5.3 The MBeanServer Interface

Now that we know how to create an MBeanServer and how to name MBeans, it's time to explore the functionality provided by the MBeanServer itself. In this section we'll examine the core of the `MBeanServer` interface: the methods it provides for creating and manipulating MBeans, and the methods it provides for retrieving information about itself. Subsequent sections will discuss the MBeanServer's query methods and the notification facility.

As we mentioned earlier, an MBeanServer's primary responsibility is to maintain a registry of MBeans. The registry provides a common naming model, based on object names, across all management applications and makes its MBeans, and the manageable resources they represent, available to multiple management applications. MBeans become available to JMX management applications when they are added to an MBeanServer's registry, and when an MBean is removed from the registry it is no longer available to those applications. An MBean is registered with an MBeanServer, and the attributes and methods in its management interface are accessible via MBeanServer methods.

5.3.1 MBean Lifecycle Methods

The MBeanServer provides a set of *lifecycle methods.* These methods provide the mechanism for adding MBeans to the MBeanServer's registry and removing MBeans from that registry. In addition, there are methods that allow management applications to instantiate new MBeans via the MBeanServer.

5.3.1.1 Instantiation

At the beginning of this chapter we said that an MBean's management lifecycle begins when it is registered with an MBeanServer. Of course an MBean has to exist before it can be registered. One obvious way to instantiate an MBean is to call one of its constructors directly. The MBeanServer also provides a suite of methods that management applications can use to create new MBeans without directly invoking a constructor:

```
public Object instantiate(String classname)
public Object instantiate(String classname,
                          Object[] params,
                          String[] signature)
public Object instantiate(String classname, Obj
public Object instantiate(String classname,
                          ObjectName loader,
                          Object[] params,
                          String[] signature)
```

If the class named in the first argument has a no-args constructor, no additional arguments are required; the designated class will be loaded and the no-args constructor invoked to create a new instance. If the class doesn't provide a no-args constructor, or if the management application needs to use a different constructor, we can implicitly specify one by providing a signature and the actual arguments to the constructor in the `signature` and `params` arguments. After the class is loaded, the constructor with the specified signature will be invoked with the values given in the `params` array to generate a new instance of the class.

What about the `loader` parameter? The `loader` parameter indicates the object name of the MLet (management applet) that should be used to load the specified class. (See Chapter 7 for the details on MLets, or for now, just think of them as a form of class loader.) When a loader's object name is specified, we use the MLet referred to by that name to load the class.

What about when no loader is specified? In that case the MBeanServer turns to `DefaultLoaderRepository`. `DefaultLoaderRepository` is the collection of loaders that includes the class loader that loaded the MBeanServer and all of the MLets registered with the MBeanServer. When no loader is specified, the MBeanServer iterates over `DefaultLoaderRepository`'s collection of loaders looking for one that can load the specified class. If it finds one, the class instance is created as described already; if not, an instance of `ReflectionException` that wraps `java.lang.ClassNotFoundException` is thrown.

This is all well and good, but why bother with `instantiate()`? Why not just use the class's constructor without all this indirection? The reason is that the management application's class loader may not have access to the desired class. We won't go into all the details here, but the bottom line is that the MLet mechanism allows us to specify additional code sources that the MBeanServer's `instantiate()` and `createMBean()` methods can use to load classes. So in some cases even though the application's class loader may not know how to load a given class, the MBeanServer does have the necessary information and the application can use its `instantiate()` method to create instances of that class.

5.3.1.2 Registration

Once we have a reference to an instance of an MBean, it can be registered through this statement:

```
ObjectInstance registerMBean(Object mbean, Obje
```

The `registerMBean()` method takes MBean's object reference and an `ObjectName` instance as parameters, creates an entry associating the two in the MBeanServer's registry, and returns

an instance of `ObjectInstance` that maps `ObjectName` to the underlying Java class, as discussed earlier.

5.3.1.3 Creation

The "instantiate, then register" approach just described is a natural one for Java programmers used to working with `Collection` classes and the like, but it has the disadvantage of leaving a "live" reference to the MBean in the management application. JMX solves this problem by providing MBeanServer methods that, from a management application's perspective, automatically instantiate and register an instance of an MBean. The signatures for these methods are shown here:

```
ObjectInstance createMBean(String mbeancls,
                           ObjectName objname);
ObjectInstance createMBean(String mbeancls,
                           ObjectName objname,
                           Object[] params,
                           String[] sig);
ObjectInstance createMBean(String mbeancls,
                           ObjectName objname,
                           ObjectName loader);
ObjectInstance createMBean(String mbeancls,
                           ObjectName objname,
                           ObjectName loader,
                           Object[] params,
                           String[] sig);
```

The `mbeancls` and `objname` parameters that are common to all of these methods specify the fully qualified class name for the MBean and the object name under which an instance of that class is to be registered, respectively. As in the `instantiate()` methods, the `loader` parameter is used to

indicate the object name of the MLet that should be used to load the MBean's class.

When the `params` and `sig` array pairs are not present, `createMBean()` uses Java reflection to invoke the no-args constructor on the specified class and registers the resulting instance under the given `ObjectName` instance. When the `params` and `sig` parameters are present, the `mbeancls` constructor whose parameter types match the class name strings in the `sig` array is invoked, again via Java reflection, with the parameters provided by the `params` array. Note that the class name strings in the `sig` array must be the *fully qualified* class names for the parameters; for example, if there is a `String` parameter in the `params` array, the corresponding element in the `sig` array must be "java.lang.String".

For example, consider the following simple MBean interface:

```
public interface SimpleSwitchMBean {
  public void flip();
  public int getState();
}
```

and the `SimpleSwitch` class that implements it:

```
public class SimpleSwitch implements SimpleSwit
  private int state;

  public SimpleSwitch() {this(0); }
  public SimpleSwitch(int state) {this.state =

  public void flip() {state = (state == 0) ? 1
  public int getState() {return state; }
}
```

The following code registers three separate `SimpleSwitch` MBeans with an MBeanServer:

```
MBeanServer mbs = MBeanServerFactory.createMBea

// Instantiate and register
ObjectName sone = new ObjectName("book:name=Swi
mbs.registerMBean(new SimpleSwitch(), sone);

// Create via no-args constructor
ObjectName stwo = new ObjectName("book:name=Swi
mbs.createMBean("SimpleSwitch", stwo);

// Create via specific constructor
ObjectName sthree = new ObjectName("book:name=S
Object[] params = {new Integer(1) };
String[] sig = {"java.lang.Integer" };
mbs.createMBean("SimpleSwitch", sthree, params,
```

Even though the calls are relatively simple, what's going on under the covers is complex, and lots of things can go wrong. When something does go wrong, an exception is thrown to indicate there is a problem. All told, `createMBean()` has six possible exceptions in its `throws` clause, and `registerMBean()` has four. By far the most common are

- `ReflectionException`, which wraps an exception that occurred while `createMBean()` was trying to invoke the MBean's constructor. Usually the wrapped exception is an instance of `java.lang.ClassNotFoundException`. `ReflectionException` provides a `getTargetException()` method that returns the wrapped exception.

- `MBeanException`, which indicates that the MBean's constructor threw an exception. `MBeanException` also provides a `getTargetException()` method that returns the exception thrown by the constructor.

- `RuntimeOperationsException`, which generally wraps an instance of `IllegalArgumentException` and indicates that one of the parameters of `createMBean()` or `registerMBean()` was bogus.

5.3.1.4 The MBeanRegistration Interface

Once a JMX developer has a handle on creating MBeans and adding them to an MBeanServer, he inevitably confronts one or more of the following questions:

- How do I know when my MBean is (de)registered?

- How does my MBean get access to the MBeanServer it's registered with?

- How does my MBean know its name?

- How can I prevent my MBean from being registered with an MBeanServer?

- How can I specify an action for my MBean to take immediately after it is (de)registered?

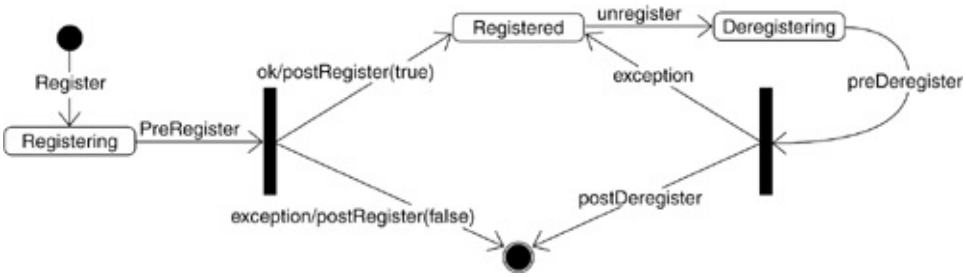
Although these questions could be answered in a variety of ad hoc ways, the JMX specification defines the `MBeanRegistration` interface to address each of them in a simple, flexible, and consistent manner. The

`MBeanRegistration` interface is composed of four methods:

```
ObjectName preRegister(MBeanServer mbserver, Ob
void postRegister(Boolean registered);
void preDeregister();
void postDeregister();
```

As the names suggest, these methods are invoked before and after registration and before and after deregistration, respectively. The state chart in Figure 5.2 illustrates the invocation of each of these methods as an MBean moves through its lifecycle.

Figure 5.2. MBean Registration States and Actions



When an MBean is registered, the MBeanServer inspects it to see if it implements the `MBeanRegistration` interface. If it does, the MBean's `preRegister()` method is called with a reference to the MBeanServer it is being registered with and a proposed object name. The `ObjectName` instance is the parameter passed to the `createMBean()` or `registerMBean()` method that started the registration process. If that `ObjectName` instance is `null`, then the MBean must provide its own object name; otherwise the proposed name may be accepted unchanged or augmented by the MBean —for example, by the addition of other key properties. In either case the name that is ultimately chosen and returned by `preRegister()` is the name under which the MBean is registered. If the MBean will need access to the MBeanServer—

for example, to inspect the attributes or invoke the methods of other MBeans—it should save the reference provided for that purpose.

The MBeanServer catches any exception thrown by the `preRegister()` method. If the exception is an instance of `MBeanRegistrationException`, it is simply rethrown. Otherwise the actual exception is wrapped in `MBeanRegistrationException`, which is then thrown by the MBeanServer. When `preRegister()` throws an exception, the MBean is not registered with the MBeanServer.

The `postRegister()` method is invoked immediately after registration. If the registration succeeded, `postRegister()`'s `registered` parameter is `true`. If the registration failed—for example, if `preRegister()` threw an exception—`registered` is `false`. MBeans should use `postRegister()` to perform additional setup and configuration actions: adding notification listeners, creating additional MBeans, allocating resources, and so on.

The `preDeregister()` and `postDeregister()` methods are the deregistration analogs of `preRegister()` and `postRegister()`, respectively. The MBeanServer invokes `preDeregister()` prior to removing an MBean from its registry, and `postDeregister()` immediately after an MBean is deregistered. Exceptions thrown by `preDeregister()`, like those thrown by `preRegister()`, are caught by the MBeanServer, wrapped in `MBeanRegistrationException` and rethrown. If an exception is thrown by `preDeregister()`, the MBean is not deregistered and `postDeregister()` is not invoked.

Note that throwing an exception in `preRegister()` prevents the MBean from being *registered,* whereas throwing an exception in `preDeregister()` prevents the MBean from

being *deregistered.* The former behavior can be useful when the MBean's environmental requirements aren't satisfied—for example, when the database it needs to connect to isn't up, or a license for the MBean can't be obtained from a license server. The latter behavior can be used to prevent an active MBean from being deregistered until either all outstanding requests have been serviced or the service that the MBean manages has been shut down.

5.3.1.5 Deregistration

MBeans are removed from the MBeanServer's registry with the following statement:

```
void unregisterMBean(ObjectName objname);
```

Deregistering an MBean makes it inaccessible through the `MBeanServer` interface. From a JMX perspective the resource represented by the MBean is no longer manageable. Of course, if the MBean was instantiated outside the MBeanServer and then registered, any reference to it retained by the application is still valid. MBeans created by a call to one of the `createMBean()` methods become candidates for garbage collection.

5.3.2 MBean Access Methods

The MBeanServer provides a common interface to the MBeans in its registry. That interface consists of six methods:

```
Object getAttribute(ObjectName objname, String
AttributeList getAttributes(ObjectName objname,
void setAttribute(ObjectName objname, Attribute
AttributeList setAttributes(ObjectName objname,
```

```
    Attribute[] attrs);
Object invoke(ObjectName objname,
              String method,
              Object[] params,
              String[] sig);
MBeanInfo getMBeanInfo(ObjectName objname);
```

The `ObjectName` parameters in these APIs refer to MBeans registered with the MBeanServer. The `getAttribute()` and `getAttributes()` methods return the values of the specified MBean's attributes, `setAttribute()` and `setAttributes()` set the values of the specified MBean's attributes, and `invoke()` executes an operation on the specified MBean.

Three additional convenience methods exist to simplify common management application tasks:

```
ObjectInstance getObjectInstance(ObjectName obj
boolean isInstanceOf(ObjectName objname, String
boolean isRegistered(ObjectName objname);
```

The `getObjectInstance()` method returns the `ObjectInstance` for the specified object name or throws an `InstanceNotFoundException` if an MBean is not registered under that name. As we discussed earlier, `ObjectInstance` provides a mapping between an MBean's object name and the MBean's underlying Java class. Given an arbitrary MBean's object name, a management application can get the name of the Java class that implements it as follows:

```
String cls = mbs.getObjectInstance(objname).get
```

where `mbs` is a reference to an MBeanServer and `objname` is the object name of the MBean whose implementation class we are interested in.

The `isInstanceOf()` method answers the question, Is the MBean associated with this object name implemented by the specified class? Like `getObjectInstance()`, `isInstanceOf()` throws an `InstanceNotFoundException` if an MBean is not registered under the given object name.

Finally, `isRegistered()` gives us a way to check whether an MBean is registered with the specified object name before invoking `getObjectInstance()` or `isInstanceOf()`.

5.3.3 MBeanServer Methods

The MBeanServer also provides a couple of convenience methods that provide access to information about the MBeanServer itself:

```
public String getDefaultDomain()
public Integer getMBeanCount()
```

These methods do what their names imply. The `getDefaultDomain()` method returns the MBeanServer's default domain string, and `getMBeanCount()` returns the number of MBeans currently registered with the MBeanServer. Both methods tend to be used in JMX "console" applications that provide an MBeanServer GUI.

Chapter 7. JMX Agent Services

Agent services are MBeans that are distinguished from other MBeans that represent managed resources by the roles they play and functions they provide rather than by the interfaces they support. A service's role may be management oriented; for example, monitoring attribute values is an obvious management-oriented service that is supported in JMX by the monitor agent services described in Chapter 6. Another management-oriented service is a *status aggregator* that examines the status of a set of related MBeans to determine the status of a higher-level MBean in an application. Alternatively, an agent service may play a more utilitarian role, like the XML service described in Chapter 4 that supports management applications by instantiating model MBeans from an XML file and persisting model MBeans to an XML file. In this chapter we describe the JMX agent services that must be provided by every JMX implementation: the timer service, the MLet service, and the relation service. We also discuss the connector services that provide remote access to an otherwise local MBeanServer.

Chapter 6. Monitors and Monitoring

Monitoring is an essential activity for a management application. Instrumentation provides raw information about the status of an application or device. Administrators specify acceptable values, or ranges of values, for the status information the instrumentation provides. In many cases the administrator would also like to specify an action to take if the status information indicates a problem. For example, an important aspect of managing a database is ensuring that the file system that holds the database's tables has sufficient space; if the file system begins to fill up, an e-mail or a page needs to be sent to the administrator so that the appropriate action—for example, extending the file system or archiving data—can be taken.

Management applications use monitors to automate administration; they configure their monitors to check an application's status information periodically and send notifications if the values don't meet the administrator's specifications. The management application reacts to those notifications, perhaps by taking some prearranged corrective action or by sending e-mail to, or even paging, a human administrator.

Because monitoring plays such a central role in management applications, JMX provides a set of monitor services that simplify the task of building sophisticated management applications.

Chapter 4. Model MBeans

4.1 Introduction

Creating instrumentation with MBeans from scratch can be time-consuming and tedious. The JMX model MBean specification[1] provides generic instrumentation that can be quickly customized for many resources. This chapter describes the model MBean in detail and provides multiple examples of its use in practice.

A model MBean is a fully customizable dynamic MBean. The `RequiredModelMBean` class, which implements the `ModelMBean` interface, is guaranteed to exist on every JMX agent. A model MBean implements the `ModelMBean` interface that extends a set of interfaces including `DynamicMBean`, `PersistentMBean`, and `ModelMBeanNotificationBroadcaster`.

Like the open MBean, the model MBean has it's own extended `MBeanInfo` interface called `ModelMBeanInfo`. `ModelMBeanInfo` associates new, extensible metadata objects called *descriptors* with the management interface. Descriptors add additional information about the managed resource and are also used to customize the behavior of a `RequiredModelMBean` instance. In this chapter, when we say simply "model MBeans" we are referring to implementations of the model MBean or `RequiredModelMBean`. When we are referring to the `ModelMBean` interface, we will say "the `ModelMBean` interface."

Model MBeans were added to JMX for several reasons:

- **Ease of use for developers**. Because model MBeans have an actual implementation class, `RequiredModelMBean`, as part of every JMX implementation, developers don't need to write their entire MBean and address all of the

accompanying support for logging, notifications, errors, persistence, and so on. All of this is provided as part of the `RequiredModelMBean` implementation. Because it is an actual implementation, you can create subclasses for it and override its behavior with your own. Model MBeans make it possible to instrument your managed resource in as little as five lines of code (not including configuration of the model MBean).

- **Ease of support by development tools**. Because model MBeans are fully configurable through metadata kept in descriptors, the creation of `ModelMBeanInfo` requires either a lot of programmatic setting of metadata or a lot of XML[2] in a file. However, the use of metadata with a portable XML file or API to create it makes it much easier to create and maintain model MBeans with development tools. Part of the intent for model MBeans was to enable the creation of management objects during development by development tools and wizards. Eventually we hope that developing the management objects and interfaces for your application will be as much a part of your application development cycle as developing remote interfaces and logging or tracing support.

- **Common services**. Model MBeans support a common set of behaviors in all implementations, including logging, generic notifications, attribute change notifications, persistence, and value caching. These behaviors can be customized and overridden.

- **Dynamic behavior**. Model MBeans are completely customized during development or during runtime. During development, the `ModelMBeanInfo` class supports APIs to customize the management interface and behavior of the model MBean. During runtime, external files, properties files, or XML files can be used to customize the model

MBean. Likewise the resource may use the `ModelMBeanInfo` APIs to customize the model MBean so that it conforms to its own runtime factors.

