Pro JMX: Java Management Extensions

J. JEFFREY HANSON

Apress™

ISBN (pbk): 1-59059-101-1

Printed and bound in the United States of America 12345678910

Technical Reviewers: Robert Castaneda and Nathan Lee

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steven Rycroft, Dominic Shakeshaft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Beth Christmas

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Production Editor: Lori Bring

Proofreader: Thistle Hill Publishing Services, LLC

Compositor: Kinetic Publishing Services, LLC

Indexer: Nancy Guenther

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 3

The Three-Level Model: Agents

Whenever the term *agent* is brought up in software development circles, we tend to think of independent, nomadic components that roam about the Web searching, gathering, prying, and engaging in other potentially scary activities. However, in nonspecific terms, a generic software agent can be thought of as an autonomous component that communicates and interacts with other entities on behalf of one or more controlling entities. Extensions of generic agents include intelligent agents, proxy agents, master agents, subagents, and others.

A software agent has historically been thought of as including such properties as autonomy, reactivity, social ability, collaborative behavior, adaptability, and mobility. Although JMX agents are certainly not precluded from possessing these properties, they are generally defined in far simpler terms. JMX agents are software modules that link management applications with managed resources.

A software management agent can be generically defined as a software module representing a managed entity that stores and delivers management information to other interested devices or software applications over a given protocol. For example, within the realm of the Simple Network Management Protocol (SNMP), agents are loosely defined as network devices, such as hosts, bridges, hubs, routers, etc., that can be communicated with and monitored by a management application. Typically, a management agent responds to a management application's requests, performs actions requested by the management application, and may send notification messages to the application.

In this chapter, we will look at the manner in which JMX defines management agents and how JMX agents interact with MBeans and management applications. We will also investigate how JMX defines services in respect to agents.

The Agent Level of JMX

The JMX agent level defines management agents, which contain and expose an MBean server, agent services, and MBeans. The agent level of the JMX specification is aimed at the management solutions development community. The specifications for JMX agent design and implementation allow developers to build applications that manage resources without prior knowledge of the attri butes or operations of the resources.

An agent facilitates access to managed resources by acting as the connecting link between management applications and registered MBeans. An agent is responsible for dispatching requests to the MBeans that it contains as well as broadcasting notifications to interested notification receivers that have previously registered with the agent. A JMX agent consists of several mandatory services, known as *agent services,* an MBean server, and at least one protocol adaptor or connector.

Services that a JMX agent must provide include a monitoring service, a timer service, a relation service, and a dynamic class-loading service. Additional services can be provided as well. These services are also registered as MBeans.

An agent can reside in a local JVM or in a JVM that is remote to interested management applications or remote to MBeans that the agent contains. Later, I will discuss how an agent can be constructed to enable access from applications residing in remote JVMs.

MBean Servers

As discussed in Chapter 1, an *MBean server* is an object defined by the `javax.management.MBeanServer` interface and/or the `javax.management.MBeanServerConnection` interface and acts as a registry and repository for MBeans contained by an agent. An MBean server performs all life cycle operations on an MBean such as creation, registration, and deletion. An MBean server also performs query operations that find and retrieve registered MBeans.

The `MBeanServer` interface extends the `javax.management.MBeanServerConnection` interface. The `MBeanServerConnection` interface defines methods that can be called on an MBean server regardless of whether the MBean server is remote or local. The `MBeanServer` interface extends the `MBeanServerConnection` interface to define methods that can only be called on a local MBean server.

All interaction with MBeans must be directed through an agent, and thus the agent acts as a proxy for the MBean server that it contains.

Figure 3-1 illustrates the relationships between an agent, its agent services, and an MBean server.

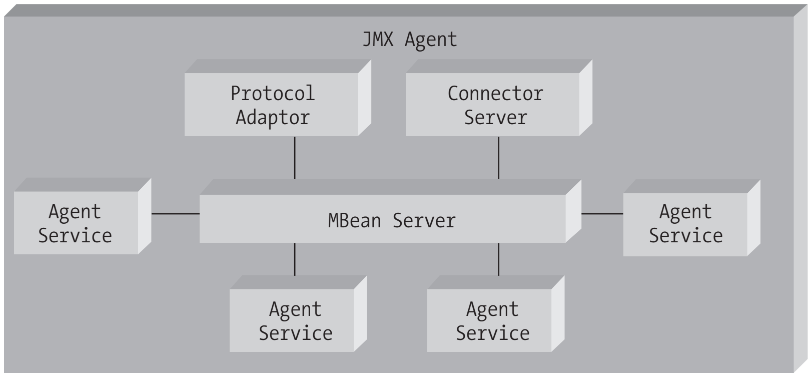

Figure 3-1. A JMX agent with its agent services and MBean server

An agent typically does not have prior knowledge of the details of the resources that it contains or the client applications that will use the resources. It simply relies on the standard generic instrumentation functionality of the JMX framework in order to discover the resources' capabilities and to act on them accordingly.

Recall how earlier I demonstrated a service agent that registered and exposed one MBean, the `DateTimeService` MBean:

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

ObjectName oName =
   new ObjectName("services:name=DateTime,type=information");

mbServer.registerMBean(new DateTimeService(), oName);
```

This type of agent implies that you know beforehand the exact type and number of MBeans that your agent will contain. Most situations do not afford an agent the luxury of knowing beforehand the number and types of MBeans that it will contain. This makes it very important that agents have a mechanism for dynamically adding, removing, and maintaining their list of registered MBeans.

Let's take a look at how you might refactor your service agent to handle adding MBeans dynamically.

Opening Up the ServiceAgent

The first thing you need to do is move the declaration for your MBean server instance out of the `main` method and declare it as a global, private field of your `ServiceAgent` class.

```
private MBeanServer mbServer = null;

public static void main(String[] args)
{
   ServiceAgent agent = new ServiceAgent();
}
```

Declaring an Effective Constructor

Next, explicitly declare a no-arg constructor that will serve to create your MBean server instance and to create and register the `HtmlAdaptorServer` instance.

