the `sendPastNotifications` attribute of the `Timer` class:

- `sendPastNotifications` = `true` - All one-time notifications with a date before the start date are broadcast, and all periodic notifications will be broadcast as many times as they should have occurred before the start date, including those that are removed because they cannot be updated beyond the start date.
- `sendPastNotifications` = `false` - Notifications with a date before the start date are ignored; if a notification is periodic, its notification date is updated but no notifications are triggered.

Setting the `sendPastNotifications` flag to `true` can cause a flood of notifications to be broadcast when the timer is started. The default value for this flag is `false`. Setting this flag to `true` ensures that notification dates that occur while 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.

Calling `start` on a `Timer` that has already been started, or `stop` on a `Timer` that has already been stopped, has no effect. After a `stop`, the timer is in its initial state, and it can be started again with `start`.

CHAPTER 11

Relation Service

As part of the agent specification, the Java Management extensions (JMX) specification also defines 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 that 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 that satisfy the role. Through the relation service, users create new types and then create, update, or remove relations that satisfy 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 that 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 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 relation model in the JMX specification 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 participates in 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 satisfy, and it serves as a template for creating and maintaining relations.
relation	A current association between MBeans that satisfies 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 can also have properties and methods that operate on its MBeans.
role value	The list of MBeans that currently satisfies 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, as defined by 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 that 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.sub

To represent this relation in the JMX specification 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`, representing the concept of book ownership.

The following diagram represents this sample relation, as compared to the UML modeling of its corresponding association.

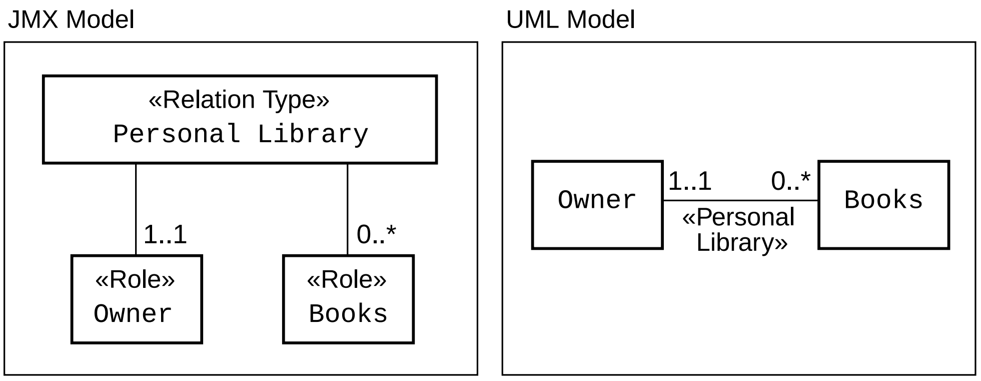


FIGURE 11-1 Comparison of the Relation Models

In the JMX specification model, the relation type is a static set of roles. Relation types can be defined at runtime, 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, can 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 cannot 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, such as the identifier of the relation and the new role values. The notification also indicates whether the relation was internally or externally defined (see "External Relations" on page 168).

There is a difference between the two models presented in FIGURE 11-1. The UML association implies that each one of the `Books` can only have one owner. The relation type in the JMX specification 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 specification relation model only guarantees that an MBean satisfies its designated role, it does not allow a user to define how many relations an MBean can appear in. This implies that the relation service does not perform inter-relation consistency checks. These are the responsibility of the management application when creating or modifying relation instances.

If this level of consistency is needed, the designer of a management solution must implement the necessary verifications in the objects that use the relation service. In our example, the designer would need to ensure that the same book MBean is not added to more than one `Personal Library` relation. One way to do this is by calling the query methods of the relation service before performing any operation.

Implementation

The JMX specification defines the Java classes the behavior of which implements this relation model. Each of the concepts defined in "Terminology" on page 164 has a corresponding Java class (see FIGURE 11-2). Along with the behavior of the relation service object itself, these classes determine how the relation service is used in management solutions.

This section explains 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 API documentation generated by the Javadoc tool 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 178). 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.

javax.management.relation

«relation service»

RelationService
RelationServiceMBean
RelationNotification
MBeanServerNotificationFilter

«relation support»

RelationType
RelationTypeSupport
Relation
RelationSupport
RelationSupportMBean

«role description»

RoleInfo
Role
RoleList
RoleUnresolved
RoleUnresolvedList
RoleResult
RoleStatus

«exception superclass»

RelationException

«relation type creation errors»

InvalidRoleInfoException
InvalidRelationTypeException

«relation creation errors»

InvalidRelationServiceException
RelationServiceNotRegistered-Exception
RoleInfoNotFoundException
InvalidRoleValueException
RelationTypeNotFoundException
InvalidRelationIdException

«relation access errors»

RelationNotFoundException
RoleNotFoundException

FIGURE 11-2 Classes of the `javax.management.relation` package

On the other hand, relation types and relation instances are controlled by the relation service to maintain the consistency of the relation model. The implementation of the JMX specification 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 the relation service and added under its control. Users can access these objects 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 175 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 can be defined statically, for example, in a class constructor. 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 to create both internal and 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 171)

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 that defines the methods that the relation service uses 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 the behavior of which is thereafter undefined.