- **Isolation**. Model MBeans isolate the application manageability instrumentation from differences in Java runtimes and JMX implementations. Manageable components today may be installed and used in different editions of Java: J2SE,[3] J2EE,[4] and eventually J2ME.[5] These different JVMs may use different approaches when providing common services to model MBeans. For example, a J2SE JMX `RequiredModelMBean` implementation may support persistence by writing the data to files. If persistence to a database were required, then `RequiredModelMBean` would use JDBC[6] directly. However, a J2EE JMX `RequiredModelMBean` may be implemented as a local EJB[7] or wrapped as a remote EJB and use container-managed persistence. As JMX becomes more pervasive in the J2ME JVMs, these differences in MBean behavior will become more apparent.

  In a J2ME JVM, persistence and logging may not be allowed at all if the JVM is run on a diskless device. Even value caching in memory may be too costly. If you were writing your own MBeans for your component, you would have to write very different MBeans for each of these scenarios. However, because you use the MBeanServer as a factory to instantiate the `RequiredModelMBean` class that has been supplied by the JMX implementation, your use of the MBean in your managed resource does not change. The JMX implementation is required to make sure that the `RequiredModelMBean` implementation is appropriate for the installation and current JVM environment. Your investment in instrumentation of your managed resources is protected and requires less development and maintenance.

4.1.1 The Simplest Model MBean Example

In this section we will look at a simple example of how to use a model MBean. This example shows how to make a resource manageable in as little as five lines of code. In this example we expose attributes and operations of the managed resource and send a notification upon an error condition.

Recall from Chapter 3, on MBeans, that the Apache server application had its own standard MBean. Let's look at what it would take to create a model MBean for the Apache server. We will have to write a Java utility, `ApacheServer`, to get the information from the Apache module and return it to the model MBean. In our scenario, the `ApacheServer` class is similar to the `Apache` implementation, but it does not implement the `ApacheMBean` class, and it does not have to support value caching (see Code Block 3 in the text that follows). The model MBean will provide caching support at a much finer grain. The caching policy will be defined in the attribute descriptors in `ModelMBeanInfo`. Therefore, we don't support the `isCaching()`, `getCacheLifetime()`, `setCacheLifetime()`, `getCaching()`, and `setCaching()` methods in the original standard MBean interface. The `ApacheServer` class is the managed resource.

The first code block that follows shows how to find the MBeanServer, instantiate the model MBean, instantiate `ApacheServer`, associate it with the model MBean, configure the model MBean, and send a notification if the server is down at this time. The second code block shows how to configure the model MBean through `ModelMBeanInfo`. Notice that we can selectively cache the data from the Apache server, choosing not to cache some of the more time-sensitive counters. The third code block shows the slightly modified `Apache` class from Chapter 3. All of the examples in this book (plus some others) are available on the book's Web site:

http://www.awprofessional.com/titles/0672324083.

Code Block 1

```
package jnjmx.ch4;

import javax.management.*;
import javax.management.modelmbean.*;
import java.lang.reflect.Constructor;
import java.util.ArrayList;

/**
 * @author Heather Kreger
 * Chapter 4:  Java and JMX Building Manageable
 * ApacheServerManager: Manages the ApacheServe
 * resource using a model MBean.
 *
 * ModelMBeanInfo is created using ModelMBeanIn
 * rather than an XML file.
 */

public class ApacheServerManager {

  public static void main(String[] args) {
    boolean tracing = false;
    String serverURL = "";
    try {

      if (args.length != 0) {
        System.out.println(args[0]);
        if (args[0] != null)
          serverURL = serverURL.concat(args[0])
        else
```

```
    serverURL = "http://www.apache.org/se
}
// This serverURL default mechanism is ju
// We would not recommend this in a real
if (!serverURL.startsWith("http"))
  serverURL = "http://www.apache.org/serv

/** Find an MBeanServer for our MBean */
MBeanServer myMBS = MBeanServerFactory.cr

/** Instantiate an unconfigured RequiredM
RequiredModelMBean apacheMMBean =
  (RequiredModelMBean) myMBS.instantiate(
    "javax.management.modelmbean.Required

/** Create a name for the MBean
 * domain = apacheManager
 * one attribute: id = ApacheServer*/
ObjectName apacheMBeanName =
  new ObjectName("apacheManager: id=Apach

/** Instantiate the ApacheServer utility
jnjmx.ch4.ApacheServer myApacheServer =
  new jnjmx.ch4.ApacheServer(serverURL);

/** Set the configuration of the ModelMBe
 ** This method is below */
ApacheServerManager asmgr = new ApacheSer
ModelMBeanInfo apacheMMBeanInfo =
  asmgr.createApacheModelMBeanInfo(
    myApacheServer,
    apacheMBeanName);
apacheMMBean.setModelMBeanInfo(apacheMMBe
```

```
      // ** Set the ApacheServer as the managed
      apacheMMBean.setManagedResource(myApacheS

      // ** Register the ModelMBean instance th
      // MBeanServer
      ObjectInstance registeredApacheMMBean =
        myMBS.registerMBean((Object) apacheMMBe

      // ** Use the ModelMBean to send the noti
      // sent to a manager adapter that is regi
      // Apache MBeans and sends them to a page

      String apacheStatus =
        (String) apacheMMBean.invoke("getState"
      if ((apacheStatus.equals("DOWN"))
        || (apacheStatus.equals("NOT_RESPONDING
        System.out.println("Apache Server is Do
        apacheMMBean.sendNotification("Apache S
        /*  You could create monitor to notify
         * again so Web traffic can be routed b
      } else
        System.out.println("Apache Server is "
    } catch (Exception e) {
      System.out.println(
        "Apache Server Manager failed with " +
    }
    System.exit(0);
  }
```

Because model MBeans support the `NotificationBroadcaster` interface, when the `getStatus()` method returns `Down` or `Not Responding`, we

have the model MBean send a notification to the MBeanServer, which in turn sends it to all registered notification listeners. This is fairly simple to do; we merely

1. **Find our `MBeanServer`.**

- Instantiate the `RequiredModelMBean` class using the MBeanServer.

- Customize `RequiredModelMBean` in two steps:

a. **Set up the `ModelMBeanInfo instance`.**

- Associate the model MBean with the access to the actual managed resource.

- Register `RequiredModelMBean` with the MBeanServer.

- Invoke the model MBean to get the `Status` attribute.

- Send the notification if the status is `Down` or `Stopped`. In the downloadable examples for the book, we provide an example of a notification listener for this event that you can experiment with.

Here's how we set the `ModelMBeanInfo` instance that we used to customize this model MBean. The `ModelMBeanInfo` object created for the ApacheServer managed application used in the Apache model MBean defines the following:

- `String` attributes that are read-only: `BusyWorkers`, `BytesPerSec`, `BytesPerReq`, `IdleWorkers`, `ReqPerSec`, `Scoreboard`, `State`, `TotalAccesses`, `TotalKBytes`, and `Uptime`

- `String` attributes that are read/write: `Server`
- Operations: `start` and `stop`
- Notification: ApacheServer `Down`
- Operations to support the attributes on the `ApacheServer` class: `getBusyWorkers()`, `getBytesPerReq()`, `getBytesPerSec()`, `getIdleWorkers()`, `getReqPerSec()`, `getScoreboard()`, `getServer()`, `getState()`, `getTotalAccesses()`, `getTotalKBytes()`, `getUptime()`, `setServer(String URL)`

Code Block 2 illustrates the programmatic way to create a `ModelMBeanInfo` instance. `ModelMBeanInfo` can also be initialized from data saved in a file. This is shown in the XML primer examples later in this chapter (see Section 4.7). It is much simpler code, but it requires the use of an XML service that is not a standard service of JMX. This entire method is relatively lengthy, but it is very simple to program and it is easy for tools to generate. The method in its entirety is listed here. We will explain how to set an attribute, an operation, and a notification in the next section.

Code Block 2

```
// Create the ModelMBeanInfo for the Apache ser
  ModelMBeanInfo createApacheModelMBeanInfo(
    ApacheServer managedApache,
    ObjectName apacheMBeanName) {

    // Set the ApacheServer's class name, there
    // more flexible ways to do this
```

```
Class apacheClass = null;
try {
  apacheClass = Class.forName("jnjmx.ch4.Ap
} catch (Exception e) {
  System.out.println("ApacheServer Class no
}

// Set the MBean's descriptor with default
// MBean name is apacheMBeanName
// logging notifications to jmxmain.log
// caching attributes for 10 seconds
Descriptor apacheDescription =
  new DescriptorSupport(
    new String[] {
      ("name=" + apacheMBeanName),
      "descriptorType=mbean",
      ("displayName=ApacheServerManager"),
      "type=jnjmx.ch4.ApacheServer",
      "log=T",
      "logFile=jmxmain.log",
      "currencyTimeLimit=10" });

// Define attributes in ModelMBeanAttribute

ModelMBeanAttributeInfo[] apacheAttributes
  new ModelMBeanAttributeInfo[11];

// Declare BusyWorkers attribute
// cache BusyWorkers for 3 seconds,
// use get method
Descriptor busyWorkersDesc =
  new DescriptorSupport(
    new String[] {
```

```
        "name=BusyWorkers",
        "descriptorType=attribute",
        "displayName=Apache BusyWorkers",
        "getMethod=getBusyWorkers",
        "currencyTimeLimit=3" });

apacheAttributes[0] =
  new ModelMBeanAttributeInfo(
    "BusyWorkers",
    "int",
    "Apache Server Busy Workers",
    true,
    false,
    false,
    busyWorkersDesc);

// Declare BytesPerSec attribute
// Cache ByetesperSec for 10 seconds
Descriptor bytesPerSecDesc =
  new DescriptorSupport(
    new String[] {
      "name=BytesPerSec",
      "descriptorType=attribute",
      "displayName=Apache BytesPerSec",
      "getMethod=getBytesPerSec",
      "currencyTimeLimit=10" });

apacheAttributes[1] =
  new ModelMBeanAttributeInfo(
    "BytesPerSec",
    "int",
    "Apache Server Bytes Per Sec",
    true,
```

```
    false,
    false,
    bytesPerSecDesc);

// Declare BytesPerReq attribute
Descriptor bytesPerReqDesc =
  new DescriptorSupport(
    new String[] {
      "name=BytesPerReq",
      "descriptorType=attribute",
      "displayName=Apache BytesPerReq",
      "getMethod=getBytesPerReq",
      "currencyTimeLimit=10" });

apacheAttributes[2] =
  new ModelMBeanAttributeInfo(
    "BytesPerReq",
    "int",
    "Apache Server Bytes Per Request",
    true,
    false,
    false,
    bytesPerReqDesc);

// Declare IdleWorkers attribute
Descriptor idleWorkersDesc =
  new DescriptorSupport(
    new String[] {
      "name=IdleWorkers",
      "descriptorType=attribute",
      "displayName=Apache IdleWorkers",
      "getMethod=getIdleWorkers",
      "currencyTimeLimit=10" });
```

```
apacheAttributes[3] =
  new ModelMBeanAttributeInfo(
    "IdleWorkers",
    "int",
    "Apache Server Idle Workers",
    true,
    false,
    false,
    idleWorkersDesc);

// Declare ReqPerSec attribute
Descriptor reqPerSecDesc =
  new DescriptorSupport(
    new String[] {
      "name=ReqPerSec",
      "descriptorType=attribute",
      "displayName=Apache ReqPerSec",
      "getMethod=getReqPerSec",
      "currencyTimeLimit=5" });

apacheAttributes[4] =
  new ModelMBeanAttributeInfo(
    "ReqPerSec",
    "int",
    "Apache Server Requests Per Second",
    true,
    false,
    false,
    reqPerSecDesc);

// Declare Scoreboard attribute
Descriptor ScoreboardDesc =
```

```
    new DescriptorSupport(
      new String[] {
        "name=Scoreboard",
        "descriptorType=attribute",
        "displayName=Apache Scoreboard",
        "getMethod=getScoreboard",
        "currencyTimeLimit=10" });

  apacheAttributes[5] =
    new ModelMBeanAttributeInfo(
      "Scoreboard",
      "java.lang.String",
      "Apache Server Scoreboard",
      true,
      false,
      false,
      ScoreboardDesc);

  // Declare TotalAccesses attribute
  // Do not cache TotalAccesses
  Descriptor totalAccessesDesc =
    new DescriptorSupport(
      new String[] {
        "name=TotalAccesses",
        "descriptorType=attribute",
        "displayName=Apache TotalAccesses",
        "getMethod=getTotalAccesses",
        "currencyTimeLimit=-1" });

  apacheAttributes[6] =
    new ModelMBeanAttributeInfo(
      "TotalAccesses",
      "int",
```

```
    "Apache Server total accesses",
    true,
    false,
    false,
    totalAccessesDesc);

// Declare TotalKBytes attribute
// Do not cache TotalKBytes
Descriptor totalKBytesDesc =
  new DescriptorSupport(
    new String[] {
      "name=TotalKBytes",
      "descriptorType=attribute",
      "displayName=Apache TotalKBytes",
      "getMethod=getTotalKBytes",
      "currencyTimeLimit=-1" });

apacheAttributes[7] =
  new ModelMBeanAttributeInfo(
    "TotalKBytes",
    "int",
    "Apache Server total KiloBytes",
    true,
    false,
    false,
    totalKBytesDesc);

// Declare Uptime attribute
Descriptor uptimeDesc =
  new DescriptorSupport(
    new String[] {
      "name=Uptime",
      "descriptorType=attribute",
```

```
        "displayName=Apache Uptime",
        "getMethod=getUptime",
        "currencyTimeLimit=10" });
apacheAttributes[8] =
  new ModelMBeanAttributeInfo(
    "Uptime",
    "java.lang.String",
    "Apache Server Up Time",
    true,
    false,
    false,
    uptimeDesc);

// Declare State attribute
// State has a getMethod
Descriptor stateDesc =
  new DescriptorSupport(
    new String[] {
      "name=State",
      "descriptorType=attribute",
      "displayName=Apache State",
      "getMethod=getState",
      "currencyTimeLimit=10" });

apacheAttributes[9] =
  new ModelMBeanAttributeInfo(
    "State",
    "java.lang.String",
    "Apache Server state",
    true,
    false,
    false,
    stateDesc);
```

```
// Declare Server attribute
// Server has a getMethod and a setMethod
Descriptor serverDesc =
  new DescriptorSupport(
    new String[] {
      "name=Server",
      "descriptorType=attribute",
      "displayName=Apache Server URL",
      "getMethod=getServer",
      "setMethod=setServer",
      "currencyTimeLimit=10" });

  apacheAttributes[10] =
    new ModelMBeanAttributeInfo(
      "Server",
      "java.lang.String",
      "Apache Server Busy Workers",
      true,
      true,
      false,
      serverDesc);

// Declare constructors for the managed res
// one constructor which accepts one parame
// a String URL
Constructor[] myConstructors = apacheClass.

ModelMBeanConstructorInfo[] apacheConstruct
  new ModelMBeanConstructorInfo[1];

MBeanParameterInfo[] constructorParms =
  new MBeanParameterInfo[] {
```

```
    (new MBeanParameterInfo("serverURL",
    "java.lang.String",
    "Apache Server URL"))};

Descriptor apacheBeanDesc =
  new DescriptorSupport(
    new String[] {
      "name=ApacheServer",
      "descriptorType=operation",
      "role=constructor" });

apacheConstructors[0] =
  new ModelMBeanConstructorInfo(
    "Apache",
    "ApacheServer(): Constructs an ApacheSe
    constructorParms,
    apacheBeanDesc);

// Define operations in ModelMBeanOperation
/* Operations: getBusyWorkers, getBytesPerS
  * getIdleWorkers, getReqPerSec, getScoreb
  * getTotalKBytes, getUptime are satisfied
  * getServer, setServer(String URL), start

// Declare getValue operation String getVal
// Set parameter array
ModelMBeanOperationInfo[] apacheOperations
  new ModelMBeanOperationInfo[6];
MBeanParameterInfo[] getParms = new MBeanPa

MBeanParameterInfo[] getValueParms =
  new MBeanParameterInfo[] {
    (new MBeanParameterInfo("FieldName",
```

```
      "java.lang.String",
      "Apache status field name"))};

  Descriptor getValueDesc =
    new DescriptorSupport(
      new String[] {
        "name=getValue",
        "descriptorType=operation",
        "class=ApacheServer",
        "role=operation" });

  apacheOperations[0] =
    new ModelMBeanOperationInfo(
      "getValue",
      "getValue(): get an apache status field
      getValueParms,
      "java.lang.String",
      MBeanOperationInfo.INFO,
      getValueDesc);

  Descriptor getStateDesc =
    new DescriptorSupport(
      new String[] {
      "name=getState",
      "descriptorType=operation",
      "class=ApacheServer",
      "role=operation" });


  apacheOperations[1] =
    new ModelMBeanOperationInfo(
      "getState",
      "getState(): current status of apache s
```

```
    getParms,
    "java.lang.String",
    MBeanOperationInfo.INFO,
    getStateDesc);

Descriptor getServerDesc =
  new DescriptorSupport(new String[] {"name
     "descriptorType=operation",
     "class=ApacheServer",
     "role=operation" });

apacheOperations[2] =
  new ModelMBeanOperationInfo("getServer",
    "getServer(): URL of apache server",
    getParms,
    "java.lang.Integer",
    MBeanOperationInfo.INFO,
    getServerDesc);

MBeanParameterInfo[] setParms =
  new MBeanParameterInfo[] {
    (new MBeanParameterInfo("url",
    "java.lang.String",
    "Apache Server URL"))};

Descriptor setServerDesc =
  new DescriptorSupport(new String[] {"name
     "descriptorType=operation",
     "class=ApacheServer",
     "role=operation" });

apacheOperations[3] =
  new ModelMBeanOperationInfo("setServer",
```

```
    "getServer(): URL of apache server",
    setParms,
    "java.lang.String",
    MBeanOperationInfo.ACTION,
    setServerDesc);

MBeanParameterInfo[] startParms = new MBean

Descriptor startDesc =
  new DescriptorSupport(new String[] {"name
    "descriptorType=operation",
    "class=ApacheServer",
    "role=operation" });

apacheOperations[4] = new ModelMBeanOperati
  "start(): start apache server",
  startParms,
  "java.lang.Integer",
  MBeanOperationInfo.ACTION,
  startDesc);

MBeanParameterInfo[] stopParms = new MBeanP
Descriptor stopDesc = new DescriptorSupport
   "descriptorType=operation",
   "class=ApacheServer",
   "role=operation" });

apacheOperations[5] = new ModelMBeanOperati
  "stop(): start apache server",
  stopParms,
  "java.lang.Integer",
  MBeanOperationInfo.ACTION,
  stopDesc);
```

```
/* getters/setters for operations */
//    MBeanParameterInfo[] getParms = new M

Descriptor bytespsDesc =
  new DescriptorSupport(new String[] {"name
  "descriptorType=operation",
  "class=ApacheServer",
  "role=operation" });

apacheOperations[6] =
  new ModelMBeanOperationInfo("getBytesPerS
    "number of bytes per second processed",
    getParms,
    "int",
    MBeanOperationInfo.ACTION,
    bytespsDesc);

// Define notifications in ModelMBeanNotifi
// declare an "Apache Server Down" notifica
ModelMBeanNotificationInfo[] apacheNotifica
  new ModelMBeanNotificationInfo[1];

Descriptor apacheDownEventDesc =
  new DescriptorSupport(
    new String[] {
      "descriptorType=notification",
      "name=jmx.ModelMBean.General.Apache.D
      "severity=1",
      "MessageId=Apache001" });

apacheNotifications[0] =
  new ModelMBeanNotificationInfo(
```

```
            new String[] {"jmx.ModelMBean.General.A
            "jmx.ModelMBean.General",
            "Apache Server Down",
            apacheDownEventDesc);

        // Create the ModelMBeanInfo
        ModelMBeanInfo apacheMMBeanInfo =
          new ModelMBeanInfoSupport(
            "Apache",
            "ModelMBean for managing an Apache Web
            apacheAttributes,
            apacheConstructors,
            apacheOperations,
            apacheNotifications);

        // Set the MBean's Descriptor for the Model
        try {
          apacheMMBeanInfo.setMBeanDescriptor(apach
        } catch (Exception e) {
          System.out.println("CreateMBeanInfo faile
        }
        return apacheMMBeanInfo;
      }
```