```
public ServiceAgent()
{
   try
   {
      mbServer = MBeanServerFactory.createMBeanServer();

      ObjectName adaptorOName =
         new ObjectName("adaptors:protocol=HTTP");

      HtmlAdaptorServer htmlAdaptor = new HtmlAdaptorServer();
      mbServer.registerMBean(htmlAdaptor, adaptorOName);
      htmlAdaptor.start();
   }
   catch(MBeanRegistrationException e) {}
   catch(NotCompliantMBeanException e) { }
   catch(MalformedObjectNameException e) {}
   catch(InstanceAlreadyExistsException e) {}
}
```

Adding the addResource Method

Notice that the lines of code that instantiate and register your `DateTimeService` MBean have been removed. In their place you include a method named `addResource`, which allows MBeans to be added dynamically. The `addResource` method takes as parameters the desired name for the MBean, the property list for the MBean, and the class name of the MBean. The property list is passed in as a `Hashtable`. The `addResource` method throws a generic `ServiceException` that you define in order to encapsulate any exception that is caught within the method.

The first thing you need to do is add an import for the `Hashtable` class, as follows:

```
import java.util.Hashtable;
```

Now add your implementation for the `addResource` method as shown here:

```
public void addResource(String name, Hashtable properties, String className)
   throws ServiceAgentException
{
   try
   {
      Class cls = Class.forName(className);
      Object obj = cls.newInstance();
      Hashtable allProps = new Hashtable();
      allProps.put("name", name);
      properties.putAll(allProps);
      ObjectName oName = new ObjectName("services", allProps);
      mbServer.registerMBean(obj, oName);
   }
   catch (IllegalAccessException e)
   {
      throw new ServiceAgentException(e.getMessage());
   }
   catch (InstantiationException e)
   {
      throw new ServiceAgentException("Unable to create instance of MBean: "
                                      + e.getMessage());
   }
   catch (ClassNotFoundException e)
   {
      throw new ServiceAgentException("Unable to find class for MBean: "
                                      + e.getMessage());
   }
   catch (MalformedObjectNameException e)
   {
      throw new ServiceAgentException("Invalid object name: "
                                      + e.getMessage());
   }
   catch (InstanceAlreadyExistsException e)
   {
      throw new ServiceAgentException("The MBean already exists: "
                                      + e.getMessage());
   }
   catch (MBeanRegistrationException e)
   {
      throw new ServiceAgentException("General registration exception: "
                                      + e.getMessage());
      }
      catch (NotCompliantMBeanException e)
      {
         throw new ServiceAgentException("The class is not MBean compliant: "
                                         + e.getMessage());
      }
   }
```

Now, you add your `ServiceAgentException` class. For the time being, keep it very simple, as follows:

```
public class ServiceAgentException extends Exception
{
   public ServiceAgentException()
   {
      super();
   }

   public ServiceAgentException(String message)
   {
      super(message);
   }
}
```

Adding the getResources Method

Next, you add a method, `getResources`, to facilitate the retrieval of previously registered MBeans from the MBean server within the agent. For now, you will only retrieve MBeans by name. Later, you will add the ability to query for MBeans based on parts of a name, property list fragments, and wildcards.

The first thing you need to do is add an `import` statement for the `Set` class, as follows:

```
import java.util.Set;
```

Now you add your implementation for the `getResources` method:

```
   public Set getResources(String name)
      throws ServiceAgentException
   {
      try
      {
         Hashtable allProps = new Hashtable();
         allProps.put("name", name);
         ObjectName oName = new ObjectName("services", allProps);
         Set resultSet = mbServer.queryMBeans(oName, null);
         return resultSet;
      }
      catch (MalformedObjectNameException e)
      {
         throw new ServiceAgentException("Invalid object name: "
                                         + e.getMessage());
      }
   }
```

Categorizing Resource Groups Using Domains

JMX categorizes resource groups within a *domain.* A domain is a namespace for a specific set of information and resources within an agent, management system, or application.

Let's look at some of the details and uses for domains and the JMX domain naming conventions.

Domain Names

A JMX domain name is an application-independent, case-sensitive string that labels the namespace defined by an agent, management system, or application. A JMX domain name may be constructed of any string of characters excluding those characters that are used as separators or wildcards for the `ObjectName` class. The excluded characters include the colon (:), comma (,), equal sign (=), asterisk (*), and question mark (?).

Uniquely Naming MBeans Using Key Properties

The `ObjectName` class comprises two elements: the domain name and a key property list. The key property list allows MBeans to be given a unique name within the realm of a domain. Key properties are arbitrary property-value pairs, for example, `type=service`.

The key property list of an `ObjectName` can contain any number of key properties, but must contain at least one key property. The following examples define valid `ObjectName` instances for a domain named `MyDomain`:

```
ObjectName oName = new ObjectName("MyDomain:type=service");
ObjectName oName = new ObjectName("MyDomain:type=service,persistence=transient");
```

Extending an Agent with Agent Services

JMX agents must supply a set of predefined services for operating on MBeans that are registered with the agent's MBean server. These mandatory services are referred to as *agent services* and are usually implemented as MBeans themselves.

Implementing the agent services as MBeans enables an agent to operate on the services in the same manner that it would for an ordinary managed resource. The set of mandatory agent services include the following:

- *Dynamic class loading:* This service is exposed using a technology referred to as the management applet (M-Let). The M-Let technology allows an agent to load classes from arbitrary remote hosts defined by URLs. The classes can then be created and registered as MBeans in the same manner as local MBeans.

- *Monitor:* Monitors observe the value for a given MBean's attribute or attributes and notify listening objects of changes that occur on the value.

- *Timer:* Timers allow notifications to be scheduled as a one-time-only event or as a repeated, periodic event.

- *Relation:* This service defines predefined relation types for associations between MBeans.

Let's look at each of these agent services in more detail.

Dynamic MBean Loading

The dynamic MBean loading service is defined by classes, interfaces, and MBeans within the `javax.management.loading` package. The dynamic MBean loading service enables an agent to retrieve, create, and register MBeans from a remote host.

Dynamic MBean loading is enabled using a compact framework known as a management applet or M-Let. The M-Let framework facilitates loading MBeans specified by an arbitrary URL from a remote host.

The management applet framework is a compact framework that defines the following:

- An XML-like tag, called `MLET`, which describes the information for each MBean, as the following example illustrates:

```
<MLET
   CODE = com.jeffhanson.mlets.MyMLET
   CODEBASE = http://myhost/mlets
   ARCHIVE = "MyMLET.jar"
   NAME = MyMLET>
</MLET>
```

- A URL-locatable text file that contains `MLET`-tagged elements defining MBeans. When an M-Let text file is located, all of the classes specified by its `MLET` tags are downloaded. The classes are then instantiated as MBeans and registered with the MBean server.

- A class loader for retrieving and loading MBean classes from a remote host.

- An MBean that encapsulates the M-Let service itself. The M-Let MBean is registered in the MBean server.

The relationships between components defined by the M-Let framework are illustrated in Figure 3-2.

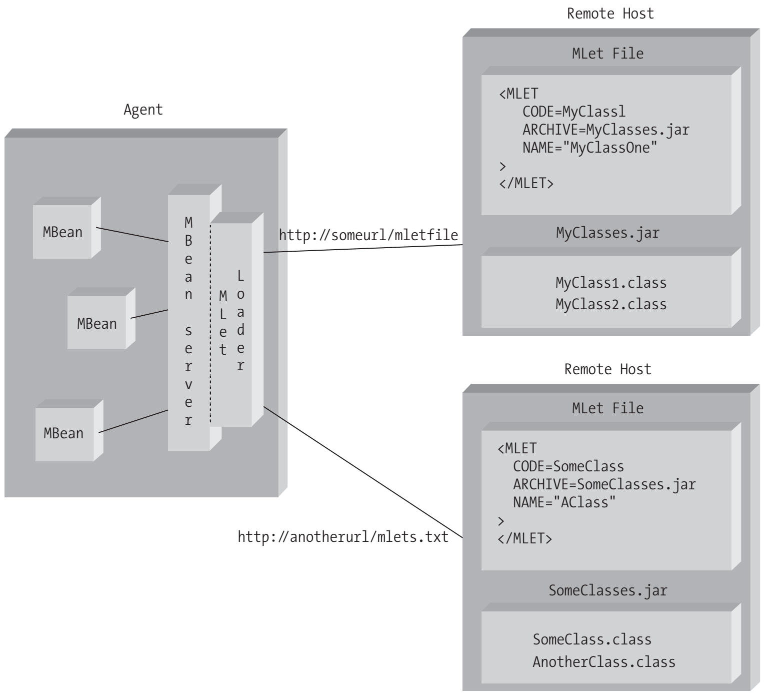

Figure 3-2. A JMX agent, an MLet class loader, and remote hosts

Now, let's look at how the JMX monitor service allows you to observe over time the variations of attribute values in MBeans and to emit notification events based on configurable thresholds.

Using Monitors to Track Changes

Monitors are a set of MBeans defined by classes and interfaces within the `javax.management.monitor` package that define components which allow interested listeners to observe, over a period of time, the value of a given attribute or the changes that occur in the values of a given attribute for a particular MBean. A monitor can transmit notification events for changes that occur within the observed attribute's values at given thresholds.

All monitors implement an MBean interface that extends a common interface, `MonitorMBean`. The `MonitorMBean` interface defines the following attributes and operations:

- `observedAttribute`: This flags the attribute as able to be observed.
- `granularityPeriod`: This attribute defines the granularity period in milliseconds over which to observe.
- `active`: An attribute that tests whether or not a monitor is active. This attribute is set to `true` when the `start` operation is invoked. It is set to `false` when the `stop` operation is called.
- `start`: This operation starts the monitor.
- `stop`: This operation stops the monitor.

Predefined Monitor Types

Three predefined monitors, implemented as MBeans, are provided by a JMX implementation: counter, gauge, and string. The class for each of these extends an abstract class, `Monitor`. The three predefined monitor types are detailed as follows:

- `CounterMonitor`: Monitors attributes that have the characteristics of a counter. These characteristics include the following: The value is always greater than or equal to zero, the value can only be incremented, and the value may roll over.
- `GaugeMonitor`: Monitors attributes that have characteristics of a gauge. This means that the attribute's value may randomly increase or decrease.
- `StringMonitor`: Monitors an attribute that returns a value of type `String`.

Because monitors are interested in the value of an attribute and not the attribute itself, they can monitor any attribute that returns a value with the data type that the monitor recognizes. This allows the `StringMonitor`, for example, to monitor a given attribute so long as the attribute returns a `String` as its value.

Types of Monitor Notifications

A monitor allows the value of an attribute for a particular MBean to be observed for certain thresholds. The value is monitored at intervals specified by a granularity period. Thresholds can be either an exact value or the difference derived from two given observation points. The threshold is known as a *derived gauge.*

When a condition for a derived gauge has been satisfied or when an error condition is encountered, a specific notification type is sent by the monitor. The specific conditions are defined by the monitor's management interface. The predefined notification types are as follows:

- `jmx.monitor.error.mbean`: This notification type is fired by all monitor types and indicates that the observed MBean is not registered with the MBean server.

- `jmx.monitor.error.attribute`: This notification type is fired by all monitor types and indicates that the observed attribute is not found.

- `jmx.monitor.error.type`: This notification type is fired by all monitor types and indicates that the observed attribute type is incorrect.

- `jmx.monitor.error.threshold`: This notification type is only fired by counter and gauge monitors and indicates that a characteristic of the threshold is incorrect.

- `jmx.monitor.error.runtime`: This notification type is fired by all monitor types and indicates that an unknown error has occurred when accessing the value of the observed attribute.

- `jmx.monitor.counter.threshold`: This notification type is only fired by counter monitors and indicates that the threshold has been reached for the observed attribute's value.

- `jmx.monitor.gauge.high`: This notification type is only fired by gauge monitors and indicates that the threshold's upper limit has been reached or exceeded for the observed attribute's value.

- `jmx.monitor.gauge.low`: This notification type is only fired by gauge monitors and indicates that the threshold's lower limit has been reached or exceeded for the observed attribute's value.

- `jmx.monitor.string.matches`: This notification type is only fired by string monitors and indicates that the monitor's comparison string has been matched.

- `jmx.monitor.string.differs`: This notification type is only fired by string monitors and indicates that the monitor's comparison string has been compared against an unequal value.

Supporting Monitor Listeners

Let's see how you would modify your service agent to expose a method that will accept a notification listener and set the listener as the listener to a simple `CounterMonitor`:

```
 import javax.management.monitor.*;
public class ServiceAgent
{
   static String counterMonitorClass =
      "javax.management.monitor.CounterMonitor";

   private MBeanServer mbServer = null;
   private ObjectName counterMonitorName = null;
   private CounterMonitor counterMonitor = null;

   public ServiceAgent()
   {
         mbServer = MBeanServerFactory.createMBeanServer();

         initializeCounterMonitor();

         // Existing code omitted here for sake of brevity.
   }
```

Adding a CounterMonitor

Now, create and register the `CounterMonitor` in your service agent. You register it under the MBean server's default domain.