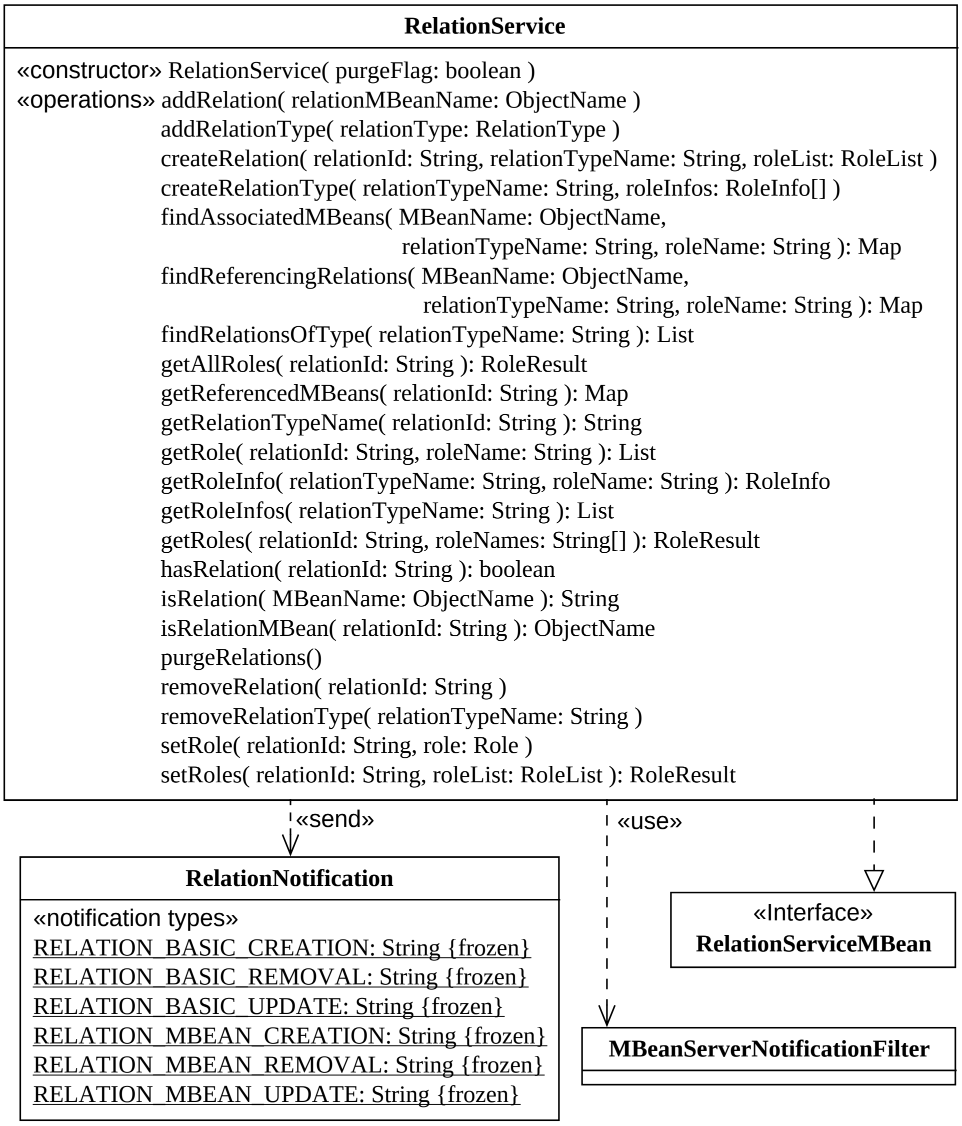
The major benefit of external relations is the ability to provide methods that return information about the relation’s members or even operate on the role values. Because the external relation is also an MBean, it can choose to expose these methods as attributes and operations.

Returning to “Example of a Relation” on page 164, the book ownership relation can 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` that all apply to the current members of the relation.

See “RelationSupport Class” on page 178 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.



RelationService
«constructor» RelationService(purgeFlag: boolean)
«operations» addRelation(relationMBeanName: ObjectName)
addRelationType(relationType: RelationType)
createRelation(relationId: String, relationTypeName: String, roleList: RoleList)
createRelationType(relationTypeName: String, roleInfos: RoleInfo[])
findAssociatedMBeans(MBeanName: ObjectName, relationTypeName: String, roleName: String): Map
findReferencingRelations(MBeanName: ObjectName, relationTypeName: String, roleName: String): Map
findRelationsOfType(relationTypeName: String): List
getAllRoles(relationId: String): RoleResult
getReferencedMBeans(relationId: String): Map
getRelationTypeName(relationId: String): String
getRole(relationId: String, roleName: String): List
getRoleInfo(relationTypeName: String, roleName: String): RoleInfo
getRoleInfos(relationTypeName: String): List
getRoles(relationId: String, roleNames: String[]): RoleResult
hasRelation(relationId: String): boolean
isRelation(MBeanName: ObjectName): String
isRelationMBean(relationId: String): ObjectName
purgeRelations()
removeRelation(relationId: String)
removeRelationType(relationTypeName: String)
setRole(relationId: String, role: Role)
setRoles(relationId: String, roleList: RoleList): RoleResult
«send»
«use»
RelationNotification
«notification types»
RELATION_BASIC_CREATION: String {frozen}
RELATION_BASIC_REMOVAL: String {frozen}
RELATION_BASIC_UPDATE: String {frozen}
RELATION_MBEAN_CREATION: String {frozen}
RELATION_MBEAN_REMOVAL: String {frozen}
RELATION_MBEAN_UPDATE: String {frozen}
«Interface»
RelationServiceMBean
MBeanServerNotificationFilter

FIGURE 11-3 Relation Service Classes

The relation service MBean is a notification broadcaster and the only object to send `RelationNotification` objects. 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 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 is identified by a name passed as a parameter and that 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 175).

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 is identified by an identifier passed as a parameter and that 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 165).

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 (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 in which 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 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 165). 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 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 can update its internal data.

`RelationNotification` Class

An instance of this class is created and broadcast as a notification when a relation is created, added, updated, or removed. It defines two separate notification types 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 can send (see FIGURE 11-3).