Finally, just to complete the example, even though this is basically what's in Chapter 3 as the `Apache` class, here is the `ApacheServer` class for getting the data for the Apache server resource we are managing:

Code Block 3

```
package jnjmx.ch4;

/**
```

```
 * @author Heather Kreger
 * Chapter 4:  Java and JMX: Building Manageabl
 * ApacheServer: interacts with an Apache serve
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.util.HashMap;
import java.util.StringTokenizer;
import javax.management.*;

public class ApacheServer {
  // String constants for each of the Apache-re
  private static final String BUSY_WORKERS = "B
  private static final String BYTES_PER_REQ = "
  private static final String BYTES_PER_SEC = "
  private static final String CPU_LOAD = "CPULo
  private static final String IDLE_WORKERS = "I
  private static final String REQ_PER_SEC = "Re
  private static final String SCOREBOARD = "Sco
  private static final String TOTAL_ACCESSES =
  private static final String TOTAL_KBYTES = "T
  private static final String UPTIME = "Uptime"

  private static final int MAX_FAILURES = 3;

  // state constants for the Apache Server (bro
  public String DOWN = "DOWN";
  public String NOT_RESPONDING = "NOT_RESPONDIN
```

```
public String RUNNING = "RUNNING";
public String UNKNOWN = "UNKNOWN";

private int failures = 0;
private String state = UNKNOWN;
private long tzero;
private URL url;

// constructor that accepts the URL for the A
public ApacheServer(String url)
  throws MalformedURLException, IllegalArgume
  setServer(url);
  // test the ability to communicate using th
  try {
    getValue(UPTIME);
  } catch (Exception e) {
    System.out.println(
      "Apache server at " + url + " does not
  }
}

// Return the URL of the Apache server being
// Server is a readable and writable attribut
// it has a getter and setter
public String getServer() {
  return this.url.toString();
}

// Set the URL of the Apache server being man
public void setServer(String url)
  throws MalformedURLException, IllegalArgume
  this.url = new URL(url);
  // test to be sure the URL is really an Apa
```

```
  if (isStatusUrl(this.url) == false) {
    throw new IllegalArgumentException(url.to
  }
  this.state = ApacheMBean.UNKNOWN;
}

// return the current state of the Apache ser
public String getState() {
  return this.state;
}

// Example operation for start
// If this were running in the same JVM, this
// where the apache process initialzation wou
public String start() {
  this.state = RUNNING;
  return this.state;
}

// Sample operation for stop
// If this were running in the same JVM, this
// where the Apache process initialization wo
public String stop() {
  this.state = DOWN;
  return this.state;
}

// Validate that the URL is an Apache server
private boolean isStatusUrl(URL url) {
  return url.toString().endsWith("server-stat
}

// Get methods for metrics retrieved via the
```

```
// These are read-only attributes because no
// defined for them

public int getBusyWorkers() throws ApacheMBea
  return getIntValue(BUSY_WORKERS);
}

public int getBytesPerSec() throws ApacheMBea
  return getIntValue(BYTES_PER_SEC);
}

public float getBytesPerReq() throws ApacheMB
  return getFloatValue(BYTES_PER_REQ);
}

public float getCpuLoad() throws ApacheMBeanE
  return getFloatValue(CPU_LOAD);
}

public int getIdleWorkers() throws ApacheMBea
  return getIntValue(IDLE_WORKERS);
}

public float getReqPerSec() throws ApacheMBea
  return getFloatValue(REQ_PER_SEC);
}

public String getScoreboard() throws ApacheMB
  return getStringValue(SCOREBOARD);
}

public int getTotalAccesses() throws ApacheMB
  return getIntValue(TOTAL_ACCESSES);
```

```
}

public long getTotalKBytes() throws ApacheMBe
  return getLongValue(TOTAL_KBYTES);
}

public long getUptime() throws ApacheMBeanExc
  return getLongValue(UPTIME);
}

// These methods are used as internal utiliti
// There is one for each type returning the b

private String getValue(String value)
  throws ApacheMBeanException, NoSuchFieldExc
  String result;
  URLConnection k = establishConnection();

  result = (String) readStatus(k, value);

  if (result == null) {
    throw new NoSuchFieldException(value);
  }
  return result;
}

private float getFloatValue(String value) thr
  float result;
  try {
    result = Float.parseFloat((String) getVal
  } catch (IOException x) {
    throw new ApacheMBeanException(x);
  } catch (NoSuchFieldException x) {
```

```
    throw new ApacheMBeanException(x);
  }
  return result;
}

private int getIntValue(String value) throws
  int result;
  try {
    result = Integer.parseInt((String) getVal
  } catch (IOException x) {
    throw new ApacheMBeanException(x);
  } catch (NoSuchFieldException x) {
    throw new ApacheMBeanException(x);
  }
  return result;
}

private long getLongValue(String value) throw
  long result;
  try {
    result = Long.parseLong((String) getValue
  } catch (IOException x) {
    throw new ApacheMBeanException(x);
  } catch (NoSuchFieldException x) {
    throw new ApacheMBeanException(x);
  }
  return result;
}

private String getStringValue(String value) t
  String result;
  try {
```

```
    result = (String) getValue(value);
  } catch (IOException x) {
    throw new ApacheMBeanException(x);
  } catch (NoSuchFieldException x) {
    throw new ApacheMBeanException(x);
  }
  return result;
}

// Establishes the connection to the URL
// If the connection fails, then the state
// is changed to DOWN or NOT_RESPONDING

private URLConnection establishConnection()
  throws IOException, ApacheMBeanException {
  URLConnection k = this.url.openConnection()
  try {
    k.connect();
    this.failures = 0;
    this.state = ApacheMBean.RUNNING;
  } catch (IOException x) {
    if (++this.failures > MAX_FAILURES) {
      this.state = DOWN;
    } else {
      this.state = NOT_RESPONDING;
    }
    throw new ApacheMBeanException("state: "
  }
  return k;
}

// Parses the data returned from the URL
```

```
  private String readStatus(URLConnection k, St
    throws IOException {
    BufferedReader r =
      new BufferedReader(new InputStreamReader(
    for (String l = r.readLine(); l != null; l
      StringTokenizer st = new StringTokenizer(
      if (st.nextToken().trim().equals(value))
        // if it's the right value
        return (String) (st.nextToken().trim())
      } else {
        st.nextToken(); // past unwanted value
      }
    }
    return "";
  }

}
```

Now, let's look at the capabilities of model MBeans in detail.

4.2 The ModelMBean Interface

The model MBean implementation, including `RequiredModelMBean`, must implement the `ModelMBean` interface, which extends three other JMX interfaces: `DynamicMBean`, `PersistentMBean`, and `ModelMBeanNotificationBroadcaster`. Here's how these interfaces are used in the model MBean.

4.2.1 DynamicMBean

Because it is a dynamic MBean, the model MBean must implement the `DynamicMBean` interface, which consists of the `getAttribute()`, `setAttribute()`, `invoke()`, and `getMBeanInfo()` methods. The `getMBeanInfo()` method returns a `ModelMBeanInfo` instance that extends `MBeanInfo`. The model MBean uses the additional information in the descriptors in `ModelMBeanInfo` to delegate and map these operations to the correct operations on your managed resources. You do not need to implement these methods unless you need to override the default behavior. The `DynamicMBean` and basic `MBeanInfo` interfaces were explained fully in Chapter 3.

4.2.2 PersistentMBean

The model MBean is also responsible for its own persistence and therefore must implement the `PersistentMBean` interface. The `PersistentMBean` interface is simply the `load()` and the `store()` methods. This does not mean that a model MBean *must* persist itself. Some implementations of the JMX agent may be completely transient in nature. In this situation, the `load()` and `store()` methods would be empty. The `load()` method is invoked when the model MBean is instantiated. The

`store()` method is invoked according to the policy defined in the descriptors in the `ModelMBeanInfo` of the MBean (to be discussed later in this chapter).

In a simple implementation, the model MBean may be saved to a file. Alternatively, it may be saved to a database through JDBC (a.k.a., the Java Database Connectivity API). In a more complex environment, the JMX agent may handle persistence on behalf of the model MBean. If you are using an implementation of JMX that has an implementation of `RequiredModelMBean` that does not support persistence and you need support for persistence, then you must override and implement the `load()` and `store()` methods. Here is the `PersistentMBean` interface:

```
interface PersistentMBean {
        void load();
        void store();
}
```

The model MBean constructor will attempt to prime itself by calling the model MBean `load()` method. The model MBean `load()` method must determine if this model MBean has a persistent representation. It can do this by invoking the `findInPersistent()` method; this is not a JMX standard, but it is a common practice. Then the `load()` method must determine where this data is located, retrieve it, and initialize the model MBean. The model MBean can, through JDBC operations, persist data to and populate the model MBeans from any number of data storage options, such as an LDAP[8] server, a DB2[9] application, a flat file, an NFS[10] file, or an internal high-performance cache. If the model MBean were implemented as an EJB object, object loading might be managed by the EJB container and the `load()` method would do nothing.

Because the `load()` method is not typically used by the

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

4.2.3 ModelMBeanNotificationBroadcaster

The `ModelMBeanNotificationBroadcaster` interface extends the `NotificationBroadcaster` interface and adds support for issuing generic text notifications and attribute change notifications. These events are sent only if at least one `NotificationListener` instance is registered with the model MBean. Generic text notifications are simple to use, accepting any text string and sending it as a notification. Attribute change notifications are sent by the model MBean whenever one of its attribute's values changes. The MBean can also send an attribute change notification to the managed resource itself. For example, if a configuration program has just changed an attribute in the MBean and a setter does not exist for that attribute, the MBean could send an attribute change notification to the managed resource, given that the managed resource implements the `NotificationListener` interface. The managed resource would detect the change in its external configuration (held by the MBean), and it would make the internal adjustments necessary to comply. Application-specific notifications are supported like they are for any other MBean. Here is the `ModelMBeanNotificationBroadcaster` interface:

```
    interface ModelMBeanNotificationBroadcaster
    NotificationBroadcaster {
    addAttributeChangeNotificationListener(
              javax.management.NotificationListen
              java.lang.String inAttributeName,
              java.lang.Object inhandback);
    removeAttributeChangeNotificationListener(
              javax.management.NotificationListen
              java.lang.String inAttributeName);
    sendAttributeChangeNotification(
              javax.management.Attribute inOldVal
              javax.management.Attribute inNewVal
    sendAttributeChangeNotification(
               javax.management.AttributeChangeNo
    sendNotification(javax.management.Notificati
    sendNotification(java.lang.String ntfyText);
}
```

The `ModelMBean` interface has two additional methods:

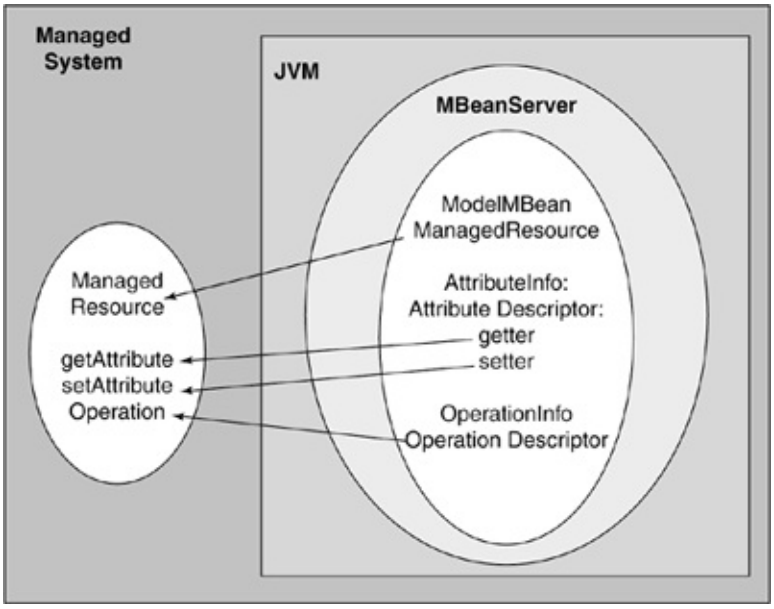
```
void setManagedResource(java.lang.Object manage
              java.lang.String managedResourceRef
void setModelMBeanInfo(ModelMBeanInfo ModelMBea
```

Let's look at how these methods are used to customize your model MBean.

4.3 Managed Resources

As illustrated in Figure 4.1, when attributes are retrieved or operations are invoked on a model MBean, it will delegate those requests to a managed resource. The managed resource is a runtime object associated with a model MBean via the `setManagedResource()` method. The `setManagedResource()` method sets the default managed resource for the entire model MBean. A different managed resource can be set per operation and attribute as well. The model MBean delegates operations to the managed resource associated with it.

Figure 4.1. JMX Model MBeans and a Managed Resource



Because attribute setter and getter methods are represented as operations in the model MBean, attribute values may be retrieved from or set to different managed resources as well. This capability eliminates the need to write a management facade or delegation class if your application's management characteristics are distributed across several classes or

instances of the same class. The model MBean can be customized to perform as that facade.

If you want to stay simple and set one managed-resource object for your entire model MBean, you can use the `ModelMBean` interface's `setManagedResource()` method:

The first parameter is the reference of your managed resource class. The second parameter indicates the type of reference you are passing. It can be set to one of the following strings:"ObjectReference", "Handle", "IOR", "EJBHandle", "RMIReference". Although "ObjectReference" is always supported, an MBeanServer does not have to support all of these types. You should read the documentation for your JMX implementation to determine which of these types, or any additional types, are valid for your implementation.

```
void setManagedResource(
        java.lang.Object managedResourceReferen
        java.lang.String managedResourceReferen
```

Think back to the earlier ApacheServer example. The statement

```
apacheMMBean.setManagedResource(myApacheServer,
```

causes the `ApacheServer` instance, `myApacheServer`, to be the managed resource for the model MBean. This managed resource, `myApacheServer`, will now be the default managed resource for executing all operations in this model MBean—that is, `start()`, `stop()`, all get methods (including `getServer()`), and `setServer()`. You can override this default on a particular attribute or operation by using the descriptors in `ModelMBeanInfo` for that particular operation. We will show how to do this in the next section.

4.4 ModelMBeanInfo

The model MBean uses a `ModelMBeanInfo` class for metadata. `ModelMBeanInfo` extends the `MBeanInfo` class by adding a descriptor to each of the elements of the management interface: attributes, operations, and notifications. Descriptors are a list of name/value pairs that describe additional metadata about the resource and behavioral policy for the model MBean instance. The descriptors control how attribute values are cached or persisted, which attributes are retrievable from other objects, whether notifications are logged, where logs are located, along with other descriptive data. This is the `ModelMBeanInfo` interface:

```
interface ModelMBeanInfo {
  Object clone():
  Descriptor getMBeanDescriptor();
  void setMBeanDescriptor(Descriptor inDescript
  Descriptor getDescriptor(String inDescriptorN
                           String inDescriptorT
  Descriptor[] getDescriptors (String inDescrip
  void setDescriptor(Descriptor inDescriptor,
                     String inDescriptorType );
  void setDescriptors(Descriptor[] inDescriptor

  ModelMBeanAttributeInfo getAttribute(String i
  ModelMBeanNotificationInfo getNotification(St
  ModelMBeanOperationInfo getOperation(String i
  MBeanAttributeInfo[]getAttributes();
  MBeanNotificationInfo[]getNotifications() ;
  MBeanOperationInfo[]getOperations();
  MBeanConstructorInfo[]getConstructors() ;
  String getClassName();
```

```
  String getDescription();
}
```

In addition, `ModelMBeanInfo` is required to support the following constructors:

- `ModelMBeanInfo();`

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

- `ModelMBeanInfo (ModelMBeanInfo);`

  The copy constructor constructs a `ModelMBeanInfo` instance that is a duplicate of the one passed in.

- `ModelMBeanInfo (className, String description,`

  `ModelMBeanAttributeInfo[], ModelMBeanConstru`
  `ModelMBeanOperationInfo[], ModelMBeanNotific`

  The `MBeanInfo`-compliant constructor creates a `ModelMBeanInfo` instance with the provided information, but the MBean descriptor contains default value. Because the MBean descriptor must not be `null`, the default descriptor will contain at least the `name` and `descriptorType` fields. The name will match the MBean name.

- `ModelMBeanInfo (className, String description,`

  `ModelMBeanAttributeInfo[], ModelMBeanConstru`
  `ModelMBeanOperationInfo[], ModelMBeanNotific`
  `MBeanDescriptor)`

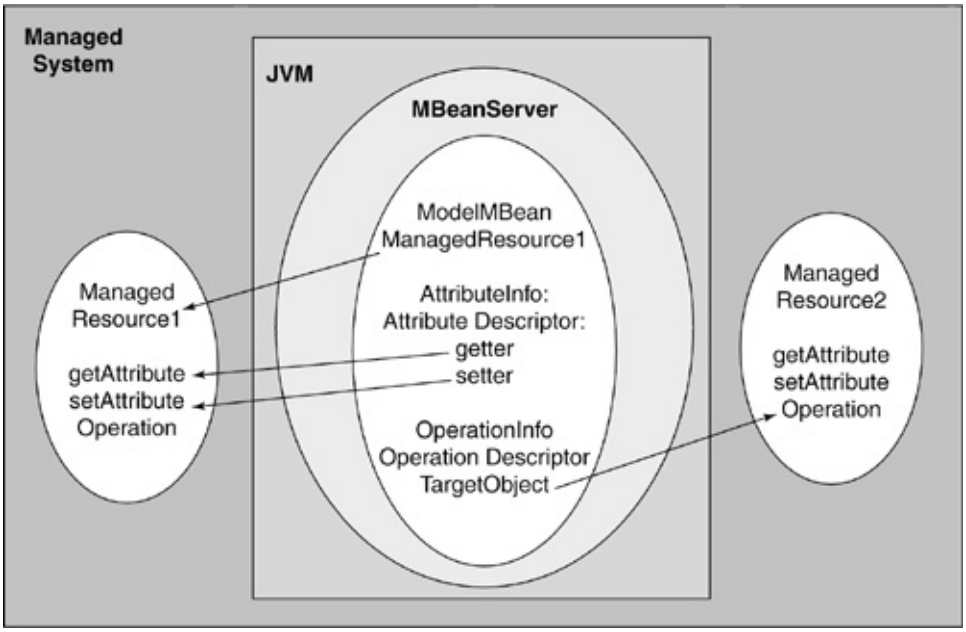
The full constructor creates a `ModelMBeanInfo` instance with the provided information. The MBean descriptor is verified: If it is not valid, an exception will be thrown and a default MBean descriptor will be set. You can see that there is a descriptor associated with the model MBean itself. It must not be `null`.