```
protected void initializeCounterMonitor()
{
   ObjectName counterMonitorName = null;
   counterMonitor = new CounterMonitor();

   // Get the domain name from the MBeanServer.
   //
   String domain = mbServer.getDefaultDomain();

   // Create a new CounterMonitor MBean and add it to the MBeanServer.
   //
   try
   {
      counterMonitorName = new ObjectName(domain + ":name="
                                              + counterMonitorClass);
   }
   catch (MalformedObjectNameException e)
   {
      e.printStackTrace();
      return;
   }

   try
   {
      mbServer.registerMBean(counterMonitor, counterMonitorName);
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }
}
```

Registering a CounterMonitor Listener

Now, add a method to set an object as a listener to the CounterMonitor. The threshold value, the offset value, and the granularity period can all be modified for your particular needs.

```
public void setCounterMonitorListener(ObjectName observedObjName,
                                      NotificationListener listener,
                                      String attrName)
{
   // Register a notification listener
   // with the CounterMonitor MBean, enabling the listener to receive
```

```
      // notifications transmitted by the CounterMonitor.
      //
      try
      {
         Integer threshold = new Integer(1);
         Integer offset  = new Integer(1);
         counterMonitor.setObservedObject(observedObjName);
         counterMonitor.setObservedAttribute(attrName);
         counterMonitor.setNotify(true);
         counterMonitor.setThreshold(threshold);
         counterMonitor.setOffset(offset);
         counterMonitor.setGranularityPeriod(1000);

         NotificationFilter filter = null;
         Object handback = null;
         counterMonitor.addNotificationListener(listener, filter, handback);
         if (counterMonitor.isActive() == false)
            counterMonitor.start();
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
```

Testing the CounterMonitor

You can now test your `CounterMonitor` and listener with the following code. First, set the listener as the listener to the `CounterMonitor`.

```
   ObjectInstance objInst = serviceAgent.addResource("DateTime",
                                                     properties,
                                                     mbeanClassName);
   serviceAgent.setCounterMonitorListener(objInst.getObjectName(),
                                                     this,
                                                     "Second");
```

Handling CounterMonitor Notification Events

Any object that is used as a `CounterMonitor` listener must implement the `NotificationListener` interface and the `handleNotification` method defined in the `NotificationListener` interface to handle notifications sent from the `CounterMonitor`.

```
public void handleNotification(Notification notification, Object handback)
   {
      if (notification instanceof MonitorNotification)
      {
         MonitorNotification notif = (MonitorNotification) notification;

         // Get monitor responsible for the notification.
         //
         Monitor monitor = (Monitor) notif.getSource();

         // Test the notification types transmitted by the monitor.
         String t = notif.getType();
         Object observedObj = notif.getObservedObject();
         String observedAttr = notif.getObservedAttribute();