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 can be reused elsewhere.

Interfaces and Support Classes

External relation types and relation instances rely on the interfaces defined in the following figure and can choose to extend the support classes for convenience. Implementations of the JMX specification can also rely on these classes internally.

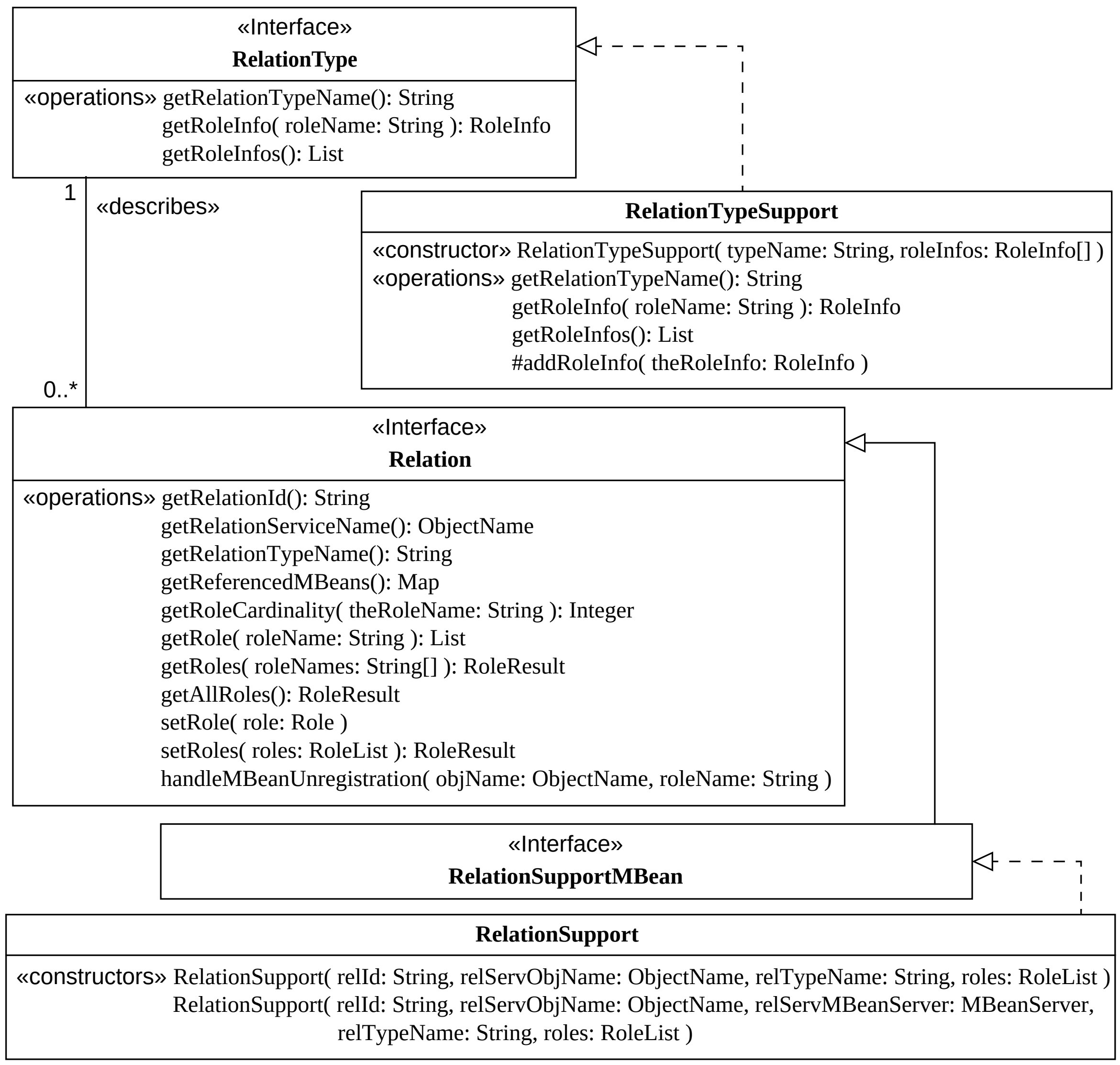

«Interface»
RelationType
«operations» getRelationTypeName(): String
getRoleInfo(roleName: String): RoleInfo
getRoleInfos(): List
1
«describes»
0..*
RelationTypeSupport
«constructor» RelationTypeSupport(typeName: String, roleInfos: RoleInfo[])
«operations» getRelationTypeName(): String
getRoleInfo(roleName: String): RoleInfo
getRoleInfos(): List
#addRoleInfo(theRoleInfo: RoleInfo)
«Interface»
Relation
«operations» getRelationId(): String
getRelationServiceName(): ObjectName
getRelationTypeName(): String
getReferencedMBeans(): Map
getRoleCardinality(theRoleName: String): Integer
getRole(roleName: String): List
getRoles(roleNames: String[]): RoleResult
getAllRoles(): RoleResult
setRole(role: Role)
setRoles(roles: RoleList): RoleResult
handleMBeanUnregistration(objName: ObjectName, roleName: String)
«Interface»
RelationSupportMBean
RelationSupport
«constructors» RelationSupport(relId: String, relServObjName: ObjectName, relTypeName: String, roles: RoleList)
RelationSupport(relId: String, relServObjName: ObjectName, relServMBeanServer: MBeanServer, relTypeName: String, roles: RoleList)

FIGURE 11-4 Interfaces and Support Classes

`RelationType` Interface

Any external representation of a relation type must implement the `RelationType` interface 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 179).

The relation service invokes 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 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 cannot be added after an instance of this class has been used to define an external relation type in the relation service.