The normal `MBeanInfo` methods return model MBean extensions of the `AttributeInfo`, `OperationInfo`, and `NotificationInfo` classes respectively. Each of the model MBean extensions implements the `DescriptorAccess` interface. The `DescriptorAccess` interface gives you access to `attribute descriptor`, `operation descriptor`, and `notification descriptor`, which are associated with `AttributeInfo`, `OperationInfo`, and `NotificationInfo`, respectively. This is the `DescriptorAccess` interface:

```
interface DescriptorAccess {
  Descriptor getDescriptor();
  void setDescriptor(Descriptor inDescriptor);
}
```

The descriptor also defines the method signatures that will be executed by the model MBean on its managed resource to satisfy the `getAttribute()`, `setAttribute()`, and `invoke()` operations. As Figure 4.2 shows, defining the managed resource and method signature in the operation descriptor allows the actual methods called for an operation to have different signatures and to be delegated to a wide range of objects at runtime. This flexibility enables the management of distributed, dynamic applications directly from a model MBean.

Figure 4.2. JMX Model MBeans and Multiple Managed Resources



The descriptor-oriented methods in `ModelMBeanInfo` allow you to list and update any descriptor in the model MBean without having to retrieve the specific `ModelMBeanAttributeInfo`, `ModelMBeanOperationInfo`, `ModelMBeanConstructorInfo`, or `ModelMBeanNotificationInfo` class for it first. This is strictly a convenience for the programmer.

You can customize the model MBean during development by defining the `ModelMBeanInfo` instance in the program with the `ModelMBeanInfo` interface. Defining an instance of `MBeanInfo` directly in a program creates a static model MBean. You can also create model MBeans more dynamically. The descriptor values and `ModelMBeanInfo` attributes and operations can be loaded from the application or a file at runtime. This means that you can update or upgrade the management data along with your application without having to remove and reinstantiate all of the data. You could also use an Interface Definition Language (IDL)[11] or Managed Object Format (MOF)[12] file. Using external files whose formats are defined by standards permits management interfaces to be defined in a language-independent manner. The JMX

specification does not define how to use XML, IDL, or MOF files for `MBeanInfo` data, but we include an example at the end of this chapter to illustrate the use of an XML service to load `ModelMBeanInfo` from an XML file.

4.5 Descriptors

Model MBeans use the descriptors to support additional metadata and policy for caching, persistence, and logging for the management interface defined in `ModelMBeanInfo`. Simply put, a descriptor is a set of unordered `keyword= value` pairs that can be accessed with the `Descriptor` interface.

Copies of descriptors are retrieved from the management interface element with the `DescriptorAccess` interface. A different descriptor type is defined for and implemented by `ModelMBeanOperationInfo`, `ModelMBeanConstructor-Info`, `ModelMBeanAttributeInfo`, and `ModelMBeanNotificationInfo`. You can also retrieve and set descriptors from methods on `ModelMBeanInfo`. You can get and set descriptors by name or descriptor type (`mbean`, `attribute`, `operation`, `notification`):

```
interface Descriptor {
  java.lang.Object clone();
  java.lang.String[] getFieldNames();

  java.lang.String[] getFields();
  java.lang.Object getFieldValue(java.lang.String fieldName);
  java.lang.Object[] getFieldValues(java.lang.String[] fieldNames);
  boolean isValid();
  void removeField(java.lang.String fieldName);
  void setField(java.lang.String fieldName,
                java.lang.Object fieldValue);
  void setFields(java.lang.String[] fieldNames,
     java.lang.Object[] fieldValues);
}
```

The `Descriptor` interface allows a variety of accesses. You can retrieve one descriptor field, a specific set of descriptor fields, or all descriptor fields. The `Descriptor` interface also provides a method that can be used to validate that the current value of the descriptor is valid for its descriptor type.

A set of descriptor keywords are defined by the JMX specification to ensure standard and uniform functionality and treatment. We will cover these in detail later in this section.

Note that you can set values for descriptors as well. You can replace an entire field value. In the `setField()` method the `fieldValue` parameter is copied into the descriptor. New keywords can be added by applications or adapters at any time (see Section 4.8.3).

The `Descriptor` interface is implemented by the `DescriptorSupport` class. You can use this class to create descriptors in a variety of ways. Choose the one that's easiest for you because there is no difference in the net effect:

- Instantiate with an array of `name=value` strings:

  ```
  // Instantiate Descriptors using the array
  ```

```
Descriptor timeStateDescA =
     new DescriptorSupport(new String[] {
          "name=Time",
          "descriptorType=attribute",
          "displayName=currentTime",
          "getMethod=getTime",
          "setMethod=setTime",
          "currencyTimeLimit=20" });
```

This style does not support setting object values for the descriptor.

- Make a series of `setField()` method calls:

```
// Instantiate Descriptors using the field setters
Descriptor timeStateDescF = new DescriptorSupport();
timeStateDescF.setField("name","Time");
timeStateDescF.setField("descriptorType","attribute");
timeStateDescF.setField("displayName","currentTime");
timeStateDescF.setField("getMethod","getTime");
timeStateDescF.setField("setMethod","setTime");
timeStateDescF.setField("currencyTimeLimit","20");
```

This style is somewhat less efficient in processing, but it is easier to adjust during development because you can just comment out the statements for the fields you no longer want to set. This is the style you should use if the type for the field is `Object` and you need to save an object value as the descriptor value. Here is how to set a value for an object attribute that reflects the start date for an application:

```
Date currDate = new Date();
timeStateDescF.setField("value", currDate);
```

- Call the `setField()` method with an array of names and values:

```
Descriptor timeStateDescFA = new DescriptorSupport();
String[] fieldnames = new String[] {"name",
        "descriptorType",
        "displayName",
        "getMethod",
        "setMethod",
        "currencyTimeLimit"};
Object[] fieldvalues = new Object[] {
        new String("Time"),
        new String("attribute"),
        new String("currentTime"),
        new String("getTime"),
        new String("setTime"),
        new String("20") };
timeStateDescFA.setFields(fieldnames,fieldvalues);
```

This style does support object values.

The `Descriptor` interface also supports a `clone()` operation that is very convenient for creating new descriptors from existing ones as templates. The `remove()` and `setField()` methods make it very simple to selectively modify the descriptor.

Sections 4.5.1 through 4.5.5 cover the standard descriptor keywords supported by all model MBeans. These descriptor keywords are defined in the JMX specification[13] as well. Section 4.6 then discusses caching and persistence behavior in the model MBean.

4.5.1 Model MBean Descriptors

Descriptors at the MBean level define default policies for the attributes, operations, and notifications for persistence, caching, and logging. Table 4.1 shows the descriptors that are valid for model MBeans.

Table 4.1. Valid Model MBean Descriptors

Descriptor	Meaning
`name`, `displayName`	The logical and displayable names of the model MBean.
`descriptorType`	Must be "mbean" to signal that this descriptor applies to the model MBean.
`version`	The version of this model MBean.
`export`	If this is set to a value, then the model MBean is exported to a registry (like JNDI) or somehow made visible to entities outside the MBeanServer. The value is used to assist with the export. If the value is `null`, then an export is not performed. See Section 4.6.6.3 (Export).
`persistPolicy`, `persistPeriod`, `persistLocation`, `persistName`	See Section 4.6.4 (Persistence).
`currencyTimeLimit`, `lastReturnedValue`, `lastUpdatedTimeStamp`	See Section 4.6.1 (Caching).
`log`, `logFile`	See Section 4.6.5 (Logging).

`visibility`	See Section 4.6.6.2 (Visibility).
`presentationString`	See Section 4.6.6.1 (Presentation String).

Looking back at the Apache server example again, we can see an example of the model MBean descriptor. Here is another example for an Apache server that also defines a persistence policy:

```
//   Set MBean descriptor example
  Descriptor mmbDescription = new DescriptorSupport(new String[] {
    "name=apacheServerManager",         // server MBean name
    "descriptorType=mbean",             // MBean descriptor
    "displayName=apache server MBean", // screen name of MBean
    "log=T",                            // log all notifications
    "logFile=jmxmain.log",              // log in jmxmain.log
    "currencyTimeLimit=10",             // cache all attribute
                                        // values for 10 seconds
    "persistPolicy=noMoreOftenThan",    // persist on change
    "persistPeriod=10",                 // no more often than
                                        // 10 seconds
    "persistLocation=jmxRepository",    // save in jmxRepository
                                        // directory
    "persistName=serverMBean"});        // save in file called
                                        // serverMBean
```

The `descriptorType` field is set to "mbean" to indicate that this is an MBean descriptor. The `displayName` field is set to "serverMBean". The line `log=T` indicates that all notifications will be logged into the file named in the `logFile` field: `jmxmain.log`. The line `currencyTimeLimit=10` means that all attributes in this model MBean will be cached for up to 10 seconds. After a value is 10 seconds old, it will be retrieved from the managed resource again. The `persistPolicy` field is set to `noMoreOftenThan` with a `persistPeriod` value of `10`. This means that the attribute values will be saved to a directory called `jmxRepository` in a file called `serverBean` whenever they are updated, but no more often than every 10 seconds. This restriction prevents high rates of updates that are not meaningful.

4.5.2 Attribute Descriptors

The model MBean attribute descriptor includes policy and configuration for managing its persistence, caching, protocol mapping, and handling of get and set requests. When the model MBean is created by the managed resource, the managed resource defines the operations that will be executed by the model MBean that will satisfy the get and set of the attribute. By defining operations, the actual methods called to satisfy `getAttribute()` and `setAttribute()` requests are allowed to vary and to be delegated to a wide range of objects at runtime. This flexibility enables management of distributed, dynamic applications If an attribute has no operation associated with it, the values are maintained in the value descriptor field in the model MBean.

Let's look at the implications of this a little more closely. If an attribute 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 cached in the model MBean and any `AttributeChangeNotification` listeners are sent an attribute change notification —that is, an `AttributeChangeNotification` instance. For `getAttribute()`, the currently cached value for the attribute in the model MBean itself is simply returned. In this case there is no delegation to a managed resource. This can be useful for static information and helps minimize the interruption of managed resources to retrieve static resource information.

When attribute value caching is supported, if the data requested is current, then the model MBean returns its cached value and the managed resource is not interrupted with a data retrieval request. Because direct interaction with the managed resource is not required for each interaction with the management system, the impact of management activity on runtime application resources and performance is minimized.

Table 4.2 shows the descriptors that are supported in the `ModelMBeanAttributeInfo` descriptor.

This attribute descriptor example is based on the `State` attribute of the Apache model MBean example and illustrates most of the fields:

```
// Set attribute descriptor examples
ModelMBeanAttributeInfo[] aAttributes = new
  ModelMBeanAttributeInfo[3];
// State
Descriptor stateDesc = new DescriptorSupport();

stateDesc.setField("name","State");
stateDesc.setField("descriptorType","attribute");
stateDesc.setField("displayName","Apache Server State");
stateDesc.setField("getMethod","getState");
stateDesc.setField("setMethod","setState");
stateDesc.setField("currencyTimeLimit","20");
aAttributes[0] = new ModelMBeanAttributeInfo("State",
        "java.lang.String",
        "State: state string.",
        true,
        true,
        false,
        stateDesc);
```

Table 4.2. `ModelMBeanAttributeInfo` **Descriptors**

Descriptor	Meaning
`name`, `displayName`	The logical and displayable names of the model MBean attribute. The displayable name should be human readable and may be internationalized. If `displayName` is not set, the

	`name` descriptor field will be used.
`descriptorType`	Must be set to `attribute`.
`value`, `default`, `legalValues`	See Section 4.6.2 (Values and Validation).
`protocolMap`	See Section 4.6.6.4 (Protocol Map).
`getMethod`, `setMethod`	See Section 4.6.3 (Delegation).
`persistPolicy`, `persistPeriod`	See Section 4.6.4 (Persistence).
`currencyTimeLimit`, `lastUpdatedTimeStamp`	See Section 4.6.1 (Caching).
`iterable`	Defines if this attribute must be iterated over. The default is `false`.
`visibility`	See Section 4.6.6.2 (Visibility).
`presentationString`	See Section 4.6.6.1 (Presentation String).

In this example the name of the attribute is `State`, but the displayed name will be "Apache Server State". The `getMethod` descriptor field defines `getState()` as the get method. The `setMethod` descriptor field defines `setState()` as the set method. The value `20` in the `currencyTimeLimit` field means that this attribute value will be cached for 20 seconds.

In the following `ModelMBeanAttributeInfo` instance for total accesses to the Apache server, we have added an example for the use of `default` and `protocolMap` fields:

```
// TotalAccesses
Descriptor totalAccessesDesc = new DescriptorSupport();
    totalAccessesDesc.setField("name","TotalAccesses");
    totalAccessesDesc.setField("descriptorType", "attribute");

    totalAccessesDesc.setField("default", "0");
```

```
    totalAccessesDesc.setField("displayName",
                               "Total Accesses on Server");
    totalAccessesDesc.setField("getMethod","getTotalAccesses");
    totalAccessesDesc.setField("setMethod","setTotalAccesses");
    Descriptor TotalAccessesMap =
       new DescriptorSupport(new String[] {
         "SNMP=1.3.6.9.12.15.18.21.0",
         "CIM=ManagedResource.Version"});
    totalAccessesDesc.setField("protocolMap",(TotalAccessesMap));

aAttributes[1] = new ModelMBeanAttributeInfo("TotalAccesses",
    "java.lang.Integer",
    "TotalAccesses: number of times the State string",
    true,
    false,
    false,
    totalAccessesDesc);
```

A value of `0` in the `default` field means that the default value for `TotalAccesses`, if it is not set, should be `0`. The protocol map contains two hints to the protocol adapters. One is for an SNMP adapter indicating that when the MIB object ID (OID) 1.3.6.9.12.15.18.21.0 is requested, the value in `TotalAccesses` should be returned. The other is for a CIM adapter indicating that this same attribute and value is equivalent to the `Version` attribute in the CIM class `ManagedResource`.

In the following example we are defining a static attribute, the maximum thread pool to be used in each JVM for an Apache server. The `value` field of the descriptor is set to `99`. The `currencyTimeLimit` descriptor field is set to `–1`, which means that the value never becomes stale. Notice that no get or set methods have been defined:

```
// MaxPool static value
Descriptor maxPoolHardValueDesc = new DescriptorSupport();
maxPoolHardValueDesc.setField("name","maxPool");
maxPoolHardValueDesc.setField("descriptorType","attribute");
maxPoolHardValueDesc.setField("value", new Integer("99"));
maxPoolHardValueDesc.setField("displayName",
  "maximum Thread Pool for Apache Server, a Hard Coded Value");
maxPoolHardValueDesc.setField("currencyTimeLimit","-1");

aAttributes[2] = new ModelMBeanAttributeInfo("maxPoolHardValue",
  "java.lang.Integer",
  "maxPool: static maximum thread pool value for the Apache server",
  true,
  false,
  false,
  maxPoolHardValueDesc);
```

Just for the sake of completeness, if `we needed to define a static value that`

contained a `Date` object, in this case the time that the Apache server was started, the setting of the descriptor would look like this:

```
// StartDate as a static value
  Descriptor startDateDesc = new DescriptorSupport();
  startDateDesc.setField("name","startDate");
  startDateDesc.setField("descriptorType","attribute");
  startDateDesc.setField("displayName","ApacheServerStartTime");
  startDateDesc.setField("value", new Date());
  startDateDesc.setField("currencyTimeLimit","-1");
```

And the setting of the `ModelMBeanAttributeInfo` object would look like this:

```
// Set the attribute info
  aAttributes[3] =
     new ModelMBeanAttributeInfo("startDateHardValue",
     "java.util.Date",
     "startDate of ApacheServerHardValue: static Date value",
     true,
     false,
     false,
     startDateDesc);
```

Here is how you would get these attributes using the model MBean `myMMBean`:

```
// Retrieving these attributes using the model MBean myMMBean:
RequiredModelMBean myMMBean = new RequiredModelMBean();
// Set ModelMBeanInfo and managedResource here
String myState = (String) myMMBean.getAttribute("State");
String myAccesses = (String) myMMBean.getAttribute("TotalAccesses");
Integer myPool = (Integer) myMMBean.getAttribute("maxPool");
```

4.5.3 Constructor Descriptors

Every MBean must have a public constructor. It is also recommended that a default constructor—that is, a constructor that accepts no parameters—be provided. If you define constructors that require parameters, then you must define the constructor with an `MBeanConstructorInfo` instance. The `ModelMBeanInfo` descriptor for the constructor defines the constructor to be an operation with a method role of constructor, as opposed to operation, getter, or setter, examples of which we will see later. Constructor signatures are defined just like operation signatures (which are discussed in the next section):

```
  // Constructor descriptor example

ModelMBeanConstructorInfo[] aConstructors =
  new ModelMBeanConstructorInfo[1];
Class apacheServerClass = null;
try {
  apacheServerClass = Class.forName("jnjmx.ch4.ApacheServer");
```

```
} catch(Exception e) {
  System.out.println("Apache Server Class not found");
}

Constructor[] constructors = apacheServerClass.getConstructors();

Descriptor apacheServerDesc = new DescriptorSupport();
apacheServerDesc.setField("name","ApacheServer");
apacheServerDesc.setField("descriptorType", "operation");
apacheServerDesc.setField("role","constructor");

aConstructors[0] = new ModelMBeanConstructorInfo(
    "ApacheServer(): Constructs an apache server query tool",
     constructors[0],
     apacheServerDesc);
```

In this example we see that a default constructor has been defined for the model MBean. Table 4.3 gives the details of all the descriptors standardized for `ModelMBeanConstructorInfo`.