         try
         {
            if (t.equals(MonitorNotification.OBSERVED_OBJECT_ERROR))
            {
               System.out.println(observedObj.getClass().getName()
                                   + " is not registered in the server");
            }
            else if (t.equals(MonitorNotification.OBSERVED_ATTRIBUTE_ERROR))
            {
               System.out.println(observedAttr + " is not contained in " +
                                   observedObj.getClass().getName());
            }
            else if (t.equals(MonitorNotification.OBSERVED_ATTRIBUTE_TYPE_ERROR))
            {
               System.out.println(observedAttr + " type is not correct");
            }
            else if (t.equals(MonitorNotification.THRESHOLD_ERROR))
            {
               System.out.println("Threshold type is incorrect");
            }
            else if (t.equals(MonitorNotification.RUNTIME_ERROR))
            {
               System.out.println("Unknown runtime error");
            }
            else if (t.equals(MonitorNotification.THRESHOLD_VALUE_EXCEEDED))
            {
               System.out.println("observedAttr"
                                   + " has reached the threshold\n");
            }
            else
            {
               System.out.println("Unknown event type");
            }
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }
   }
```

Scheduling Notifications Using Timers

The timer service is a set of classes, interfaces, and MBeans located in the `javax.management.timer` package, which transmits notifications according to scheduled times. The timer service may repeat the notifications at equal intervals called *periods,* and the notifications can be repeated for a number of given *occurrences.* As with all notifications, the timer notifications will be transmitted to all objects registered with the timer. It is then up to the listener to filter the notifications.

When an object adds a notification to a timer, the caller supplies the notification type and the date the notification is to be sent, or, if the notification is to be sent more than once, the period and the number of occurrences.

Figure 3-3 shows the relationships and concepts of periods and occurrences for timer notifications. The timeline demonstrates three occurrences for illustration purposes only.

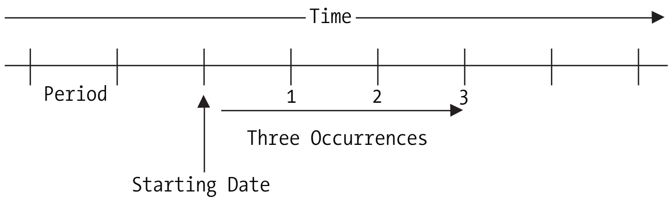

Figure 3-3. Timeline with periods and occurrences

Adding Notifications to a Timer

Notifications are added to a timer using one of the overloaded `addNotification` methods. The timer adds the notification to an internal list and transmits the notification depending on the parameters that are passed to the `addNotification` method. The parameters for all `addNotification` methods are defined as follows:

- `type`: The dot-delimited notification type.
- `message`: The notification message.
- `userData`: Optional user data.
- `date`: The date when the notification will occur. If the `period` and `occurrences` are greater than zero, the date will serve as the starting date for a number of notification events. If the date is prior to the current date, an attempt is made to create the notification by adding the `period` to the date until it is greater than the current date. If the `period` is not specified, or if `date + (period * occurrences)` is earlier than the current date, an exception is thrown.
- `period`: The interval in milliseconds between notification occurrences. Repeating notifications are not enabled if this parameter is zero or null.
- `occurrences`: 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.

Figure 3-4 illustrates a notification with three occurrences that is added with a date prior to the current date. Notice that the end result is a notification with two occurrences.

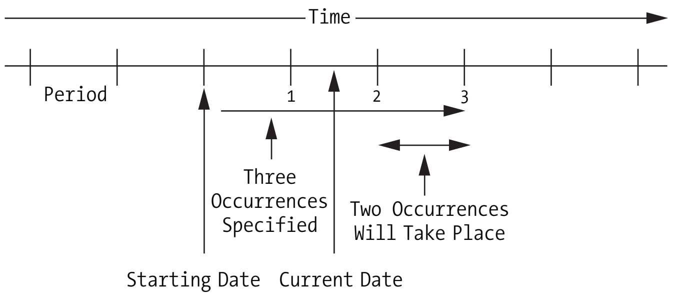

Figure 3-4. Notification added with a date prior to the current date

The `addNotification` method returns an `Integer` identifying the notification. This identifier is subsequently passed to the timer to retrieve information about the notification or to remove the notification from the timer.

Receiving Timer Events

The timer MBean extends the `javax.management.NotificationBroadcasterSupport` class and is thus a valid notification broadcaster. This implies, as with all notifications, that a listener to a timer will receive all notifications that the timer transmits. It is up to the listener to pass in an appropriate filter and handback object, when it registers, in order to filter the notification events as needed.

Each time a timer transmits a notification event and the number of occurrences for the notification is greater than zero, the number of occurrences is decremented by one. Thus, when the `getNbOccurrences(Integer notificationID)` method is called, the number returned will reflect the decremented amount.

When a timer has transmitted a notification the total number of times specified, the notification is removed from the timer. At this point, any subsequent attempts to retrieve information from the timer about the notification will result in an error or a null return value.

Starting and Stopping a Timer

The `javax.management.timer.TimerMBean` interface specifies a `start` operation and a `stop` operation for starting/activating and stopping/deactivating a timer. Once a timer is started, it is considered to be active. When a timer is stopped, it is considered to be inactive. The `isActive` operation, specified in the `javax.management.timer.TimerMBean` interface, can be called to determine the active state of a timer.

A timer maintains its list of notifications when it is stopped, and therefore must process its list of notifications when it is restarted. How a timer processes its list of notifications depends on the value of its `sendPastNotifications` flag. The `sendPastNotifications` flag is set to false by default. If it is set to true, the timer will attempt to send notifications from its list when it is restarted from a stopped/inactive state. When the timer is restarted, it checks its list of notifications to see if it is empty. If the list is not empty, the timer will process the notifications in the list as follows:

- If the `sendPastNotifications` flag of the timer is set to true, the timer will iterate over the list of notifications, transmit each notification, and increment the date for each notification according to the `period` and `occurrences` properties of each notification. If after incrementing a notification's date it is still less than the current date, the notification will be sent again and the date will be incremented again according to its `period` and `occurrences` properties. This is repeated until the notification's date is greater than or equal to the current date or the number of occurrences has been exhausted. If the number of occurrences is exhausted, the notification is removed from the timer's list.

- If the sendPastNotifications flag of the timer is set to false, the timer will iterate over the list of notifications and increment the date for each notification according to the period and occurrences properties of each notification. This is repeated until the notification's date is greater than or equal to the current date or the number of occurrences has been exhausted. If the number of occurrences is exhausted, the notification is removed from the timer's list.

Let's see how you can include a simple timer in your service agent and methods to add and remove listeners.

Adding a Timer to the ServiceAgent

The first thing you need to do is add an import statement for your timer, as follows:

```
import javax.management.timer.Timer;
```

The next thing you do is add a member field to the service agent class to hold the ObjectName of the timer.

```
   private ObjectName timerOName = null;
```

Now, add methods to start and stop the timer. Notice that for this example you create the timer on the first invocation of the startTimer method.

```
   public void startTimer()
      throws ServiceAgentException
   {
      if (timerOName == null)
      {
         try
         {
            timerOName = new ObjectName("TimerServices:name=SimpleTimer");
            mbServer.registerMBean(new Timer(), timerOName);
            mbServer.invoke(timerOName, "start", null, null);
         }
         catch (Exception e)
         {
            throw new ServiceAgentException("Error starting timer: "
                                             + e.toString());
         }
      }
```

```
        else
        {
           try
           {
              if (((Boolean)mbServer.invoke(timerOName, "isActive",
                                               null, null)).booleanValue() == false)
              {
                 mbServer.invoke(timerOName, "start", null, null);
              }
           }
           catch (Exception e)
           {
              throw new ServiceAgentException("Error starting timer: "
                                                 + e.toString());
           }
        }
     }

     public void stopTimer()
        throws ServiceAgentException
     {
        if (timerOName != null)
        {
           try
           {
              if (((Boolean)mbServer.invoke(timerOName, "isActive",
                                                null, null)).booleanValue() == true)
              {
                 mbServer.invoke(timerOName, "stop", null, null);
              }
           }
           catch (Exception e)
           {
              throw new ServiceAgentException("Error starting timer: "
                                                 + e.toString());
           }
        }
     }
```

The addTimerNotification Method

Now, you add methods that facilitate adding and removing notifications to the timer. For the sake of brevity, funnel all exceptions to a generic `Exception` block and wrap them in a `ServiceAgentException` instance:

```
public Integer addTimerNotification(String type,
                                    String message,
                                    Object userData,
                                    java.util.Date startDate,
                                    long period,
                                    long occurrences)
   throws ServiceAgentException
{
   Object[] param = new Object[]
   {
      type,
      message,
      userData,
      startDate,
      new Long(period),
      new Long(occurrences)
   };

   String[] signature = new String[]
   {
      String.class.getName(),
      String.class.getName(),
      Object.class.getName(),
      Date.class.getName(),
      long.class.getName(),
      long.class.getName()
   };

   try
   {
      Object retVal = mbServer.invoke(timerOName, "addNotification",
                                    param, signature);
      return (Integer)retVal;
   }
   catch (Exception e)
   {
      throw new ServiceAgentException("Error adding notification: "
                                    + e.toString());
   }
}

public void removeTimerNotification(Integer id)
   throws ServiceAgentException
   {
      Object[] param = new Object[]
      {
         id
      };