Users can 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 can determine the `RoleInfo` objects to be passed to the superclass constructor. This effectively encapsulates a relation type definition in a class that can 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 through 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 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 that 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 178). Individual roles that cannot be accessed cause an exception whose class indicates the nature of the error (see the exception classes in FIGURE 11-2).

The methods for bulk role access follow a “best effort” policy: access to all indicated roles is attempted and roles that cannot be accessed do not block the operation. Those that 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 181).

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 in which the MBean is a member.

Maintaining Consistency

The relation service delegates the responsibility of maintaining role consistency to the relation object. In this 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 relation type has the corresponding role information for the named role.
- 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 with respect to cardinality and MBean class.

An implementation of the `Relation` interface can rely on the `checkRoleReading` and `checkRoleWriting` methods of the relation service MBean, provided to simplify the above verifications.

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 to maintain consistency.

The relation service must be informed of all new role values so that it can listen for a unregistration notification concerning any of the member MBeans. When a member MBean of an external relation is unregistered from the MBean server, the relation service checks the new cardinality of the role it satisfied.

If the cardinality is no longer valid and if the purge flag is `true`, the relation service removes this relation instance (see “RelationService Class” on page 171). If the external relation is still valid, the relation service calls its `handleMBeanUnregistration` method.

When called, this method removes the MBean from the role where it was referenced (because 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 unregistrations itself. It also allows the relation implementation to define how the corresponding role will be updated. For example, the unregistration 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 uses 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 satisfied by the given role list.

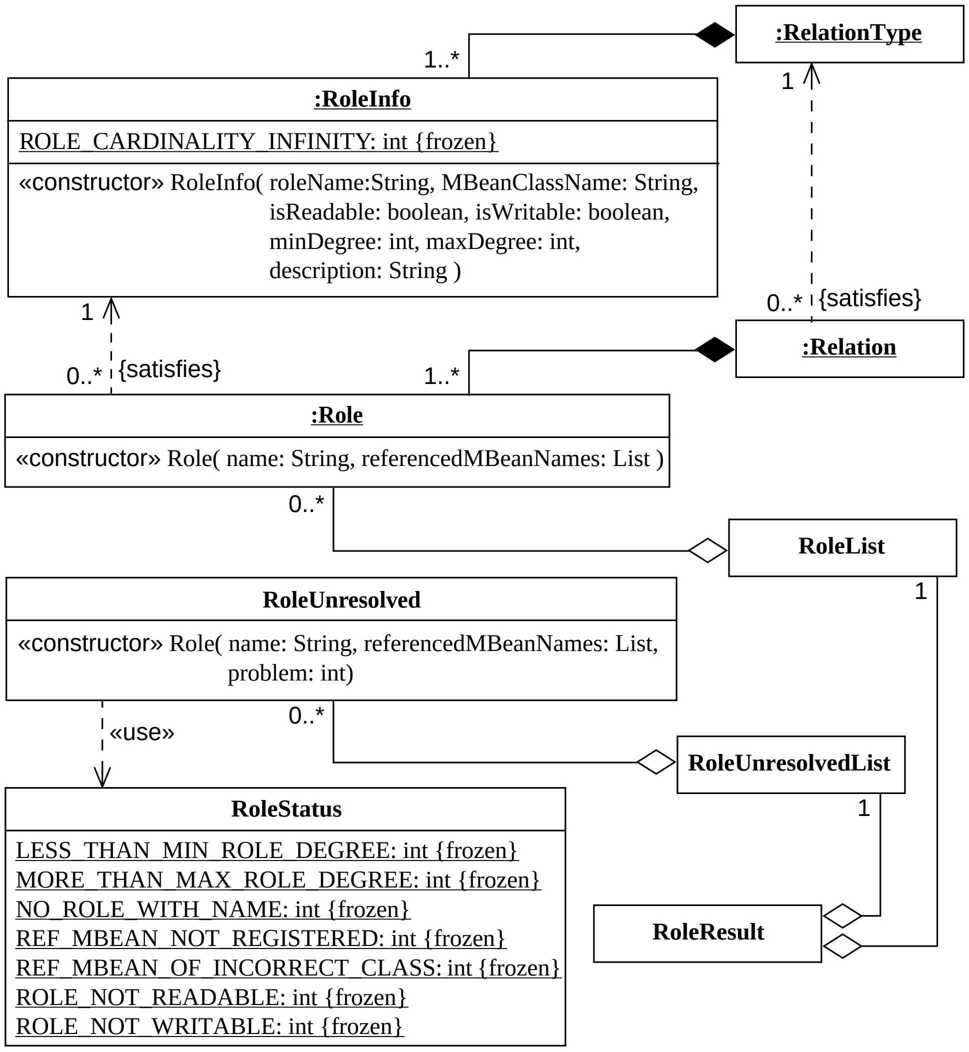
In fact, the `RelationSupport` class implements the `RelationSupportMBean` that extends the `Relation` interface. This implies that it is also a standard MBean whose management interface exposes all the relation access methods. Because an external relation must first be registered in the MBean server, external instances of the relation support class can be managed by remote applications.

Users can also extend the `RelationSupport` class to take advantage of its implementation when developing a customized external relation. Users can also choose to extend its MBean interface 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 usually instantiates 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.