4.5.4 Operation Descriptors

For operations, the method signature must be defined in `ModelMBeanOperationInfo`. For each operation defined, a target object must be identified on which to invoke the operations. This target object can be set for the model MBean as a whole (by the `setManagedResource()` method) or per operation in the `targetObject` descriptor field. You can even have different target objects for different operations. This flexibility allows distributed, component-based applications to be supported by the model MBean. It also supports management of applications that do not already have a management facade. Like attributes, the model MBean supports caching the last returned value of the operation. Caching can reduce the interruptions to the managed application. Table 4.4 lists the descriptors that are supported in `ModelMBeanOperationInfo`.

Table 4.3. `ModelMBeanConstructorInfo` Descriptors

Descriptor	Meaning
`name`, `displayName`	The logical and displayable names of the model MBean constructor. The displayable name should be human readable and may be internationalized. If `displayName` is not set, then the `name` descriptor field will be used.
`descriptorType`	Must be set to "operation".
`role`	Set to "constructor", "operation", "getter", or "setter", depending on how the operation is used.

`Class`	Class where method is defined (fully qualified).
`visibility`	See Section 4.6.6.2 (Visibility).
`presentationString`	See Section 4.6.6.1 (Presentation String).

Table 4.4. `ModelMBeanOperationInfo` **Descriptors**

Descriptor	Meaning
`name`, `displayName`	The logical and displayable names of the model MBean operation. The displayable name should be human readable and may be internationalized. If `displayName` is not set, then the `name` descriptor field will be used.
`DescriptorType`	Must be set to "operation".
`role`	Set to "constructor", "operation", "getter", or "setter", depending on how the operation is used.
`targetObject`, `targetType`	See Section 4.6.3 (Delegation).
`currencyTimeLimit`, `lastReturnedValue`, `lastUpdatedTimeStamp`	See Section 4.6.1 (Caching).
`Iterable`	Defines if this attribute must be iterated over. The default is `false`.
`visibility`	See Section 4.6.6.2 (Visibility).
`presentationString`	See Section 4.6.6.1 (Presentation String).

The following example illustrates a simple `Integer start()` operation. A value of `3` in the `currencyTimeLimit` descriptor field specifies that the value returned from the operation will be valid in the cache for three seconds. The value of

`jnjmx.ch4.ApacheServer for the class` field defines what the class for the target object will be. Then we can see where we are creating an `apacheServerInitiator` instance. This object is set as the target object for *just* the start operation because it must use a command line to start the Apache process. The `targetObjectType` descriptor field must be set to `objectReference`. The `stop()method` is still invoked on the default target object for the model MBean.

```
  // Operation descriptor example
ModelMBeanOperationInfo[] dOperations =
   new ModelMBeanOperationInfo[1];
MBeanParameterInfo[] params = null;

Descriptor stopDesc = new DescriptorSupport();
stopDesc.setField("name","stop");
stopDesc.setField("descriptorType","operation");
stopDesc.setField("class","jnjmx.ch4.ApacheServer");
stopDesc.setField("role","operation");

dOperations[0] = new ModelMBeanOperationInfo("stop",
     "stop(): stop the apache server",
     params ,
     "void",
     MBeanOperationInfo.ACTION,
     stopDesc);

Descriptor startDesc = new DescriptorSupport(new String[] {
    "name=start",
    "class=jnjmx.ch4.ApacheServerInitiator",
    "descriptorType=operation",
    "role=operation",
    "currencyTimeLimit=3"});

// Create an instance of ApacheServerInitiator just for the
// start operation
ApacheServerInitiator asf =
  new ApacheServerInitiator("http://www.apache.org");
startDesc.setField("targetObject",asf);
startDesc.setField("targetObjectType","objectReference");

aOperations[1] = new ModelMBeanOperationInfo("start",
        "start(): start the apache server",
        params,
        "java.lang.Integer",
        MBeanOperationInfo.INFO,
        startDesc);
```

Here is how you would invoke the start operation on the model MBean myMMBean where it has no parameters:

```
// invoking the operation
Integer startRC = (Integer) apacheMMBean.invoke("start", null,
    null);
```

4.5.5 Notification Descriptors

The model MBean notification descriptor defines a severity value, visibility value, and message ID for national language support, as well as whether the notification should be logged. A response operation may also be defined. Table 4.5 lists the descriptor keywords supported in the model MBean notification descriptor.

Table 4.5. `ModelMBeanNotificationInfo` **Descriptors**

Descriptor	Meaning
`name`	The logical and displayable name of the notification.
`descriptorType`	Must be "notification".
`severity`	The `severity` integer ranges from `0` to `6`. Here are the meanings of these numbers: 0: Unknown, indeterminate 1: Nonrecoverable 2: Critical, failure 3: Major, severe 4: Minor, marginal, or error 5: Warning 6: Normal, cleared, or informative

The `severity` descriptor field can be used with a `notificationListener` notification filter or to influence logging. The assignment of `severity` values is very subjective and should be done by someone with an understanding of the managed resource, the notification, and how the management systems might be using `severity` to guide their reaction to the notification.

`messageId`

log

logFile

visibility

presentationString

Here is a notification descriptor example from the Apache model MBean:

```
// Declare an "Apache Server Down" notifica
ModelMBeanNotificationInfo[] apacheNotifica
  new ModelMBeanNotificationInfo[1];

Descriptor apacheDownEventDesc =
  new DescriptorSupport(
    new String[] {
      "descriptorType=notification",
      "name=jmx.ModelMBean.General.Apache.D
      "severity=1",
      "MessageId=Apache001",
      "log=T",
      "logfile=jmx.apachelog" });

apacheNotifications[0] =
  new ModelMBeanNotificationInfo(
    new String[] {"jmx.ModelMBean.General.A
    "jmx.ModelMBean.General",
    "Apache Server is Down",
    apacheDownEventDesc);
```

In this case the name of the notification is `jmx.ModelMBean.General.Apache.Down`. The severity of the notification is set to `1`, and the message ID is "Apache001". This notification will always be logged in the file `jmx.apachelog`.

Here is how you would send this notification from the `myMMBean` model MBean:

```
apacheMMBean.sendNotification("Apache Server is
```

4.6 Behavior of the Model MBean

Model MBeans provide support for some common MBean requirements—caching, persistence, logging—and a host of miscellaneous other services. This section describes the processing that model MBeans perform in these services. Remember, because `RequiredModelMBean` is an actual concrete class (and you have the source for it), you can extend it and override its behavior.

4.6.1 Caching

Caching is supported for both attribute values and operation responses. In general, if the data requested is current, then the model MBean returns the cached value and does not retrieve the value from or invoke the operation on the managed resource. The value is cached in the `value` descriptor field of `ModelMBeanAttributeInfo` or the `lastReturnedValue` field of `ModelMBeanOperationInfo`. Caching policy is defined by two descriptor fields: `currencyTimeLimit` and `lastUpdatedTimeStamp`. This is the algorithm for caching; it is relatively straightforward:

```
if (currencyPeriod < 0)
{  /* if currencyTimeLimit is -1, then value is
  returnCachedValue = true;
  resetValue = false;
} else if (currencyPeriod == 0)
{  /* if currencyTimeLimit is 0, then value is
  returnCachedValue = false;
  resetValue = true;
} else
{  /* if now < currencyTimeLimit + LastUpdateTi
  String tStamp = (String)
```

```
      descr.getFieldValue("lastUpdatedTimeStamp
  if (tStamp == null)
  {
    tStamp = "0";
  }
  long lastTime = (new Long(tStamp)).longValue(
  long now = (new Date()).getTime();
  if (now < (lastTime + currencyPeriod))
  {
    returnCachedValue = true;
    resetValue = false;
  } else
  { /* value is expired */
    returnCachedValue = false;
    resetValue = true;
  }
}

if (resetValue == true)
  { /* value is not current, so remove it */
    descr.setField("lastUpdatedTimeStamp", "0")
    descr.setField("value", null);
    response = null;
  }

If (returnCachedValue == true) {
  response = descr.getFieldValue("value");

} else {
  response = getAttribute();
  descr.setField("lastUpdatedTimeStamp", now);
  descr.setField("value", response);
}
```

```
return response;
```

For attributes, if `getAttribute()` is called for an attribute with a stale value (or no value), then the `getMethod` defined in the attribute's descriptor will be invoked and the returned value will be recorded in `value` for the attribute, and the `lastUpdatedTimeStamp` field will set to the current time. The requester will be handed the new value. If no `getMethod` descriptor field is defined and the `value` field in the attribute descriptor is not set, then the value in the `default` descriptor field from the attribute's descriptor will be returned. Note that the cached value is set lazily; that is, it is not updated just because it is stale; someone must ask for it to trigger the update. Here is an example of caching descriptor fields for an attribute:

```
"getMethod=getMyOwnAttribute"
"currencyTimeLimit=30"

"lastUpdatedTimeStamp=03302002120905003"
"value=highspeed"
```

Similarly for operations, if an `invoke()` method is executed for an operation with a stale value (or no value) set in the `lastReturnedValue` field of the descriptor, then the `invoke()` method is executed for the operation and the returned value will be recorded in the `lastReturnedValue` field for the operation. Likewise, `lastUpdatedTimeStamp` will be set to the current time. The requester will be handed the new value. This cached value is also maintained lazily:

```
"name=getRateOfSpeed"
"currencyTimeLimit=30"
```

```
"lastUpdatedTimeStamp=0330200212090500 3"
"lastReturnedValue=30mph"
```

The `currencyTimeLimit` descriptor field may be in the model MBean's descriptor. When `currencyTimeLimit` is set here, it applies to all attributes and operations. If `currencyTimeLimit` is not defined, then the default is to do no caching. We can override the model MBean's default policy by defining it in the attribute or operations descriptor. As in the preceding examples, `currencyTimeLimit` is set to the number of seconds that a cached value is current. If `currencyTimeLimit` is set to `0`:

```
"currencyTimeLimit=0"
```

then no caching is performed and the value is retrieved or operation invoked for every request.

If `currencyTimeLimit` is set to `–1`:

```
"currencyTimeLimit=-1"
```

then the value is never stale. This is appropriate for static information or information that the managed resource populates in the model MBean. These types of managed resources may not be able to support or tolerate interruptions.

4.6.2 Values and Validation

The attribute descriptor contains three fields that govern the attribute's value outside of the caching algorithm: `value`, `default`, and `legalValues`. The `value` field is set by a `setAttribute()` operation and by caching support. It can be set statically and not updated from a managed resource, or it can be set on every iteration of `getAttribute()`. The `default` field defines the default value to be returned on

`getAttribute()` for this attribute if there is no current data in the `value` field and a value cannot be retrieved from the managed resource. The `legalValues` field defines an array of values that are legal for the attribute. This field was meant to be a hint to a generated console or model MBean user. These values are not enforced by the model MBean or the descriptor. It is up to the model MBean user to consult and use them. Likewise it is up to the console developer to use `legalValues` appropriately.

```
"defaultValue=True"
"legalValues=True,False"
```

4.6.3 Delegation

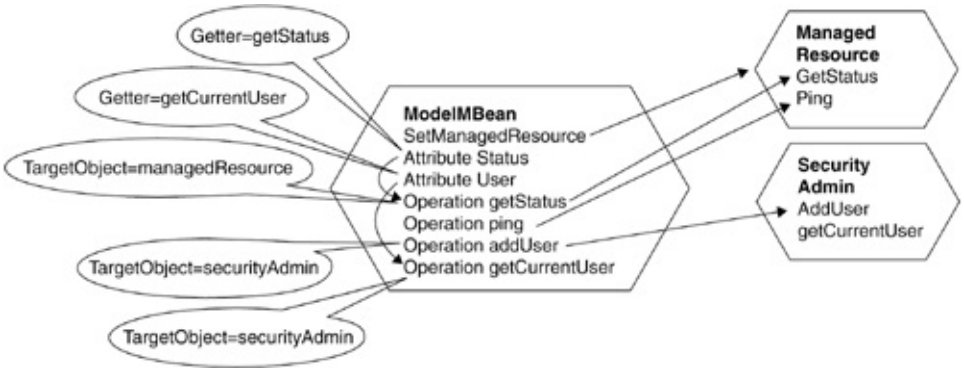
Model MBeans delegate `getAttribute()` and `setAttribute()` method calls, as well as operations, to your managed resource. In every case the method invoked on your managed resources can be any method with any signature. The attribute descriptor contains a `getMethod` field and a `setMethod` field. These fields contain the names of the operations (in the same `ModelMBeanInfo` instance) that are to be executed to satisfy the get or the set. The operation's `ModelMBeanOperationInfo` instance defines the method signatures to be invoked on the managed resource. The managed resource is set for an entire model MBean with the `setManagedResource()` method on the model MBean itself.

The interesting thing here is that the operation descriptor allows the definition of its own target object. The `targetObject` descriptor field overrides `setManagedResource()` for just that operation. The override causes the MBean to send the `invoke()` command to a new object instance, perhaps even a different class of object. This is a very powerful feature of model MBeans and allows a great deal of flexibility for defining

model MBeans for existing applications.

Figure 4.3 shows an MBean with its management interface and how attributes delegate to operations and operations delegate to methods on different target objects.

Figure 4.3. Operation Delegation to Target Objects



During initialization of the MBean itself, we would use a dashboard object as the default target for all methods:

```
setManagedObject(dashboard)
```

On the attribute descriptor for `Rate`, we would have

```
"name=Rate"
"getMethod=getRateOfSpeed"
```

Then on the operation descriptor for `getRateOfSpeed`, we would see

```
"name=getRateOfSpeed"
"targetObject=" + mphGauge
"targetObjectType=ObjectReference"
```

This means that whenever the value of the `Rate` attribute is requested, the model MBean will invoke the

`mphGauge.getRateOfSpeed()` method.

Now, the `targetObjectType` descriptor field identifies what type of reference the `targetObject` field's value is. The `targetObjectType` field may be set to `ObjectReference`, `IOR`, `EJBHandle`, or `RMIReference`. It may be set to other values as well, as long as the model MBean implementation recognizes the type and can deal with the reference appropriately. The JMX reference implementation supports only `ObjectReference`, which means that the reference in `targetObject` must be a local one. However, you can see that it was designed to work with remote objects as well.

4.6.4 Persistence

Model MBeans are responsible for persisting themselves. Persistence policy can be defined in the model MBean's descriptor and overridden in the attribute's descriptor. If a model MBean supports persistence, then some or all of the attributes may be persisted. This is important if you have highly volatile attributes (like counters) that have no meaning when they are saved. Using selective persistence, you can avoid the overhead of saving unstable, unusable data, like `currentThreadPoolSize` or `currentSessionCount`. Persistence is useful when the attributes that are persisted may be used to prime the next instantiation of the application or its model MBeans. For example, configuration attributes like `traceState` or long-running counters like `totalNumberOfInvocationsOfApplicationInJuly` would be good candidates for persistence. Persistence policy is defined by four descriptor fields: `persistName`, `persistLocation`, `persistPolicy`, and `persistPeriod`.

The `persistName` and `persistLocation` descriptor fields are defined only in the MBean-level descriptor. The `persistName` field defines the name of the model MBean in the persistence

medium. If file persistence is being used, this is the file name. If database persistence is being used, the name is the primary key for the record. The `persistLocation` field defines where the model MBean is persisted. For file-based persistence this is the fully qualified directory. For database-based persistence this is the database name.

The persistence policy descriptor field, `persistPolicy`, can be set in the model MBean descriptor or attribute descriptor. It may be set to one of four policies:

1. `persistPolicy=Never` **to switch persistence off**

- `persistPolicy=OnTimer` to force persistence at checkpoint intervals

- `persistPolicy=OnUpdate` to allow persistence whenever the model MBean is updated

- `persistPolicy=NoMoreOftenThan` to throttle the update persistence so that it does not write out the information any more frequently than a certain interval

If policies `OnTimer` or `NoMoreOftenThan` are defined, then the interval must be defined in the `persistPeriod` descriptor field. If the value of `persistPolicy` is not `OnTimer` or `NoMoreOftenThan`, then the `persistPeriod` field is ignored. The following combinations are valid in any MBean descriptor or attribute descriptor:

```
"persistPolicy=Never"

"persistPolicy=OnTimer"
"persistPeriod=180"

"persistPolicy=OnUpdate"
```

```
"persistPolicy=NoMoreOftenThan"
"persistPeriod=10"
```

When persistence policy is defined in the MBean descriptor, the directory and file names (or database and key, depending on the implementation) for the MBean are required. In such cases the descriptor would contain the following:

```
"persistPolicy=OnTimer"
"persistPeriod=180"
"persistName=MBean_File_Name"
"persistLocation=MBean_Directory_Name"
```

The algorithm that the model MBean uses is predictable. Whenever a model MBean detects that an attribute has been updated, or when a checkpoint time has been reached, it will invoke its own `store()` method. The model MBean `store()` method must determine where the data should reside and store it there appropriately. If, for example, the model MBean were implemented as an `EJBObject` instance with container-managed persistence, the `store()` method might do nothing, instead relying on the EJB container to provide the persistence on transaction boundaries. If the model MBean is implemented as an `EJBObject` instance through bean-managed persistence, the `store()` method might be responsible for writing the data to the appropriate data store.

The model MBean's `load()` method is invoked when a model MBean is being constructed and is set to prime itself from the saved data. Later in this chapter there is an example of how to override the model MBean's `load()` and `store()` methods to use XML-formatted files.

The MBeanServer's persistence setting applies to all of its model MBeans unless a model MBean defines overriding policies.

4.6.5 Logging

The descriptors that are used for setting logging policy are `log` and `logFile`. Logging policy can be set in the model MBean's descriptor to define default policy for all notifications in the model MBean. The log descriptor field is a Boolean value set to `T` (`true`) or `F (false)`. If the field is set to `T`, then the `logFile` descriptor field must be set to a valid, fully qualified file name (including the drive if appropriate) in the system. If the file does not exist, then one is created. If log is set to `T` and `logFile` is not set, no logging will occur. The default logging policy, if log is not specified, is `F`, or no logging. The default logging policy can be overridden by the logging policy of any notification's descriptor. Different notifications can be directed to different log files:

```
"log=T",
"logFile=jmx.log"
```

These descriptor fields on the MBean or notification descriptors will log the notification to the `jmx.log` file. If the descriptor were

```
"log=F",
"logFile=jmx.log"
```

then the notification would not be logged to any file, even if one were defined. The `logFile` descriptor would be ignored.

4.6.6 Miscellaneous Descriptors

4.6.6.1 Presentation String

The `presentation` descriptor field was intended to hold XML

fragments that describe how to display the model MBean, attribute, operation, or notification. This information is useful to a generic or third-party console only if this XML fragment is standardized to some degree. This standardization, or agreement on what the important elements were, was not done for the first release of the JMX specification. However, the `presentation` field can still be very useful for building specific management systems for applications. Some information that we thought would be in this string was a GIF or a BMP file name for a small or large icon. If it were a configuration field, it might contain information on whether a list box, radio dial, range, or text box should be used to set the v alue. Here is an example of a `presentation` string that associates a GIF file with an attribute:

```
"presentation=tooFast.gif"
```

4.6.6.2 Visibility

Because JMX was intended to support both third-party, enterprise class management systems and application-specific management systems, it quickly became obvious that the amount of information an enterprise manager wants to display could be much less in volume and much coarser grained than the information displayed and managed by an application-specific management system. Even within application-specific management systems, in order to support the goal for allowing console generation for applications, some rating was needed about the granularity of this information.