      String[] signature = new String[]
      {
         Integer.class.getName()
      };

      try
      {
         mbServer.invoke(timerOName, "removeNotification", param, signature);
      }
      catch (Exception e)
      {
         throw new ServiceAgentException("Error removing notification: "
                                         + e.toString());
      }
   }
```

Adding Timer Notification Listener Methods

Finally, you add methods as shown in the following code block that facilitate adding notification listeners to the timer and removing notification listeners from the timer:

```
   public void addTimerListener(NotificationListener listener,
                                NotificationFilter filter,
                                Object handback)
      throws ServiceAgentException
   {
      try
      {
         mbServer.addNotificationListener(timerOName, listener,
                                          filter, handback);
      }
      catch (Exception e)
      {
         throw new ServiceAgentException("Error adding timer listener: "
                                         + e.toString());
      }
   }
```

```
public void removeTimerListener(NotificationListener listener)
   throws ServiceAgentException
{
   try
   {
      mbServer.removeNotificationListener(timerOName, listener);
   }
   catch (Exception e)
   {
      throw new ServiceAgentException("Error removing timer listener: "
                                      + e.toString());
   }
}
```

The Relation Service

The relation service is a framework of classes, interfaces, and MBeans, located in the `javax.management.relation` package, that outlines logical relations between MBeans. The relation service has a number of responsibilities. For example, the relation service

- Creates and deletes relation types and relations
- Maintains consistency between relations
- Provides a query mechanism for finding relations and MBeans that are referenced in relations
- Transmits notifications to all registered listeners when a relation is added, updated, or removed
- Registers as an MBean registration listener in order to maintain consistency between relations as MBeans are deregistered

Relationship Roles

Roles are represented by instances of the `javax.management.relation.Role` class. This class encapsulates a *role name* and a list of `ObjectNames` of MBeans that are referenced by the role. The list of `ObjectNames` is referred to as the *role value*.

Relations

MBean relations are expressed as n-ary associations between MBeans using named roles. Relations are objects that implement the `javax.management.relation.Relation` interface and represent logical associations between MBeans. They are defined by named relation types embodied in instances of objects that implement the `javax.management.relation.RelationType` interface. Relation types provide information about the named roles that they contain. Role information is encapsulated within `javax.management.relation.RoleInfo` objects that define

- The name of the role
- Whether or not the role is readable or writable
- The description of the role
- The minimum number of referenced MBeans in the role
- The maximum number of referenced MBeans in the role
- The name of the Java class that MBeans must be an instance of or extend in order to be associated with the role

Internal Relations

Relation types can be created internally to the relation service or externally. The relation service then exposes the list of previously defined relation types to objects interested in defining new relations.

Internal relations are created and maintained by the relation service. All objects of an internal relationship are inaccessible to outside objects. Thus, any object wishing to operate on an internal relation must do so by calling an operation on the relation service.

External Relations

External relations are objects created outside of the relation service. External relations must implement the `Relation` interface and be manually added to the relation service as MBeans. External relation MBeans can therefore be accessed in the same manner as any ordinary MBean.

Figure 3-5 illustrates the conceptual associations between the objects participating in the relation service.

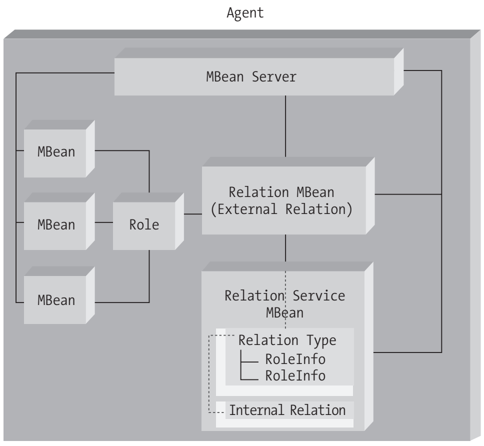

Figure 3-5. Conceptual associations between objects of the relation service

Creating and Starting the Relation Service

The relation service is created in the same manner as a standard MBean. That is, you supply one `ObjectName` for the `javax.management.relation.RelationService` MBean and register it with an MBean server. Thus, you might create and register the relation service MBean as follows:

```
public void initializeRelationService()
{
   try
   {
      Object[] params = new Object[1];
      params[0] = new Boolean(true);  // purge invalid relations immediately
      String[] signature = new String[1];
      signature[0] = "boolean";

      MBeanServer mbServer = MBeanServerFactory.createMBeanServer();

      ObjectName relationServiceOName =
         new ObjectName (mbServer.getDefaultDomain()
                        + ":name=relationService");