:RelationType
1..*
1
:RoleInfo
ROLE_CARDINALITY_INFINITY: int {frozen}
«constructor» RoleInfo(roleName:String, MBeanClassName: String, isReadable: boolean, isWritable: boolean, minDegree: int, maxDegree: int, description: String)
1
0..* {satisfies}
0..* {satisfies}
:Relation
1..*
:Role
«constructor» Role(name: String, referencedMBeanNames: List)
0..*
RoleList
1
RoleUnresolved
«constructor» Role(name: String, referencedMBeanNames: List, problem: int)
«use»
0..*
RoleUnresolvedList
1
RoleStatus
LESS_THAN_MIN_ROLE_DEGREE: int {frozen}
MORE_THAN_MAX_ROLE_DEGREE: int {frozen}
NO_ROLE_WITH_NAME: int {frozen}
REF_MBEAN_NOT_REGISTERED: int {frozen}
REF_MBEAN_OF_INCORRECT_CLASS: int {frozen}
ROLE_NOT_READABLE: int {frozen}
ROLE_NOT_WRITABLE: int {frozen}
RoleResult

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 `RoleInfo` constructor verifies that this is a proper, non-empty interval.

- The name of the class or interface of which all members must be instances, as determined by the MBean server's `isInstanceOf` method.
- 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, for each role, 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 in other relation types are identical; this is the user's responsibility.

`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 satisfy the role information of its relation's type. The role name is the key that associates the role value with its defining role information.

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 verifies 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 than or equal to the maximum cardinality.
- Each object name references a registered MBean that is an instance of the expected class or interface.

`RoleList` Class

This class extends `java.util.ArrayList` to represent a set of `Role` objects.

Instances of the `RoleList` class are used to define initial values for a relation. When calling the `createRelation` method of the relation service, roles that admit a cardinality of 0 can 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 used as parameters to the `setRoles` method of both the relation service and the `Relation` interface. These methods only set 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 failed, encoded as an integer value. The constants for decoding the problem are given in the FIGURE 11-5.

`RoleUnresolvedList` Class

This class extends `java.util.ArrayList` to represent a set of `RoleUnresolved` objects. All bulk access methods return 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 that cannot 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 that cannot 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, as given in FIGURE 11-5.

CHAPTER 12

Security

A Java Management extensions (JMX) MBean server might have access to sensitive information and might be able to perform sensitive operations. In such cases, it is desirable to control who can access that information and who can perform those operations. The JMX specification builds on the standard Java security model by defining *permissions* that control access to the MBean server and its operations.

The security checks described in this chapter are only performed when there is a *security manager*. That is, if `System.getSecurityManager()` returns `null`, then no checks are performed.

Permissions

It is assumed that the reader has some familiarity with the Java security model. An excellent reference is *Inside Java™ 2 Platform Security* by Li Gong (Addison Wesley, 1999). Documentation is also available online as part of the Java 2 platform Standard Edition (J2SE) Standard Development Kit (SDK).

Sensitive operations require permissions. Before such an operation is performed, a check is performed to ensure that the caller has the required permission or permissions.

At any given point in the execution of a program, there is a current set of permissions that a thread of execution holds. When such a thread calls a JMX specification operation, we say that these are the *held permissions.*

An operation that performs a security check does so by defining a *needed permission*. The operation is allowed if the held permissions *imply* the needed permission. That is, at least one held permission must imply the needed permission.

A permission is a Java object that is a subclass of `java.security.Permission`. This class defines the following method:

```
public boolean implies(Permission permission);
```

A held permission *held* implies a needed permission *needed* if *held*.implies(*needed)*.

For example, to call MBeanServerFactory.createMBeanServer, a thread needs the permission

```
MBeanServerPermission(“createMBeanServer”)
```

Here are some permissions a thread can hold that imply this needed permission:

- MBeanServerPermission(“createMBeanServer”)
- MBeanServerPermission(“createMBeanServer,findMBeanServer”)
- MBeanServerPermission(“*”)
- java.security.AllPermission()

A thread that does not hold any of these permissions, or any other permission that implies the needed one, will not be able to call createMBeanServer. An attempt to do so will result in a SecurityException.

The JMX 1.2 specification defines three permissions:

- MBeanServerPermission
- MBeanTrustPermission
- MBeanPermission

These permissions are described in the following sections.

MBeanServerPermission

MBeanServerPermission controls access to the static methods of the class javax.management.MBeanServerFactory. (See “MBean Server Factory” on page 121.) An MBeanServerPermission is constructed with a single string argument, and the meaning of the permission object depends on this string, as follows:

- MBeanServerPermission(“createMBeanServer”) controls access to the two overloaded methods MBeanServerFactory.createMBeanServer. Holding this permission allows the creation of an MBeanServer object that is registered in the list accessible through MBeanServerFactory.findMBeanServer.

  This permission implies the newMBeanServer permission, so holding it also allows the creation of an MBeanServer object that is not registered in that list.

- MBeanServerPermission(“newMBeanServer”) controls access to the two overloaded methods MBeanServerFactory.newMBeanServer. Holding this permission allows the creation of an MBeanServer object that is not registered in the list accessible through MBeanServerFactory.findMBeanServer.
- MBeanServerPermission(“releaseMBeanServer”) controls access to the method MBeanServerFactory.releaseMBeanServer. Holding this permission allows the removal of an MBeanServer object from the list accessible through MBeanServerFactory.findMBeanServer.
- MBeanServerPermission(“findMBeanServer”) controls access to the method MBeanServerFactory.findMBeanServer. Holding this permission allows you to find an MBeanServer object in the MBeanServerFactory’s list given its identifier, and to retrieve all MBeanServer objects in the list.

As a convenience in defining permissions, two or more of these strings can be combined in a comma-separated list. The resulting permission implies all the operations in the list. Thus, for example, holding MBeanServerPermission(“newMBeanServer,findMBeanServer”) is equivalent to holding both MBeanServerPermission(“newMBeanServer”) and MBeanServerPermission(“findMBeanServer”).