This rating was used to decide whether to put the attribute or operation information on the "first page" (i.e., the resource status or summary page), or on a detail page where the user had to click on something in order to access the page. The `visibility` descriptor was meant to help express this concept. It allows attributes, operations, and notifications to be

ranked from 1 to 4. The value 1 is supposed to indicate the coarsest-grained, most often visible elements, and 4 is supposed to indicate the finest-grained, more advanced, least often visible elements. An enterprise manager may ask only for model MBeans with a `visibility` value of 1. Or if displaying a resource summary page, the enterprise manager would display only attributes with a `visibility` value of 1. If the resource w ere clicked on, another page would be displayed that contained all of the attributes and operations regardless of visibility. Of course, assigning `visibility` values is very subjective and should be done by someone who understands the resource and the management systems managing that resource.

4.6.6.3 Export

The `export` field is valid only in theMBean-level descriptor. It signifies to the MBeanServer and model MBean constructor that this model MBean should be advertised so that other MBeanServers or remote management systems can find it. This field was intended to allow a management system to locate an MBean without necessarily knowing which MBeanServer houses it beforehand. One common place that an MBean may be exported to is a directory accessible via JNDI. If the `export` field's value is set to `null`, then the MBean is not exported. If it is not `null`, then its value is used by the export support in the MBeanServer of the model MBean itself to perform the export. So, for example, if the MBean were to be registered via JNDI, the `export` descriptor's value would contain the JNDI directory URI (Uniform Resource Identifier) and the JNDI name to be used. If the MBean were to be simultaneously registered with several MBeanServers, the `export` value might be the MBeanServer's host name and domain name. Here is an example:

```
"export=jndiserver:8080:SpeedBean"
```

Not all MBeanServers support the export of model MBeans. In fact, the JMX reference implementation does not. If the MBeanServer and model MBean implementations do not support it, this field is simply ignored. The default value for `export` is `null`.

4.6.6.4 Protocol Map

JMX was intended to allow the translation of MBean data and data models into other management technologies and their data models. This translation cannot always be generated. Sometimes JMX management data must be mapped to a specific data model in a specific technology. For example, you might define a set of MBeans for your HTTP Web server. The IETF[14] has also defined a MIB for managing Web servers.[15]

If you know that you want your MBean attributes to be accessed by SNMP[16] managers through the HTTP MIB,[17] then you can define a protocol map in the attribute descriptor that will define the protocol and object to map to. You can define multiple mappings for your attribute in the protocol map. The same attribute can be mapped to an SNMP MIB and a CIM[18] object property.

The `protocolMap` descriptor contains a reference to a `protocolMap` object. The `protocolMap` object is actually a descriptor with name/value pairs. The name is the protocol name, and the value is the mapped object name for the protocol for this attribute.

So for an MBean attribute `status` with SNMP and CIM maps, you would have the following name/value pairs:

```
"SNMP=1.3.6.1.1.1.3.1.1.5.1" // maps to this SN
"CIM=cim_service.status"     // maps to the sta
```

```
                                                    // the cim_service
```

Adapters will be the ones accessing and using the protocol map to translate the attributes into their own data model correctly. The adapters will have to recognize the protocol names as hints for their translation.

Attributes do not have to have specific protocol maps for all potential protocols that might make the attributes available through adapters. The adapter could generate a generic mapping for all attributes in all MBeans. The mapping would be realized and enforced by the adapter, not by the definition of a protocol map. For example, a MIB generator could be executed when new JMX MBeans were registered. This MIB generator would create a new SNMP MIB that would be loaded by the SNMP management system and used by the SNMP adapter. This is convenient because all the MBean's protocol map descriptors do not have to be updated whenever a new adapter to a new management protocol is registered with the MBeanServer.

4.7 XML Service: Priming ModelMBeanInfo from XML Files

Creating `ModelMBeanInfo` instances in your code using the available `ModelMBeanInfo` APIs is very tedious. Using the API also means that your management instrumentation code has to be updated every time you have a slight change in your attributes or operations that you want to make available. It makes some sense to provide `ModelMBeanInfo` customization data in a separate file that accompanies the managed application. This file could be formatted as a properties file; however, choosing an XML format allows more tooling support and portability. This file would be used to customize `ModelMBeanInfo` at runtime. Support for creating MBeans from XML files and writing MBeans to XML files is not part of the JMX specification or reference implementation. Using an XML file does illustrate a more convenient and robust means to customize your model MBeans. It is also a useful illustration of how to create and use MBean services. We will illustrate one technique for providing XML services support here.

In this case we have created a service MBean: `XMLService`. A *service MBean* is an MBean that is used by other MBeans as a utility and doesn't actually represent a managed resource. The `XMLService` MBean can create an XML file from a model MBean, or it can read an XML file and return a model MBean, or `ModelMBeanInfo` instance. It is possible to choose a number of different XML DTDs (document type definitions) to guide the file's format.

One choice is to use the DTD used by CIM.[19] Reasons to use the CIM DTD include the following: It is freely available; it is a generally accepted standard for management data representation; and it defines a mechanism to express management objects, attributes, and operations, as well as metadata (qualifiers) about them. One drawback to using the CIM DTD is that you will have to modify it to add support for

expressing notifications. These features make the CIM DTD an easy fit for JMX MBean representation.

An alternative representation for the XML file is the DTD used by the Apache Jakarta Project's Commons Modeler version 1.0. [20] The modeler uses a special DTD (at http://jakarta.apache.org/commons/modeler/mbeans-descriptors.dtd) that is a relatively straightforward use of XML elements for each of the `MBeanInfo` information types. At the time of this writing, the implementation of the Modeler did not fully support descriptors in `ModelMBeanInfo` objects. We were able to add that support and an updated version of the modeler with the support is available as part of the downloadable examples for this book on the book's Web site: http://www.awprofessional.com/titles/0672324083. Full support of descriptors is required to run these examples. The full source for these examples and the `XMLService` MBean are available on the Addison-Wesley Web site for this book. The Modeler reads XML files and creates `ModelMBean` instances. It does not provide support for creating the XML files. In order to create the XML Service envisioned here, we created our own interface that supports creating `ModelMBean` instances from an XML file, creating `ModelMBeanInfo` instances from an XML file, variations with and without the managed object, and writing an XML file.

Here is the API for the `XMLService` MBean:

```
package jnjmx.ch4;
import javax.management.*;
import javax.management.modelmbean.*;

public interface XMLServiceMBean {
  Object createMBeanFromXMLFile(String fileName
                                ObjectName mbNa
  MBeanInfo createMBeanInfoFromXMLFile(String f
```

```
    String name);

  Object createMBeanFromXMLFile(String fileName
                                Object managedR
  MBeanInfo createMBeanInfoFromXMLFile(String f
                                Object managedR

  Object createMBeanFromXMLFile(String fileName
                                Object managedR
  void writeXMLFile( ModelMBean mb, String file
}
```

First let's look at how to use `XMLService` to create a model MBean for the Apache model MBean:

```
/** Create a name for the MBean */
ObjectName apacheMBeanName = new ObjectName
  ("apacheManager: id=ApacheServer");

/** Instantiate the ApacheServer utility to be
* managed resource */
serverURL = "http://www.apache.org/server-statu
Apache myApacheServer = new Apache(serverURL);

/** Instantiate a model MBean from the XML file
ModelMBean apacheMMBean = (ModelMBean)
  XMLPrimer.createMBeanFromXMLFile(
    "jnjmx.ch4/apachembeans.xml",  // name of X
    apacheMBeanName,               // name of M
    myApacheServer);               // managed r

/ ** Register the model MBean that is now ready
* to run with the MBeanServer */
```

```
  ObjectInstance registeredApacheMMBean =
        myMBS.registerMBean(apacheMMBean,
        apacheMBeanName);
//** Write the XML file
writeXMLFile( apacheMMBean, "jnjmx.ch4/apachemb
```

You can see that first we create the `XMLService` MBean. Then we use `XMLService`'s `createMBeanFromXMLFile()` method to read the file and instantiate a model MBean. We also use `XMLService` to write the MBean's `ModelMBeanInfo` to a file called `apachembeans.xml`. This would be important to do if some of the configuration of the model MBean had been changed—that is, caching or logging policy—that needed to be persisted for the next instantiation of this model MBean.

Now we are going to look at how to use the XML service as a service MBean. Here is how to instantiate an `XMLService` MBean:

```
/** Find an MBeanServer for our MBean */
MBeanServer myMBS = MBeanServerFactory.createMB
// Create a JMX object name for the XML service
ObjectName sn = new ObjectName("todd:id=XMLPrim
// Instantiate the XML service

XMLPrimer = new XMLService();
ObjectInstance registeredXMLService =
  mbs.registerMBean(XMLPrimer, sn);
```

We created an instance of `XMLService`, created an object name for it, and registered it with the MBeanServer.

Here is how another adapter or JMX manager can use the same `XMLService` MBean to create a model MBean. In this case we have used a variation of the API that does not require the

managed resource. The managed resource is associated with it later:

```
/** Find an MBeanServer for our MBean */
MBeanServer myMBS = MBeanServerFactory.createMB

/** Instantiate the ApacheServer utility to be
* resource */
serverURL = "http://www.apache.org/server-statu
Apache myApacheServer = new Apache(serverURL);

/** Create the XML service's name */
XMLServiceName=("todd:id=XMLPrimer");
/** Instantiate an ModelMBean using the already
* XMLService */
ModelMBean apacheMMBean = (ModelMBean) mbs.invo
  xmlServiceName, "createMBeanFromXMLFile",
  new Object[] {"jnjmx.ch4/apachembeans.xml", /
                "id=ApacheServer"); // name of
  new String[] {"java.lang.String",
                "java.lang.String"});
/** Assign the managed resource to the MBean, s
* it in this time */
apacheMMBean.setManagedResource(myApacheServer,
/** Register the model MBean that is now ready
* MBeanServer */
ObjectInstance registeredApacheMMBean =
  myMBS.registerMBean(apacheMMBean,
                      apacheMBeanName);
```

The XML file containing the `ModelMBeanInfo` data using the Jakarta Modeler DTD is listed in full at the end of this chapter.

4.8 Using Model MBeans

Let's review what we know about model MBeans, and then we'll go over some examples and strategies for using them.

The MBeanServer functions as a factory for model MBeans. Therefore, model MBean instances should be created and maintained by the MBeanServer. Using the factory allows the `RequiredModelMBean` class implementation to vary depending on the needs of the environment in which the MBeanServer is installed. Your application that requests the instantiation of `RequiredModelMBean` does not have to be aware of the specifics of the `RequiredModelMBean` implementation. There are many potential implementation differences between JMX and JVM environments, including persistence, transactional behavior, caching, performance requirements, location transparency, remotability, and so on. `RequiredModelMBean` is responsible for implementing and managing these differences internally. Keep in mind that the MBeanServer that creates `RequiredModelMBean` may be specifically designed to support a particular `RequiredModelMBean` class. In fact, if you supply your own `RequiredModelMBean` implementation and try to use it, you may run into compatibility problems.

Because the JMX implementation provides the model MBean implementation, your application does not have to implement the model MBean, but just customize and use it. Your application's instrumentation code is consistent and minimal. You get the benefit of default policy and support for event logging, data persistence, data caching, and notification handling. In your application, you initialize your model MBean and pass in your identity, management interface, and any policy overrides. You can add custom attributes to the model MBean during execution; therefore information specific to the application can be modified without interruption during runtime. The model MBean then sets the behavior interface for the

MBean and does any setup necessary for event logging and handling, data persistence and caching, and monitoring for your application's model MBean instance. The model MBean default behavior and simple APIs are intended to satisfy the management needs of most applications, but they also allow more complex application management scenarios.

You should have one instance of a model MBean for each instance of a resource (application, device, and so forth) to be managed in the system. These instances define the specific characteristics of operations, invocations, event trigger rules, event types, expiration rules, event logging, persistence, and so on for a given resource managed by the MBeanServer. Each instance of a model MBean may have completely different attributes, operations, and policies because they may manage completely unrelated resources.

In the following sections we will implement the time of day (todd) server example from Chapter 1 using model MBeans and showing various approaches for doing so. This would be useful if the todd server were already implemented and deployed such that going back into the implementation and adding instrumentation would be difficult and time-consuming.

We are going to look at two styles of model MBeans we can create: static and dynamic. Static model MBeans have their `ModelMBeanInfo` set during development through the `ModelMBeanInfo` and descriptor programming APIs. Dynamic MBeans create `ModelMBeanInfo` during execution from information that is available only while a system is running—that is, an XML file or interaction with another runtime system.

4.8.1 Creating Static Model MBeans

Static model MBeans have `ModelMBeanInfo` that is known and set during development. The developer knows what the attributes, operations, and policies will be and uses the

`ModelMBeanInfo` APIs to set them. This type of model MBean can be instantiated with a `new` operation or a `create` on the MBeanServer.

Here is an example of a static model MBean approach in the todd server. Here is the todd server without an MBean being implemented as part of the todd application class. The main loop and `Server` constructor have been pulled out for discussion later:

```
package jnjmx.ch4;
// Create model MBeans for Server and Session f
// Create ModelMBeanInfo from program APIs
import java.util.SortedSet;
import java.util.TreeSet;
import java.lang.reflect.Constructor;
import javax.management.*;
import javax.management.modelmbean.*;

public class Server {
  private MBeanServer mbs;

  private SessionPool sessions;
  private SortedSet connectionQueue;
  private Listener listener;
  private boolean active;

  static boolean tracing = false;

  private long tzero;
  private int connections;
  // MAIN LOOP WILL GO HERE
  // SERVER CONSTRUCTOR WILL GO HERE
  public void activateSession(Connection k) {
```

```
  Session s;
  synchronized (sessions) {
    while ((s = sessions.reserve()) == null)
      try {
        sessions.wait();
      } catch (InterruptedException x) {
        // see if a session is available
      }
    }
  }
  s.activate(k);
  connections++;
}

/**
 * Connections attribute getter
 * @returns total number of connections handl
 */
public Integer getConnections() {
  return new Integer(connections);
}

/**
 * Sessions attribute getter
 * @returns number of active sessions
 */
public Integer getSessions() {
  int as = sessions.getAvailableSessions().in
  int sz = sessions.getSize().intValue();
  return new Integer(sz - as);
}

/**
```

```
 * Uptime attribute getter
 * @returns number of milliseconds since the
 */
public Long getUptime() {
  return new Long(System.currentTimeMillis()
}

public boolean isActive() {
  return active;
}

/**
 * Shut down the server, killing the process
 * sessions
 */
public void shutdown() {
  System.exit(0);
}
/**
 * Start a listener thread that will queue in
 */
public void start() {
  listener = new Listener(connectionQueue);
  listener.start();
}

/**
 * Stop the server's listener thread; active
 * to handle requests
 */
public void stop() {
  listener.stopListening();
}
```

```
  public Connection waitForConnection() {
    Connection k;
    synchronized (connectionQueue) {
      while (connectionQueue.isEmpty()) {
        try {
          connectionQueue.wait();
        } catch (InterruptedException x) {
          // See if the queue is still empty
        }
      }
      k = (Connection) connectionQueue.first();
      connectionQueue.remove(k);
    }
    return k;
  }
```

Here is the `SessionPool` class without an integrated MBean:

```
package jnjmx.ch4;
import java.util.HashSet;
import java.util.Set;
public class SessionPool  {
  private static final int POOLSIZE = 8;

  private Set sessions;
  private int size;

  public SessionPool() {
    this(POOLSIZE);
  }

  public SessionPool(int size) {
```

```
  this.size = size;
  sessions = new HashSet(this.size);
  fill();
}

public synchronized void grow(int increment)
  for (int i = 0; i < increment; i++) {
    Session s = new Session(this);
    sessions.add(s);
  }
  size = size + increment;
}

public Integer getSize() {
  return new Integer(size);
}
public synchronized Integer getAvailableSessi
  return new Integer(sessions.size());
}

public synchronized void release(Session sess
  sessions.add(session);
  notify();
}

public synchronized Session reserve() {
  while (sessions.isEmpty()) {
    try {

      wait();
    } catch (InterruptedException x) {
      // see if the set is still empty
    }
```

```
    }
    Session s = (Session) sessions.iterator().n
    sessions.remove(s);
    return s;
  }

  private void fill() {
    for (int i = 0; i < size; i++) {
      Session s = new Session(this);
      sessions.add(s);
    }
  }
}
```

Here's the main program loop with a model MBean added for the todd and session pool management. Note that `ModelMBeanInfo` is defined in the program and set using the `ModelMBeanInfo` APIs. To make any adjustments would require recompiling the resource. In this example the main loop is responsible for instantiating the server and its model MBean. In the next code block we will see where the server's constructor is responsible for instantiating the session pool and its model MBean:

```
public static void main(String[] args) throws E
  try {

  /* management stuff */
  /** Find an MBeanServer for our MBean */
  MBeanServer mbs = MBeanServerFactory.createMB

  /** Instantiate the todd server */
  Server server = new Server(mbs);
```

```
/** Create a name for the MBean */
ObjectName son = new ObjectName("todd:id=Serv

/** Create an unconfigured RequiredModelMBean
* MBeanServer */
  RequiredModelMBean serverMMBean = (Required

  mbs.instantiate(
      "javax.management.modelmbean.RequiredMo

/** Set the configuration of the ModelMBean t
 ** This method is below */
ModelMBeanInfo serverMMBeanInfo =
    server.createServerModelMBeanInfo(server,
serverMMBean.setModelMBeanInfo(serverMMBeanIn

/** Set the todd server as the managed resour
serverMMBean.setManagedResource(server,"Objec

/** Register the model MBean that is now read
* MBeanServer */
ObjectInstance registeredServerMMBean =
  mbs.registerMBean((Object) serverMMBean, so

mbs.invoke(son, "start", new Object[] {}, new
// executes method on MBean
System.out.println("starting connection loop"

while (server.isActive()) {
  Connection k = server.waitForConnection();
  server.activateSession(k);
  System.out.println("*");
}
```

```
  } catch (Exception e) {
    System.out.println("todd Server failed with
        e.getMessage());
  }
 }
```