      mbServer.createMBean("javax.management.relation.RelationService",
                           relationServiceOName, params, signature);
   }
   catch(Exception e)
   {
      System.out.println(e.toString());
   }
}
```

Using the Relation Service

Let's look at how you might use the relation service. A simple relationship exists between an employee/worker, the employee's supervisor, and the employee's work location. When an employee changes location or positions, the employee's records need to reflect the changes. For example, when an employee changes positions, the employee's supervisor usually changes and sometimes the employee's work location.

You can easily model your employee MBean, your supervisor MBean, and your work location MBean, but modeling the relationships between them and maintaining those relationships can be daunting. Therefore, you will leave the task up to the relation service. You will add to your design the concept of a human resources (HR) relation that will use the relation service framework to handle the employee's associations. Figure 3-6 illustrates the conceptual associations between the objects participating in the HR relation.

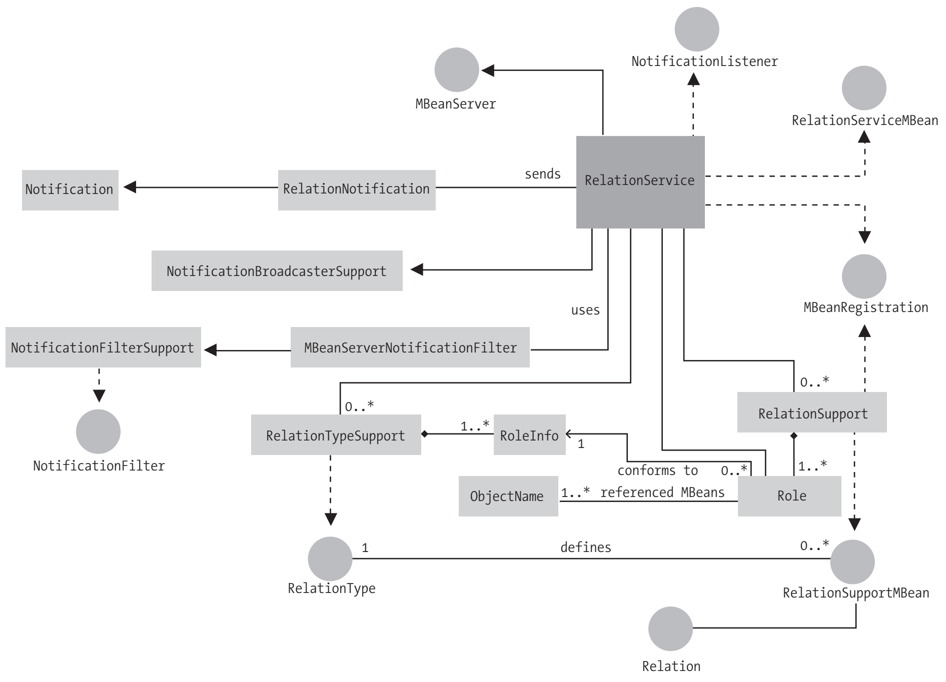

Figure 3-6. The static associations participating in the HR relation

Adding the Relation Service to the ServiceAgent

The first thing you need to do is import the relation classes as follows:

```
import javax.management.relation.*;
```

Now, let's add a convenience method for retrieving your relation-service domain name. You will simply use the default domain for now.

```
   public String getRelationServiceDomain()
   {
      return mbServer.getDefaultDomain();
   }
```

The next thing you will do is create and register the relation service in your service agent. Notice that you create and register the relation service in the same manner as a standard MBean. You pass one parameter, a Boolean, that specifies if you want each relation purged immediately when the relation becomes invalid.

```
   public void initializeRelationService()
   {
      try
      {
         relationServiceOName = new ObjectName(getRelationServiceDomain()
                                               + ":name=relationService");
         // Check to see if the relation service is already running.
         Set names = mbServer.queryNames(relationServiceOName, null);
         if (names != null && names.isEmpty() == false)
            return;

         Object[] params = new Object[1];
         params[0] = new Boolean(true);  // Purge invalid relations immediately.
         String[] signature = new String[1];
         signature[0] = "boolean";
         MBeanServer mbServer = MBeanServerFactory.createMBeanServer();
         mbServer.createMBean("javax.management.relation.RelationService",
                                   relationServiceOName, params, signature);
      }
      catch(Exception e)
      {
         System.out.println(e.toString());
      }
   }
```

I won't go into detail about each of the MBeans that are part of the relation, because their behavior is rather mundane; however, I want to point out to you the main points of each as follows:

- The `WorkerMBean` exposes attributes and operations to allow a worker's supervisor, salary, position, work location, etc., to be modified and retrieved.

- The `SupervisorMBean` exposes the same attributes and operations as the `WorkerMBean`, and exposes attributes and operations to add and remove team members and retrieve the size of the supervisor's team.

- The `WorkLocationMBean` exposes attributes and operations to retrieve the building name, the name of the administrative assistant, and the mail stop, and to add and remove employees.

- The `HRRelationMBean` exposes operations to modify an employee's supervisor, work location, and position while keeping relationships intact.

Now, let's look at the some of the details of the `HRRelationMBean`. The class declaration shows that you extend the `RelationSupport` class to benefit from the built-in features that it provides:

```
public class HRRelation extends RelationSupport
   implements HRRelationMBean, MBeanRegistration
```

You define four operations in your `HRRelationMBean` interface. The implementations of these operations use invocations on the MBean server to perform operations on the specific `Worker`, `Supervisor`, and `WorkLocation` MBeans. The four operations are as follows:

1. `define`: Creates all role objects, role information objects, and the actual relation objects

2. `transferEmployee`: Transfers an employee to a new work location and updates the employee's supervisor as needed

3. `changeEmployeePosition`: Changes an employee's position and updates the employee's work location and supervisor as needed

4. `replaceSupervisor`: Changes an employee's supervisor

Defining the HRRelation

The `define` method calls private methods on the `HRRelation` class that create the role information object, the role objects, and the relation object.

```
public void define()
   throws MBeanException
{
   createMBeans();
   RoleInfo[] roleInfo = createRoleInfos();
   ArrayList roleList = createRoles();
   createRelationType(roleInfo);
   createRelation(roleList);
}
```

The remainder of the methods implemented by the `HRRelation` MBean comprises either utility methods or the following methods that create the role objects, role information objects, and the actual relation.

Creating the MBeans for the HRRelation

The `createMBeans` method creates and registers the MBeans that will participate in the relation. You create a static list of MBeans for the sake of this example. In a production setting, the MBeans should be created dynamically as needed.

```
public void createMBeans()
   throws MBeanException
{
   try
   {
      // Register the worker.
      Object[] params = new Object[3];
      params[0] = new String("John');
      params[1] = new String("Doe");
      params[2] = new Integer(1); // Employee number
      String[] signature = new String[3];
      signature[0] = String.class.getName();
      signature[1] = String.class.getName();
      signature[2] = int.class.getName();

      ObjectName workerOName =
         new ObjectName(GetRelationServiceDomain()
                        + ":type=worker,empNum=1");
         mbServer.createMBean("com.apress.jhanson.hr.Worker",
                               workerOName, params, signature);

         // Register the supervisor.
         Object[] params1 = new Object[3];
         params1[0] = new String("Jane");
         params1[1] = new String("Smith");
         params1[2] = new Integer(2); // Employee number
         String[] signature1 = new String[3];
         signature1[0] = String.class.getName();
         signature1[1] = String.class.getName();
         signature1[2] = int.class.getName();

         ObjectName supervisorOName =
            new ObjectName(getRelationServiceDomain()
                              + ":type=supervisor,empNum=2");

         mbServer.createMBean("com.apress.jhanson.hr.Supervisor",
                               supervisorOName, params1, signature1);

         // Register the work location.
         Object[] params2 = new Object[3];
         params2[0] = new String("D");  // Building name
         params2[1] = new String("D100");  // Mail stop
         String[] signature2 = new String[3];
         signature2[0] = String.class.getName();
         signature2[1] = String.class.getName();

         ObjectName workLocationOName =
            new ObjectName(getRelationServiceDomain()
                              + ":type=worklocation");

         mbServer.createMBean("com.apress.jhanson.hr.WorkLocation',
                               workLocationOName, params2, signature2);

      }
      catch (Exception e)
      {
         throw new MBeanException(e);
      }
   }
```

Creating the RoleInfo Objects for the HRRelation

The `createRoleInfos` method creates the `RoleInfo` objects that define each role and the constraints for each role. You create `RoleInfo` objects for three roles: *Worker*, *Supervisor*, and *WorkLocation*.