Holding MBeanServerPermission(“*”) is equivalent to holding all the permissions in the list above.

MBeanPermission

MBeanPermission controls access to the methods of an MBeanServer object returned by MBeanServerFactory.createMBeanServer or MBeanServerFactory.newMBeanServer. (See Chapter 7, "MBean Server", “MBean Server”.)

An MBeanPermission is constructed using two string arguments. The first argument is conventionally called the *name* of the permission, but we refer to it here as the *target*. The second argument is the *actions* of the permission.

MBeanPermission Target

The target of an MBeanPermission groups together three pieces of information, each of which can be omitted:

- The *class name*. For a needed permission, this is the class name of an MBean being accessed. Certain methods do not reference a class name, in which case the class name is null.

For a held permission, this is either empty or a *class name pattern*. A class name pattern can be a literal class name such as `javax.management.MBeanServerDelegate` or a wildcard such as `javax.management.*`. If the class name is empty or is an asterisk (`*`), the permission covers any class name.

- The *member*. For a needed permission, this is the name of the attribute or operation being accessed. For `MBeanServer` methods that do not reference an attribute or operation, the member is null.

  For a held permission, this is either the name of an attribute or operation that can be accessed, or it is empty or an asterisk (`*`), that covers any attribute or operation.

- The *object name*. For a needed permission, this is the `ObjectName` of the MBean being accessed. (See "ObjectName Class" on page 109.) For operations that do not reference a single MBean, it is null.

  For a held permission, this is the `ObjectName` of the MBean or MBeans that can be accessed. It can be an object name pattern, that covers all MBeans with names matching the pattern; see "Pattern Matching" on page 111. It can also be empty, covering all MBeans regardless of their name.

  If the domain part of the `ObjectName` in a held permission is empty, it is not replaced by a default domain, because the permission could potentially apply to several MBean servers with different domains.

A held `MBeanPermission` only implies a needed permission if all three items match.

If a needed permission has a null class name, the class name is not relevant for the action being checked, so a held permission will match regardless of its class name.

If a held permission has an empty class name, this means that the permission covers any class name, so a needed permission will match no matter what its class name is.

The same rules apply to the member and the object name.

The three items in the target are written as a single string using the syntax:

```
className#member[objectName]
```

Any of the three items can be omitted, but at least one must be present.

Any of the three items can be the character "`-`", representing a null item. A null item is not the same as an empty item. An empty class name, for example, in a held permission is the same as "`*`" and implies any class name. A null class name in a needed permission is implied by any class name. A needed permission never has an empty class name, and usually a held permission never has a null class name.

The following are some examples of targets with their meanings:

- `com.example.Resource#Name[com.example.main:type=resource]`

This represents access to the attribute or operation called `Name` of the MBean whose object name is `com.example.main:type=resource` and whose class name is `com.example.Resource`.

- `com.example.Resource[com.example.main:type=resource]`

  `com.example.Resource#*[com.example.main:type=resource]`

  These both mean the same thing, namely access to any attribute or operation of the MBean with this object name and class name.
- `#Name[com.example.main:type=resource]`

  `*#Name[com.example.main:type=resource]`

  These both mean the same thing, namely access to the attribute or operation called `Name` of the MBean with the given object name, regardless of its class name.
- `[com.example.main:type=resource]`

  This represents access to the MBean with the given object name, regardless of its class name, and regardless of what attributes or operations can be referenced.
- `[com.example.main:*]`

  This represents access to any MBean with an object name that has the domain `com.example.main`.
- `com.example.Resource#Name`

  This represents access to the attribute or operation called `Name` in any MBean with a class name of `com.example.Resource`.

`MBeanPermission` Actions

The *actions* string of an `MBeanPermission` represents one or more methods from the `MBeanServer` interface. Not all methods in that interface are possible values for the *actions* string. The complete list is as follows:

- `addNotificationListener`
- `getAttribute`
- `getClassLoader`
- `getClassLoaderFor`
- `getClassLoaderRepository`
- `getMBeanInfo`
- `getObjectInstance`
- `instantiate`
- `invoke`
- `isInstanceOf`
- `queryMBeans`
- `queryNames`
- `registerMBean`
- `removeNotificationListener`

- `setAttribute`
- `unregisterMBean`

For a needed permission, the *actions* string will always contain exactly one of these strings.

For a held permission, it can contain one of these strings, or a comma-separated list of strings. Holding an `MBeanPermission` with *actions* `"invoke,instantiate"` is equivalent to holding two `MBeanPermissions`, one with *actions* `"invoke"` and the other with *actions* `"instantiate"`, each having the same target as the original.

A held permission can also have `"*"` as its *actions* string, which covers all *actions*.

The meanings of the different *actions* are summarized in the table below, as well as the risks associated with granting these permissions to potentially malicious code.

TABLE 12-1 `MBeanPermission` actions

Action	**Meaning when held with *className*, *member*, *objectName***
`addNotificationListener`	Add a notification listener to an MBean with a class name matching *className* and with an object name matching *objectName*. Granting this permission to malicious code could alter the behavior of notification broadcasters (see "NotificationBroadcaster and NotificationEmitter Interfaces" on page 50), by adding listeners that block forever or that throw exceptions.
`getAttribute`	Get the value of an attribute *member* from an MBean with a class name matching *className* and with an object name matching *objectName*.
`getClassLoader`	Get a class loader object with a class name matching *className* and that is registered in the MBean server with an object name that matches *objectName*. Granting this permission to malicious code could alter the behavior of the class loader. For example, if the class loader is an m-let (see "Class Loader Functionality" on page 141), malicious code could add to its list of URLs.
`getClassLoaderFor`	Get the class loader that was used to load an MBean object with a class name matching *className* and with an object name matching *objectName*. Granting this permission to malicious code carries similar risks to those for `getClassLoader`.
`getClassLoaderRepository`	Get a reference to the MBean server's class loader repository (see "The Class Loader Repository" on page 142). Granting this permission to malicious code allows it to load classes through these loaders, including over the network if the class loader repository contains `URLClassLoaders`, or something similar.
`getDomains`	See ObjectName domains in which MBeans are registered if they match *objectName*.