Here is the `Server` constructor that shows how the session pool and the session pool's model Bean are instantiated:

```
public Server(MBeanServer mbs) throws Exception
  this.mbs = mbs;
  connectionQueue = new TreeSet();
  connections = 0;
  sessions = new SessionPool();             // c
  ObjectName spon = new ObjectName("todd:id=Ses
  // name of mbean
  // Instantiate the model MBean

  RequiredModelMBean sessionMMBean = (RequiredM
    mbs.instantiate
    ("javax.management.modelmbean.RequiredModel
  // Set MBeanInfo
  ModelMBeanInfo sessionMMBInfo =
        createSessionModelMBeanInfo(sessions, s
  sessionMMBean.setModelMBeanInfo(sessionMMBInf
  // Set the managed resource
  sessionMMBean.setManagedResource(sessions, "O
  // Register the model MBean with the MBeanSer
  ObjectInstance registeredSessionMMBean =
        mbs.registerMBean((Object) sessionMMBea
  System.out.println("created Session ModelMBea
```

```
  active = true;
  tzero = System.currentTimeMillis();
}
```

The `Server` model MBean has operations `start()`, `stop()`, and `shutdown()`. It has attributes `Connections`, `Sessions`, and `Uptime`. Here are the routines to set `ModelMBeanInfo`:

```
// Create the ModelMBeanInfo instance for the t
ModelMBeanInfo createServerModelMBeanInfo(Serve
    ObjectName serverMMBeanName) {
  Class serverClass = null;
  try {
    serverClass = Class.forName("jnjmx.ch4.Serv
  } catch(Exception e) {
    System.out.println("Server Class not found"
  }

  // Set the MBean descriptor
  Descriptor mmbDescription = new DescriptorSup
      ("name="+serverMMBeanName),        // ser
      "descriptorType=mbean",
      "displayName=toddServerManager",
      "log=T",                           // log
      "logFile=jmxserver.log",           // in
      "currencyTimeLimit=10"});          // cac
                                         // val

  // Define attributes in ModelMBeanAttributeIn
  ModelMBeanAttributeInfo[] mmbAttributes =
    new ModelMBeanAttributeInfo[3];

  // Declare Connections attribute
```

```
Descriptor connectionsDesc = new DescriptorSu
    new String[] {
    "name=Connections",             // attrib
    "descriptorType=attribute",
    "displayName=todd connections",
    "getMethod=getConnections",     // call g
                                    // for ge
    "currencyTimeLimit=-1"});       // don't

mmbAttributes[0] = new ModelMBeanAttributeInf
     "Connections",
     "java.lang.Integer",           // attrib
     "todd Server connections",     // descri
     true,
     false,
     false,
     connectionsDesc);              // descri

// Declare Sessions attribute
Descriptor sessionsDesc = new DescriptorSuppo
    new String[] {
    "name=Sessions",            // attribute
    "descriptorType=attribute",
    "displayName=todd server sessions",
    "getMethod=getSessions",    // call opera
    "currencyTimeLimit=5"});    // cache valu

mmbAttributes[1] = new ModelMBeanAttributeInf
    "Sessions",
    "java.lang.Integer",      // type for ses
    "todd Server sessions",
    true,
    false,
```

```
    false,
    sessionsDesc);            // descriptor f

// Declare Uptime attribute
Descriptor uptimeDesc = new DescriptorSupport
    new String[] {
    "name=uptime",              // attribute
    "descriptorType=attribute",
    "displayName=todd server sessions",
    "getMethod=getUptime",      // call opera
    "currencyTimeLimit=0"});    // don't cach

mmbAttributes[2] = new ModelMBeanAttributeInf
    "java.lang.Integer",      // type for ala
    "todd Server length of time up",
    true,
    false,
    false,
    uptimeDesc);        // descriptor for upt

// Declare constructors for managed resource
ModelMBeanConstructorInfo[] mmbConstructors =
    new ModelMBeanConstructorInfo[1];
Constructor[] myConstructors = serverClass.ge

Descriptor mmBeanDesc = new DescriptorSupport
    "name=ServerClass",
    "descriptorType=operation",
           "role=constructor" });

mmbConstructors[0] = new ModelMBeanConstructo
    "Server(): Constructs a todd Server",
    myConstructors[0],
```

```
    mmBeanDesc);

// Define operations in ModelMBeanOperationIn
/* Operations: start, stop, shutdown
 * getUptime, getConnections, getSessions
 */

// Declare getValue operation String getValue
// Set parameter array
ModelMBeanOperationInfo[] mmbOperations =
    new ModelMBeanOperationInfo[6];

// Declare start() operation void start()
// Set parameter array
MBeanParameterInfo[] parms = null;

Descriptor startDesc = new DescriptorSupport(
    "name=start",                // operation
    "descriptorType=operation",
    "class=server",              // class to
    "role=operation" });         // this is a

mmbOperations[0] = new ModelMBeanOperationInf
    "start server",               // descript
    parms ,                       // paramete

    "java.lang.boolean",          // return t
     MBeanOperationInfo.ACTION,   // start ch
     startDesc);                  // descript
Descriptor stopDesc = new DescriptorSupport(n
    "name=stop",                 // operation
    "descriptorType=operation",
    "class=server",              // class to
```

```
    "role=operation" });         // this is a

mmbOperations[1] = new ModelMBeanOperationInf
    "stop server",               // descripti
    parms ,                      // parameter
    "java.lang.boolean",         // return ty
    MBeanOperationInfo.ACTION,   // stop chan
    stopDesc);                   // descripto

Descriptor shutdownDesc = new DescriptorSuppo
    "name=shutdown",             // operation
    "descriptorType=operation",
    "class=server",              // class to
    "role=operation" });         // this is a

mmbOperations[2] = new ModelMBeanOperationInf
    "shutdown",                  // name
    "shutdown server",           // descripti
    parms ,                      // parameter
    "java.lang.boolean",         // return ty
    MBeanOperationInfo.ACTION,   // changes s
    shutdownDesc);               // descripto

// Declare getConnections operation String ge
Descriptor getConnectionsDesc =
  new DescriptorSupport(new String[] {
    "name=getConnections",
    "descriptorType=operation",
    "class=Server",              // class to
    "role=getter"});             // this is a

MBeanParameterInfo[] noParms = null;
```

```
mmbOperations[3] =
  new ModelMBeanOperationInfo("getConnections
    "get Connections attribute",   // descrip
    noParms,                       // null parame
    "java.lang.Integer",           // return type
    MBeanOperationInfo.INFO,       // does not ch
    getConnectionsDesc);           // descriptor

// Declare getSessions operation - void getSe
Descriptor getSessionsDesc = new DescriptorSu
    "name=getSessions",            // name
    "descriptorType=operation",
    "class=Clock",                 // target c
    "role=getter"});               // operatio

mmbOperations[4] = new ModelMBeanOperationInf
    "getSessions",                 // name
    "get Sessions attribute",      // descript
    noParms,                       // paramete
    "java.lang.Integer",           // no retur
    MBeanOperationInfo.INFO,       // operatio
    getSessionsDesc);              // setSessi

// Declare getUptime operation
Descriptor getUptimeDesc = new DescriptorSupp
    "name=getUptime",              // name
    "descriptorType=operation",
    "class=Server",                // target c
    "role=getter"});               // operatio

mmbOperations[5] = new ModelMBeanOperationInf
    "getUptime",               // name
    "get uptime attribute",  // description
```

```
        noParms,                  // no parameters
        "java.lang.Integer",      // return type i
        MBeanOperationInfo.INFO, // operation doe
        getUptimeDesc);           // uptime descri

    // Define no notifications in ModelMBeanNotif
    ModelMBeanNotificationInfo[] mmbNotifications
        new ModelMBeanNotificationInfo[0];

      // Create ModelMBeanInfo
      ModelMBeanInfo mmBeanInfo = new ModelMBeanI
        "Server",
        "ModelMBeanInfo for managing a todd Serve
        mmbAttributes,
        mmbConstructors,
        mmbOperations,
        mmbNotifications);
    try {

      mmBeanInfo.setMBeanDescriptor(mmbDescriptio
    } catch (Exception e) {
      System.out.println("CreateServerMBeanInfo f
          e.getMessage());
    }
    return mmBeanInfo;
    }
```

The `Session` model MBean has operation `grow()` and attributes `availableSessions` and `size`:

```
ModelMBeanInfo createSessionModelMBeanInfo(Sess
          mmBResource, ObjectName mmbName) {
    // Create the ModelMBeanInfo instance for the
```

```
Class targetClass = null;
try {
  targetClass = Class.forName("jnjmx.ch4.Sess
} catch(Exception e) {
  System.out.println("Server Class not found"
}

// Set the MBean descriptor
Descriptor mmbDescription = new DescriptorSup
    ("name="+mmbName),           // MBean nam
    "descriptorType=mbean",
    ("displayName=Session Pool Manager"),
    "log=T",                     // log all n
    "logfile=jmxsession.log",    // in jmxmai
    "currencyTimeLimit=10"});    // cache all
                                 // for 10 se

ModelMBeanAttributeInfo[] mmbAttributes =
    new ModelMBeanAttributeInfo[2];
// Declare Connections attribute
Descriptor sizeDesc = new DescriptorSupport(n
    "name=Size",          // attribute name
    "descriptorType=attribute",
    "displayName=todd size",
    "getMethod=getSize" });

mmbAttributes[0] = new ModelMBeanAttributeInf
     "java.lang.Integer",      // attribute t
     "todd Server size",      // description
     true,

     false,
```

```
      false,
      sizeDesc);        // descriptor for attr

// Declare availSessions attribute
Descriptor availSessionsDesc =
  new DescriptorSupport(new String[] {
    "name=AvailableSessions",         // attr
    "descriptorType=attribute",
    "displayName=todd server available sessio
    "getMethod=getAvailableSessions", // call
    "currencyTimeLimit=5"});

mmbAttributes[1] = new ModelMBeanAttributeInf
    "java.lang.Integer",      // type for ses
    "todd Server available sessions",
    true,
    false,
    false,
    availSessionsDesc);      // descriptor fo

// Declare constructors for MBean
Constructor[] constructors = targetClass.getC
ModelMBeanConstructorInfo[] mmbConstructors =
  new ModelMBeanConstructorInfo[1];
Descriptor mmBeanDesc = new DescriptorSupport
    "name=SessionClass",
    "descriptorType=operation",
    "role=constructor" });

mmbConstructors[0] = new ModelMBeanConstructo
    "SessionPool(): Constructs a todd Server"
    constructors[0],
    mmBeanDesc);
```

```
ModelMBeanOperationInfo[] mmbOperations = new
    ModelMBeanOperationInfo[3];

// Declare grow() operation void grow()
// Set parameter array
MBeanParameterInfo[] noParms = null;

Descriptor growDesc = new DescriptorSupport(n
    "name=grow",               // operation/m
    "descriptorType=operation",

    "class=server",            // class to ru
    "role=operation" });       // this is an

mmbOperations[0] = new ModelMBeanOperationInf
    "grow session pool",        // descriptio
    noParms ,
    "java.lang.boolean",
    MBeanOperationInfo.ACTION,  // changes st
    growDesc);                  // descriptor


// Declare getSize operation – String getSize
Descriptor getSizeDesc = new DescriptorSuppor
    "name=getSize",
    "descriptorType=operation",
    "class=Sessions",      // class to run op
    "role=getter"});       // this is a gette

mmbOperations[1] = new ModelMBeanOperationInf
    "getSize", // name
    "get Size attribute",     // description
```

```
    noParms,                    // null paramet
    "java.lang.Integer",        // return type
    MBeanOperationInfo.INFO,    // does not cha
    getSizeDesc);               // descriptor f

// Declare getAvailableSessions operation - v
Descriptor getAvailableSessionsDesc =
  new DescriptorSupport(new String[] {
    "name=getAvailableSessions",    // name
    "descriptorType=operation",
    "class=Sessions",               // target
    "role=getter"});                // operat

mmbOperations[2] = new ModelMBeanOperationInf
    "getSessions",                  // name
    "get Sessions attribute",       // descripti
    noParms,                        // parameter
    "java.lang.Integer",            // no return
    MBeanOperationInfo.INFO,        // does not
    getAvailableSessionsDesc);      // setAvaila
                                    // descripto

ModelMBeanNotificationInfo[] mmbNotifications
  new ModelMBeanNotificationInfo[0];

// Create ModelMBeanInfo
ModelMBeanInfo mmBeanInfo = new ModelMBeanInf
    "ModelMBeanInfo for managing todd Session
    mmbAttributes,
    mmbConstructors,
    mmbOperations,
    mmbNotifications);
try {
```

```
    mmBeanInfo.setMBeanDescriptor(mmbDescriptio
  } catch (Exception e) {
    System.out.println("CreateServerMBeanInfo f
      e.getMessage());
  }
  return mmBeanInfo;
  }
```

4.8.2 Creating Dynamic Model MBeans

Dynamic model MBeans have `ModelMBeanInfo` that is known and set during the runtime. Either introspection or external files are used to set this `ModelMBeanInfo`. The developer may use a service to create a `ModelMBeanInfo` instance. One thing to keep in mind is that `ModelMBeanInfo` for a newly instantiated model MBean includes all the methods on the model MBean. You will need to override this `ModelMBeanInfo` to remove methods you do not want adapters to have, like `setManagedResource()`.

The example that follows is of a dynamic model MBean approach using an XML to `MBeanInfo` service as described earlier. Basically the main loop from the previous example uses the XML service to populate `ModelMBeanInfo`. Now we can change the management interface by changing the XML file and and without having to recompile the source. This is a more flexible and dynamic model.

Here's the main program loop and `Server` constructor with a model MBean added for the todd and session pool management primed by the XML service:

```
private static XMLService XMLPrimer;
public static void main(String[] args) throws E
  /* management stuff */
```

```
/** Find an MBeanServer for our MBean */
MBeanServer mbs = MBeanServerFactory.createMB

// Instantiate the XML service
ObjectName sn = new ObjectName("todd:id=XMLPr
XMLPrimer = new XMLService(sn);
String serverMBeanInfoFile = "ServerMBean.xml

/** Instantiate the todd server */
Server server = new Server(mbs);

/** Create a name for the MBean */
ObjectName son = new ObjectName("todd:id=Serv

/** Set the configuration of the model MBean
   * XMLPrimer service, which sets ModelMBean
   * XML file
 ** */

RequiredModelMBean serverMMBean = (RequiredMo
  XMLPrimer.createMBeanFromXMLFile(serverMBea
/** Create an unconfigured RequiredModelMBean
    MBeanServer */
/** Set the todd server as the managed resour
serverMMBean.setManagedResource(server,"Objec

/** Register the model MBean that is now read
 * MBeanServer */
ObjectInstance registeredServerMMBean =
  mbs.registerMBean((Object) serverMMBean, so

/* server stuff */
mbs.invoke(son, "start", new Object[] {}, new
```

```
  while (server.isActive()) {
    Connection k = server.waitForConnection();
    server.activateSession(k);
  }

  } catch (Exception e) {
    System.out.println("todd Server failed with
      e.getMessage());
  }
 }

public Server(MBeanServer mbs) throws Exception
  this.mbs = mbs;

  connectionQueue = new TreeSet();
  connections = 0;

  sessions = new SessionPool();
  ObjectName spon = new ObjectName("todd:id=Ses

  String sessionMBeanInfoFile = "SessionMBean.x
  RequiredModelMBean sessionMMBean = (RequiredM
    XMLPrimer.createMBeanFromXMLFile(sessionMBe
    sessionMMBean.setManagedResource(sessions,
     ObjectInstance registeredSessionMMBean =
        mbs.registerMBean((Object) sessionMMBea

  active = true;
  tzero = System.currentTimeMillis();
}
```

4.8.2.1 Creating Model MBeans from the Managed Resource

Model MBeans may be instantiated, customized, and registered by the managed resources themselves. This will usually be done during initialization of the managed resource. The managed resource then updates the data in the model MBean whenever an update is necessary. In this case `managedResource` will be set to `this()` so that all the operations are delegated back to the object that is creating the model MBean. Here is an example of a self-managing service—the `Server` class that creates its own model MBean:

```
package jnjmx.ch4;
import java.util.SortedSet;
import java.util.TreeSet;
import java.lang.reflect.Constructor;
import javax.management.*;
import javax.management.modelmbean.*;

// Managed server creates model MBeans and Mode
// XML files
// In this example we show how to create the sa
// using an XML service to create ModelMBeanInf

// Once again, the main loop of the Server (4)
// server
// The server's constructor instantiates the se
// server's model MBean
// SessionPool's constructor instantiates Sessi


public class Server {
```

```
private MBeanServer mbs;

private SessionPool sessions;
private SortedSet connectionQueue;
private Listener listener;
private boolean active;
static boolean tracing = false;
private long tzero;
private int connections;

private static XMLService XMLPrimer;
  public static void main(String[] args) thro
    /* server stuff */
  Server server = new Server();
  server.start();
  while (server.isActive()) {
    Connection k = server.waitForConnection()
    server.activateSession(k);
  }

  } catch (Exception e) {
    System.out.println("todd Server failed wi
        e.getMessage());
  }
 }

public Server() throws Exception {
  /** Find an MBeanServer for our MBean */
  MBeanServer mbs = MBeanServerFactory.create
  // Instantiate the XML service
  ObjectName sn = new ObjectName("todd:id=XML
  XMLPrimer = new XMLService(sn);
  ObjectInstance registeredXMLService =
```

```
          mbs.registerMBean((Object) XMLPrimer,

    connectionQueue = new TreeSet();
    connections = 0;

    sessions = new SessionPool();

    active = true;
    tzero = System.currentTimeMillis();

    /** Create a name for the MBean */
    ObjectName son = new ObjectName("todd:id=Se

/** Set the configuration of the model MBean th
 * XMLPrimer service, which sets ModelMBeanInfo
 * XML file
** */

    String serverMBeanInfoFile = "jnjmx.ch4/Ser
      ModelMBean serverMMBean = (ModelMBean)
      XMLPrimer.createMBeanFromXMLFile(
           serverMBeanInfoFile,
           "Server" , this);

    /** Register the model MBean that is now re
    * the MBeanServer */
    ObjectInstance registeredServerMMBean =
        mbs.registerMBean(serverMMBean, son);
  }

  // The rest of the methods are the same as in
  }
```

Here is the session class with an integrated model MBean:

```
package jnjmx.ch4;
import java.util.HashSet;
import java.util.Set;
import java.util.ArrayList;
import javax.management.*;
import javax.management.modelmbean.*;

public class SessionPool  {
  private static final int POOLSIZE = 8;
  private Set sessions;
  private int size;
  private MBeanServer mbs;

  public SessionPool() {
    this(POOLSIZE);
  }

  public SessionPool(int size) {
    this.size = size;
    sessions = new HashSet(this.size);
    fill();
    createSessionPoolMBean();
  }

  private void createSessionPoolMBean() {
    try {
    // Change to use existing MBeanServer and e
    // through the MBeanServer
    ArrayList mbsList = MBeanServerFactory.find
    if ((!mbsList.isEmpty() && (mbsList.get(0)
      mbs = (MBeanServer) mbsList.get(0); // us
```

```
    else {
      System.out.println("SessionPool can't fin
      System.exit(0);
    }

    ObjectName xmlServiceName = new
        ObjectName("todd:id=XMLPrimer");
    XMLService XMLPrimer = new XMLService(xmlSe

    ObjectName spon = new ObjectName("todd:id=S
       String sessionXMLFile = "jnjmx.ch4/Sessi

       // using the XMLService MBean created by
       // ModelMBeanInfo
      ModelMBean sessionMMBean = (ModelMBean) m
         xmlServiceName,
         "createMBeanFromXMLFile",
         new Object[] {sessionXMLFile,
               "SessionPool", (Object) this},
         new String[] { "java.lang.String",
               "java.lang.String",
               "java.lang.Object"});
       ObjectInstance registeredSessionMMBean =
         mbs.registerMBean((Object) sessionMMBe
    }
    catch (Exception e) {
     System.out.println("SessionPool MBean crea
       + e.getMessage());
    }
  }
// The rest of the methods are the same as in t
// of SessionPool
}
```

4.8.2.2 Creating Model MBeans by an External Force: Assigning a Managed Resource

An application can create model MBeans to manage a completely separate resource. The program must have a reference to the managed resource for the model MBean that it can set in the `managedResource` attribute. The program is also responsible for inspecting the managed resource or another configuration file and creating the attributes, operations, and notifications. A program may instantiate model MBeans for lots of managed resources in the system. Usually this is done when a system is initialized or a management system is initialized. Another common scenario is that this program is a connector between an existing proprietary management system and a JMX-based management system. The Apache manager examples at the beginning of this chapter are excellent examples of a program creating model MBeans for external managed resources.