```
public RoleInfo[] createRoleInfos()
   throws MBeanException
{
   RoleInfo[] roleInfos = new RoleInfo[3];

   try
   {
      roleInfos[0] = new RoleInfo("Worker",
                                          "com.apress.jhanson.hr.Worker",
                                          true, // Readable
                                          true, // Writable
                                          1, // Must have at least one
                                          100, // Can have 100, max
                                          "Worker role");

      roleInfos[1] = new RoleInfo("Supervisor",
                                          "com.apress.jhanson.hr.Supervisor",
                                          true, // Readable
                                          true, // Writable
                                          1, // Must have at least one
                                          1, // Can have 1, max
                                          "Supervisor role");

      roleInfos[2] = new RoleInfo("WorkLocation",
                                          "com.apress.jhanson.hr.WorkLocation",
                                          true, // Readable
                                          true, // Writable
                                          1, // Must have at least one
                                          1, // Can have 1, max
                                          "WorkLocation role");
   }
   catch (Exception e)
   {
      throw new MBeanException(e);
   }

   return roleInfos;
}
```

Creating the Relation Types for the HRRelation

The `createRelationType` method takes the array of `RoleInfo` objects created in the `createRoleInfos` method and forwards it to the `createRelationType` operation on the relation service to create the relation type. Give your relation type the name of `HRRelationType`.

```
public void createRelationType(RoleInfo[] roleInfos)
   throws MBeanException
{
   try
   {
      Object[] params = new Object[2];
      params[0] = "HRRelationType";
      params[1] = roleInfos;
      String[] signature = new String[2];
      signature[0] = "java.lang.String";
      signature[1] = (roleInfos.getClass()).getName();

      mbServer.invoke(relationServiceOName,
                      "createRelationType", params, signature);
   }
   catch (Exception e)
   {
      throw new MBeanException(e);
   }
}
```

Creating the Roles for the HRRelation

The `createRoles` method creates the roles that participate in your HR relation using the same names that you specified in the `createRoleInfos` method. The `createRoles` method also adds the names of the worker, supervisor, and location MBeans that participate in the relation.

```
public ArrayList createRoles()
   throws MBeanException
{
   try
   {
      ArrayList employeeRoleValue = new ArrayList();
      employeeRoleValue.add(new ObjectName(relationServiceDomain
                                     + ":name=worker,empNum=1"));
```

```
         employeeRoleValue.add(new ObjectName(relationServiceDomain
                                         + ":name=worker,empNum=2"));
         employeeRoleValue.add(new ObjectName(relationServiceDomain
                                         + ":name=worker,empNum=3"));
         Role employeeRole = new Role("Worker", employeeRoleValue);

         ArrayList supervisorRoleValue = new ArrayList();
         supervisorRoleValue.add(new ObjectName(relationServiceDomain
                                         + ":name=supervisor"));
         Role managerRole = new Role("Supervisor", supervisorRoleValue);

         ArrayList workLocationRoleValue = new ArrayList();
         workLocationRoleValue.add(new ObjectName(relationServiceDomain
                                             + ":name=worklocation"));

         Role physLocationRole = new Role("WorkLocation",
                                    workLocationRoleValue);

         ArrayList roleList = new ArrayList();
         roleList.add(employeeRole);
         roleList.add(managerRole);
         roleList.add(physLocationRole);

         return roleList;
      }
      catch (Exception e)
      {
         throw new MBeanException(e);
      }
   }
```

Creating the Actual HRRelation

The `createRelation` method creates and registers your `HRRelation` MBean with the relation type you created earlier. The `HRRelation` MBean is then added to the relation service using the `addRelation` operation of the relation service MBean.

```
   public void createRelation(ArrayList roleList)
      throws MBeanException
   {
      try
      {
         Object[] params = new Object[4];
         params[0] = "HRRelation";
         params[1] = relationServiceOName;
         params[2] = "HRRelationType";
         params[3] = roleList;

         String[] signature = new String[4];
         signature[0] = "java.lang.String";
         signature[1] = relationServiceOName.getClass().getName();
         signature[2] = "java.lang.String";
         signature[3] = roleList.getClass().getName();

         ObjectName relationMBeanName =
               new ObjectName(relationServiceDomain + ":type=RelationMBean");

         mbServer.createMBean("com.apress.jhanson.hr.HRRelation",
                              relationMBeanName,
                              params, signature);

         // Add the relation.
         params = new Object[1];
         signature = new String[1];
         params[0] = relationMBeanName;
         signature[0] = "javax.management.ObjectName";

         mbServer.invoke(relationServiceOName,
                         "addRelation", params, signature);
      }
      catch (Exception e)
      {
         throw new MBeanException(e);
      }
   }
```

Summary

The agent level defined by the JMX specification targets the management solutions development community by providing a comprehensive and flexible framework for building management agents. Agents expose instrumented resources in a manner that allows management applications to discover them and invoke operations on them in a standard fashion. An agent can reside in a local JVM or in a remote JVM. This makes it possible to build management applications that interact with agents and resources in a generic way, thus allowing management applications the ability to administrate any system that adheres to this framework.

In addition to facilitating access to instrumented resources, an agent broadcasts notifications to interested notification receivers that have previously registered with it. An agent exposes an MBean server, at least one protocol adaptor or connector, and several mandatory services, known as *agent services*. The mandatory agent services are registered as MBeans and include a monitoring service, a timer service, a relation service, and a dynamic class-loading service.