TABLE 12-1 `MBeanPermission` actions

Action	Meaning when held with ***className***, ***member***, ***objectName***
`getMBeanInfo`	Get the MBeanInfo of an MBean with a class name matching *className* and with an object name matching *objectName*. (See "MBean Metadata Classes" on page 53.)
`getObjectInstance`	Get the `ObjectName` and class name of an MBean with a class name matching *className* and with an object name matching *objectName*. Holding this permission allows the discovery of the class name of any MBean, provided that its `ObjectName` and class name match *objectName* and *className*.
`instantiate`	Instantiate a Java class whose name matches *className*, using one of the class loader possibilities offered by the overloaded `instantiate` methods of the `MBeanServer` interface. This permission is also needed by the `createMBean` methods. Granting this permission to malicious code allows it to load classes through those loaders, including over the network if there is a `URLClassLoader` or something similar.
`invoke`	Invoke the method *member* from an MBean with a class name matching *className* and with an object name matching *objectName*.
`isInstanceOf`	Determine whether an MBean with a class name matching *className* and with an object name matching *objectName* is an instance of any named Java class.
`queryMBeans`	Discover the names and classes of MBeans whose class names match *className* and whose object names match *objectName*, and apply queries to them. (See "Queries" on page 129.) Holding this permission implies the corresponding queryNames permission.
`queryNames`	Discover the names of MBeans whose class names match *className* and whose object names match *objectName*.
`registerMBean`	Register MBeans whose class names match *className* under object names that match *objectName*. This permission is needed for the `registerMBean` and `createMBean` methods of the MBean server. Granting this permission to malicious code could allow it to register a rogue MBean in the place of a legitimate one, especially if it also has the `unregisterMBean` permission.

TABLE 12-1 `MBeanPermission` actions

Action	Meaning when held with ***className***, ***member***, ***objectName***
`removeNotificationListener`	Remove a notification listener from an MBean with a class name matching *className* and with an object name matching *objectName*. The impact of granting this permission is limited, because malicious code can only remove listeners it references. But the forms of `removeNotificationListener` that identify the listener with an `ObjectName` would be vulnerable to malicious code with this permission.
`setAttribute`	Set the value of an attribute *member* in an MBean with a class name matching *className* and with an object name matching *objectName*.
`unregisterMBean`	Unregister an MBean with a class name matching *className* and with an object name matching *objectName*.

Unchecked MBean Server Methods

The following MBean server methods are not subject to permission checks:

- ***`isRegistered`*** - Code can always discover whether an MBean with a given name exists. Without the `queryMBeans` or `queryNames` permission, however, it cannot find out the names of MBeans it does not already know about.

  The reason that this operation is not subject to a permission check is that it would generate a `SecurityException` if the MBean existed but its class name was not covered by the user's permissions, while it would generate an `InstanceNotFoundException` if the MBean did not exist, regardless of what class names are covered by the user's permissions. Generating a `SecurityException` would therefore be equivalent to admitting that the MBean does indeed exist, so the permission checking would serve no purpose.

  Though it is possible to refine the permission checking semantics for this case, the `isRegistered` method is not considered sufficiently sensitive to justify it.
- ***`getMBeanCount`*** - Code can always discover how many MBeans there are. Again, this is not considered sufficiently sensitive to justify defining a permission check for it.
- ***`getDefaultDomain`*** - Code can always discover the MBean server's default domain.

Permission Checking for Queries

The `queryMBeans` and `queryNames` *actions* for `MBeanPermission` allow control of which MBeans are visible to queries. A `queryMBeans` operation proceeds as follows:

1. The held permissions are checked to see if they imply `MBeanPermission("-#-[-]", "queryMBeans")` If not, a `SecurityException` is thrown.

   If the held permissions include *any* `queryMBeans` permission, this implies the permission shown here. The exception will only be thrown if there are no `queryMBeans` permissions in the caller's set.

   Without this check, if the policy that grants permissions is accidentally configured without `queryMBeans` permissions, then all queries would return an empty set, with no indication that the reason had anything to do with security. The check described here helps avoid this confusion.

2. The `ObjectName` parameter to `queryMBeans`, that is typically an object name pattern, is used to select a set of MBeans to which the query applies. If the parameter is `null`, all MBeans are selected.

3. For each MBean in the set, an `MBeanPermission` is constructed where the *actions* parameter is `"queryMBeans"` and the target parameter contains the class name and object name of the MBean. If the held permissions do not imply this permission, the MBean is eliminated from the set.

4. For each MBean in the remaining set, the `QueryExp` parameter to `queryMBeans` is used to decide whether it is included in the final set.

The rules for the `queryNames` operation are exactly the same as just stated, but with `queryMBeans` replaced by `queryNames`.

With these rules in place, the permissions held by a thread completely govern the set of MBeans that thread can see through queries.

It is important that MBeans that are not covered by the held permissions be eliminated from the set *before* the query is executed. In other words, step 3 must happen before step 4. Otherwise, malicious code could implement the `QueryExp` interface to save each MBean in the selected set somewhere.

Permission Checking for `getDomains`

The `getDomains` permission filters the information about MBean names that is visible to a thread, in a similar fashion to `queryMBeans` and `queryNames`. The thread's held permissions should imply :

```
MBeanPermission("-#-[-]", "getDomains")
```

Otherwise, the `MBeanServer.getDomains()` method throws a `SecurityException`. Otherwise, the MBean server first gets the list of domains that would be returned if there were no security checks, and for each domain *d* in the list, it checks that the held permissions imply:

```
MBeanPermission("-#-[d:x=x]", "getDomains")
```

Otherwise, the domain is eliminated from the list.

The `x=x` is an artifact of the way `ObjectName` works. An implementation can use any other *key=value* possibility instead, but there must be one.

When defining held permissions, for instance in a security policy file, the `getDomains` permission should always either omit the `ObjectName` or supply an `ObjectName` pattern with just a `*` in the key properties, such as `"*:*"`, `"com.example.*:*"`, or `"com.example.visible:*"`.

Permission Checking for `getAttributes` and `setAttributes`

A similar scheme to the one for queries is used for the `getAttributes` and `setAttributes` operations. A `getAttributes` operation proceeds as follows:

1. The held permissions are checked to see if they imply `MBeanPermission("`*className*`#-[`*objectName*`]", "getAttribute")` where *className* and *objectName* are the class name and object name of the MBean being accessed. If not, a `SecurityException` is thrown.

   If the held permissions allow `getAttribute` on *any* attribute of the MBean, they will imply the permission shown here. Only if `getAttribute` is not allowed for any attributes in the MBean will the exception be thrown.

   Without this check, if no attributes are accessible to the caller, an empty `AttributeList` would be returned, with no indication that the reason had anything to do with security.

2. For each attribute *attr* in the list given to the `getAttributes` operation, the permission `MBeanPermission("`*className#attr*`[`*objectName*`]", "getAttribute")` is constructed. If the held permissions do not imply this permission, the attribute is eliminated from the list.

3. The `getAttributes` operation then works on the attributes remaining in the list.

The rules for the `setAttributes` operation are exactly the same as just stated, but with `getAttribute(s)` replaced by `setAttribute(s)`.

`MBeanTrustPermission`

This permission represents "trust" in a signer or codesource. If a signer or codesource is granted this permission, it is considered a trusted source for MBeans. Only MBeans from trusted sources can be registered in the MBean server.

In conjunction with MBeanPermission, MBeanTrustPermission enables fine-grained control of which MBeans are registered in the MBean server. The registerMBean action of MBeanPermission controls what entities can register MBeans. MBeanTrustPermission controls what MBeans they can register.

An MBeanTrustPermission is constructed with a single string argument. In this version of the JMX specification, the argument can have two possible values:

- "register"
- "*"

Both values have the same meaning, but if in a future version of the JMX specification other possibilities are added, "*" will cover all of them.

The details of the MBeanTrustPermission check are illustrated in the following example.

Let c be the Java class of an MBean to be registered in the MBean server via its createMBean or registerMBean methods, and let p be a permission constructed as follows:

```
p = new MBeanTrustPermission("register");
```

Then, for the createMBean or registerMBean to succeed, the following expression must be true:

```
c.getProtectionDomain().implies(p)
```

Otherwise, a SecurityException is thrown.

Policy File Examples

The following are some examples of how the permissions described in the previous sections can be granted using the standard Java policy file syntax.

The simplest MBean access policy is to grant all signers and codebases access to all MBeans:

```
grant {
    permission javax.management.MBeanPermission "*", "*";
};
```

Here is a more restrictive policy that grants the code in `appl1.jar` the permission to get the MBean server's class loader repository:

```
grant codeBase “file:${user.dir}${/}appl1.jar” {
    permission javax.management.MBeanPermission “”,
        “getClassLoaderRepository”;
};
```

Here is a policy that grants the code in `appl2.jar` the permission to call the `isInstanceOf` and `getObjectInstance` operations for MBeans from any class, provided they are registered in the domain “`d1`”:

```
grant codeBase “file:${user.dir}${/}appl2.jar” {
    permission javax.management.MBeanPermission “[d1:*]”,
        “isInstanceOf, getObjectInstance”;
};
```

Here is a policy that grants the code in `appl3.jar` the permission to find MBean servers, and to call the `queryNames` operation but restricting the returned set to MBeans in the domain “`JMImplementation`”:

```
grant codeBase “file:${user.dir}${/}appl3.jar” {
    permission javax.management.MBeanServerPermission
        “findMBeanServer”;
    permission javax.management.MBeanPermission
        “JMImplementation:*”, “queryNames”;
};
```

If the MBean server has MBeans with names in other domains, for example an MBean registered as “`com.example.main:type=user,name=gorey`”, they will never appear in the result of a `queryNames` executed by code in `appl3.jar`.

Here is a policy that grants the code in `appl4.jar` the permission to create and manipulate MBeans of class “`com.example.Foo`” under any object name:

```
grant codeBase “file:${user.dir}${/}appl4.jar” {
    permission javax.management.MBeanPermission
        “com.example.Foo”, “instantiate, registerMBean”;
    permission javax.management.MBeanPermission
        “com.example.Foo#doIt”,
        “invoke,addNotificationListener,removeNotificationListener”;
};
```

The first permission ignores the object name. The operation or attribute name is not required by these two actions.

The second permission, however, uses the member part for the "invoke" action and ignores it for the "add/removeNotificationListener" actions.

Here is a policy that allows MBeans to be registered no matter where they come from. (A thread that registers MBeans must also have the appropriate MBeanPermissions.)

```
grant {
    permission javax.management.MBeanTrustPermission “register”;
};
```

Here is a policy that only trusts MBeans signed by "Gorey":

```
grant signedBy “Gorey” {
    permission javax.management.MBeanTrustPermission “register”;
};
```