4.8.3 Adding Custom Descriptors

Applications using model MBeans and adapter MBeans can add their own descriptor fields to the model MBean descriptor. When would this be useful?

Applications using model MBeans could add their own management metadata that was unique to their own custom, specific management system, such as display information or roles. One interesting bit of metadata that we suggest all developers should consider adding to their attribute descriptors is an attribute classification, much like the operation role. Attributes should be classified as descriptive, configuration, capability, state, or metric. This classification of attributes is consistent with the one defined by the CIM Model.[21]

Taking the earlier `TotalAccesses` attribute as an example, we

classify this attribute as a metric:

```
// Custom descriptor example
Descriptor totalAccessesCustomDesc = new Descri
totalAccessesCustomDesc.setField("name","TotalA
totalAccessesCustomDesc.setField("descriptorTyp
totalAccessesCustomDesc.setField("displayName",
totalAccessesCustomDesc.setField("getMethod","g
totalAccessesCustomDesc.setField("setMethod","s
totalAccessesCustomDesc.setField("classificatio
// custom descriptor
```

Adapter MBeans that translate model MBeans to represent resources into their own management systems may cache hints to help with this translation in the descriptor. For example, they may add the SNMP OID to the descriptor or the CIM name for the resource. Or they may add a business domain (e.g., for Accounting Department) or application domain (e.g., for WebSphere) to assist with context and scope.

4.8.4 Overriding the RequiredModelMBean Class

Because the `RequiredModelMBean` class is an implementation of the model MBean that is provided with every JMX implementation, it is reasonably safe to extend the `RequiredModelMBean` class and add or override behavior. For example, if you wanted to persist your `RequiredModelMBean` data to an XML file database rather than to a serialized object file, you would need to override the `load()` and `store()` methods of `RequiredModelMBean`. Here is an example of how you could do this:

```
package jnjmx.ch4;
import javax.management.*;
import javax.management.modelmbean.*;
```

```
public class XMLModelMBean extends RequiredMode


/**
   * Constructor for XMLModelMBean
   * @throws MBeanException
   * @throws RuntimeOperationsException
   */

  public XMLModelMBean() throws MBeanException,
    RuntimeOperationsException {
      super();
  }

  public XMLModelMBean(String filename, String
    throws MBeanException, RuntimeOperationsExc
    super();
    try {
      load(filename, mbeanName);

    } catch (Exception e) {
      // didn't find the file with primer data;
    }
  }


  /**
   * Constructor for XMLModelMBean
   * @param arg0
   * @throws MBeanException
   * @throws RuntimeOperationsException
   */
```

```
public XMLModelMBean(ModelMBeanInfo arg0)
  throws MBeanException, RuntimeOperationsExc
  super(arg0);
  try {
    load();
  } catch (Exception e) {
    // didn't find the file with primer data;
  }
}

/**
 * @see javax.management.DynamicMBean#getAttr
 */

public Object getAttribute(String arg0)
  throws AttributeNotFoundException, MBeanExc
    ReflectionException {
      return super.getAttribute(arg0);
}


/**
 * @see javax.management.DynamicMBean#getAttr
 */


public AttributeList getAttributes(String[] a
  return super.getAttributes(arg0);
}

/**
 * @see javax.management.DynamicMBean#invoke(
```

```
 * String[])
 */

public Object invoke(String arg0, Object[] ar
  throws MBeanException, ReflectionException
  return super.invoke(arg0, arg1, arg2);
}

/**
 * @see javax.management.DynamicMBean#setAttr
 */
public AttributeList setAttributes(AttributeL
  return super.setAttributes(arg0);
}

/**
 * This version of load() expects a file name
 * in the file in the format required by the
 */

public void load(String filename, String name
  throws MBeanException, RuntimeOperationsExc
  InstanceNotFoundException
{
  try
  { // Look for file of that name
    XMLService XMLPrimer = new XMLService();
    ModelMBeanInfo newInfo = (ModelMBeanInfo)
      XMLPrimer.createMBeanInfoFromXMLFile(fi
    setModelMBeanInfo(newInfo);
  } catch (Exception e)
  {
    System.out.println(
```

```
        "load(): Persistent MBean XML file was
    }
  }

  public void load()
  throws MBeanException, RuntimeOperationsExcep
     InstanceNotFoundException
  {
    Descriptor mmbDesc = ((ModelMBeanInfo)
      getMBeanInfo()).getMBeanDescriptor();

    // We will use persistLocation to be the di
    // XML file
    // We will use persistName to be the file t
    // for the MBean to
    String persistDir = (String)
      mmbDesc.getFieldValue("persistLocation");
    String persistFile = (String)
      mmbDesc.getFieldValue("persistName");

    String currName = (String) mmbDesc.getField

    if ((persistDir == null) || (persistFile ==
    { // Persistence not supported for this MBe
      // without error
      return;
    }

    String loadTarget = persistDir.concat("/" +
    load(loadTarget, currName);
  }

public void store()
```

```
throws MBeanException, RuntimeOperationsExcep
   InstanceNotFoundException
{

  /* Store in a file with name = MBean name i
  /* For this example, we always persist the
  * override of this method, you need to hono
  * policy descriptors:  */

  /* persist policy:
     persist if persistPolicy != never ||
     persistPolicy == always ||
     persistPolicy == onTimer && now > lastPe
       persistPeriod ||
     persistPolicy == NoMoreOftenThan && now
       persistPeriod
     don't persist if persistPolicy == never
     persistPolicy == onUpdate ||
     persistPolicy = onTimer && now < lastPer
       persistPeriod
  */

  /* You should also remember to */
  /* set the lastPersistTime on attribute */

  /* Check to see if should be persisted */

  boolean MBeanPersistItNow = true; // always
  if (getMBeanInfo() == null)
  {
    throw new RuntimeOperationsException(new
      IllegalArgumentException("ModelMBeanInf
          "null"),
```

```
            ("Exception occured trying to set the s
             "the RequiredModelMBean"));
        }
        Descriptor mmbDesc = ((ModelMBeanInfo)
          getMBeanInfo()).getMBeanDescriptor();

        if (MBeanPersistItNow == true)
        {
          try
          {
            // Get directory
            String persistDir = (String)
              mmbDesc.getFieldValue("persistLocatio
            String persistFile = (String)
              mmbDesc.getFieldValue("persistName");
            String persistTarget = persistDir.conca
              persistFile);
            // Call another method in this app to w
            // database record
            XMLService XMLPrimer = new XMLService()
            XMLPrimer.writeXMLFile(this, persistTar
          } catch (Exception e)
          {
            System.out.println("store(): Exception
                "into file for RequiredModelMBean "
                e.getClass() + ":" + e.getMessage()
            e.printStackTrace();
          }
        }
}
}
```

If you can put the XMLModelMBean class in the class path of

the MBeanServer, then when you instantiate or create the new model MBean, it will look like this (snipped from the earlier integrated model MBean example):

```
...
  public XMLServer(MBeanServer mbs) throws Exce
    this.mbs = mbs;

    connectionQueue = new TreeSet();
    connections = 0;

    sessions = new SessionPool();
    ObjectName spon = new ObjectName("todd:id=S

    XMLModelMBean sessionMMBean = (XMLModelMBea
      mbs.instantiate("jnjmx.ch4.XMLModelMBean"
      new Object [] {"jnjmx.ch4/SessionInfo.xml
      new String[] {"java.lang.String", "java.l
      sessionMMBean.store();
    sessionMMBean.setManagedResource(sessions,
    ObjectInstance registeredSessionMMBean =
    mbs.registerMBean((Object) sessionMMBean,sp

    active = true;
    tzero = System.currentTimeMillis();
  }
...
```

Or you can use the create operation:

```
public XMLServer(MBeanServer mbs) throws Except
  this.mbs = mbs;

  connectionQueue = new TreeSet();
```

```
  connections = 0;

  sessions = new SessionPool();
  ObjectName spon = new ObjectName("todd:id=Ses

  XMLModelMBean sessionMMBean = (XMLModelMBean)
    mbs.createMBean("jnjmx.ch4.XMLModelMBean",

        new Object [] {"jnjmx.ch4/SessionInfo.x
        new String[] {"java.lang.String", "java
  sessionMMBean.setManagedResource(sessions, "O
  ObjectInstance registeredSessionMMBean =
     mbs.registerMBean((Object) sessionMMBean,s

  active = true;
  tzero = System.currentTimeMillis();
}
```

If you can't update the class path of the MBeanServer, you will have to instantiate `XMLModelMBean` with a `new` operation in the scope of your application and then register it with the MBeanServer like this:

```
XMLModelMBean myMMB = new XMLModelMBean();
mbs.register(myMMB);
```

4.9 Common Mistakes with Model MBeans

1. **Trying to set descriptors used by the model MBean code. The model MBean implementation uses some of the descriptor values to manage the logic of value caching. You should never manually set the following descriptor fields:**

    - **Attribute descriptor:** `lastUpdatedTimeStamp`
    - **Operation descriptor:** `lastReturnedValue`, `lastUpdatedTimeStamp`

    Note that the `value` field of the attribute descriptor is where the cached value of the attribute is maintained by the model MBean. If you set the `value` field and the `getMethod` field, then the value in the `value` field will be replaced according to the caching policy. You must turn caching off entirely to retain the value you place there. It is perfectly valid to put an initial value there, but it might be more appropriate to put that value in the `defaultValue` field.

- **Getting the attribute type wrong**. The attribute type fields must contain the *fully* qualified class name, not the relative class name.

- **Class loader problems**. Remember that if you are using the `instantiate()` or `create()` method to create the model MBean, then the class loader of the MBeanServer is being used.

4.10 Caveats

4.10.1 Transactionality

If a model MBean implementation is executing in an environment where management operations are transactional, then the model MBean should shield the application from this knowledge. If the application must be aware of the transaction, the application will depend on a certain version of the JMX agent and model MBean to be accessible. The application's investment in the JMX instrumentation is no longer portable and protected.

4.10.2 Remoteness

If the JMX agent is remotable, then the application and/or adapters may be accessing model MBeans that are not co-residing in the same JVM. The model MBean and JMX agent must be implemented so that the applications and adapters are not aware that the model MBean is local or remote. In other words, the model MBean must support location transparency. The JMX agent does not have to provide for remotabilty for *both* the applications and the adapters; it may be remotable to just one of these.

4.11 Summary

In this chapter we have thoroughly examined the model MBean specification and examples to illustrate the use of the `RequiredModelMBean` class. You should now understand the role that descriptors play in model MBeans, their impact on model MBean behavior, and how to customize and augment them. We have also provided you with some common patterns for using model MBeans: loading from XML files, static model MBeans, and dynamic model MBeans. We also reviewed choices in the model MBean lifecycle: being created by the managed resource or by an outside program. More patterns and usage tips are provided in Chapter 9 (Designing with JMX).

Now that we understand every variation of management beans —the components that represent all of our management data— the next chapter will explain the container for these MBeans, the MBeanServer. The MBeanServer functions as a management agent role in your management architecture. Then we will turn our attention to security and how to use JMX to do the actual distributed management: monitoring and notification.

4.12 XML File Example

File: apachembeans.xml

```
<?xml version="1.0"?>
<!DOCTYPE mbeans-descriptors PUBLIC
     "-//Apache Software Foundation//DTD Model
     "http://jakarta.apache.org/commons/dtds/mb
<mbeans-descriptors>
<mbean
     name="ApacheServer"
     description="ModelMBean for managing an Ap
     type="jnjmx.ch4.ApacheServer"
     >
<attribute
     name="BusyWorkers"
     description="Apache Server Busy Workers"
     type="int"
     readable="true"
     writeable="false"
     is="false"
     getMethod="getBusyWorkers"
     displayName="Apache BusyWorkers"
     >
     <descriptor>
         <field name="displayName" value="Apach
         <field name="currencyTimeLimit" value=
         <field name="name" value="BusyWorkers"
         <field name="getMethod" value="getBusy
         <field name="descriptorType" value="at
     </descriptor>
</attribute>
```

```
<attribute
     name="BytesPerSec"
     description="Apache Server Bytes Per Sec"
     type="int"
     readable="true"
     writeable="false"
     is="false"
     getMethod="getBytesPerSec"
     displayName="Apache BytesPerSec"
     >
     <descriptor>
         <field name="displayName" value="Apach
         <field name="currencyTimeLimit" value=
         <field name="name" value="BytesPerSec"
         <field name="getMethod" value="getByte
         <field name="descriptorType" value="at
     </descriptor>
</attribute>
<attribute
     name="BytesPerReq"
     description="Apache Server Bytes Per Reque
     type="int"
     readable="true"
     writeable="false"
     is="false"
     getMethod="getBytesPerReq"
     displayName="Apache BytesPerReq"
     >
     <descriptor>
         <field name="displayName" value="Apach
         <field name="currencyTimeLimit" value=
         <field name="name" value="BytesPerReq"
         <field name="getMethod" value="getByte
```

```
            <field name="descriptorType" value="at
        </descriptor>
    </attribute>
    <attribute
        name="IdleWorkers"
        description="Apache Server Idle Workers"
        type="int"
        readable="true"
        writeable="false"
        is="false"
        getMethod="getIdleWorkers"
        displayName="Apache IdleWorkers"
        >
        <descriptor>
            <field name="displayName" value="Apach
            <field name="currencyTimeLimit" value=
            <field name="name" value="IdleWorkers"
            <field name="getMethod" value="getIdle
            <field name="descriptorType" value="at
        </descriptor>
    </attribute>
    <attribute
        name="ReqPerSec"
        description="Apache Server Requests Per Se
        type="float"
        readable="true"
        writeable="false"
        is="false"
        getMethod="getReqPerSec"
        displayName="Apache ReqPerSec"
        >
        <descriptor>
            <field name="displayName" value="Apach
```

```
            <field name="currencyTimeLimit" value=
            <field name="name" value="ReqPerSec"/>
            <field name="getMethod" value="getReqP
            <field name="descriptorType" value="at
        </descriptor>
    </attribute>
    <attribute
        name="Scoreboard"
        description="Apache Server Scoreboard"
        type="java.lang.String"
        readable="true"
        writeable="false"
        is="false"
        getMethod="getScoreboard"
        displayName="Apache Scoreboard"
        >
        <descriptor>
            <field name="displayName" value="Apach
            <field name="currencyTimeLimit" value=
            <field name="name" value="Scoreboard"/
            <field name="getMethod" value="getScor
            <field name="descriptorType" value="at
        </descriptor>
    </attribute>
    <attribute
        name="TotalAccesses"
        description="Apache Server total accesses"
        type="int"
        readable="true"
        writeable="false"
        is="false"
        getMethod="getTotalAccesses"
        displayName="Apache TotalAccesses"
```

```
        >
        <descriptor>
            <field name="displayName" value="Apach
            <field name="currencyTimeLimit" value=
            <field name="name" value="TotalAccesse
            <field name="getMethod" value="getTota
            <field name="descriptorType" value="at
        </descriptor>
</attribute>
<attribute
        name="TotalKBytes"
        description="Apache Server total KiloBytes
        type="long"
        readable="true"
        writeable="false"
        is="false"
        getMethod="getTotalKBytes"
        displayName="Apache TotalKBytes"
        >
        <descriptor>
            <field name="displayName" value="Apach
            <field name="currencyTimeLimit" value=
            <field name="name" value="TotalKBytes"
            <field name="getMethod" value="getTota
            <field name="descriptorType" value="at
        </descriptor>
</attribute>
<attribute
        name="Uptime"
        description="Apache Server Up Time"
        type="long"
        readable="true"
        writeable="false"
```

```
      is="false"
      getMethod="getUptime"
      displayName="Apache Uptime"
      >
      <descriptor>
          <field name="displayName" value="Apach
          <field name="currencyTimeLimit" value=
          <field name="name" value="Uptime"/>
          <field name="getMethod" value="getUpti
          <field name="descriptorType" value="at
      </descriptor>
</attribute>
<attribute
      name="CpuLoad"
      description="Apache Server CPU Load"
      type="float"
      readable="true"
      writeable="false"
      is="false"
      getMethod="getCpuLoad"
      displayName="Apache Uptime"
      >
      <descriptor>
          <field name="displayName" value="Apach
          <field name="currencyTimeLimit" value=
          <field name="name" value="CpuLoad"/>
          <field name="getMethod" value="getCpuL
          <field name="descriptorType" value="at
      </descriptor>
</attribute>
<attribute
      name="State"
      description="Apache Server state"
```

```
        type="java.lang.String"
        readable="true"
        writeable="false"
        is="false"
        getMethod="getState"
        displayName="Apache State"
        >
        <descriptor>
            <field name="displayName" value="Apach
            <field name="currencyTimeLimit" value=
            <field name="name" value="State"/>
            <field name="getMethod" value="getStat
            <field name="descriptorType" value="at
        </descriptor>
    </attribute>
    <attribute
        name="Server"
        description="Apache Server Busy Workers"
        type="java.lang.String"
        readable="true"
        writeable="true"
        is="false"
        getMethod="getServer"
        setMethod="setServer"
        displayName="Apache Server URL"
        >
        <descriptor>
            <field name="setMethod" value="setServ
            <field name="displayName" value="Apach
            <field name="currencyTimeLimit" value=
            <field name="name" value="Server"/>
            <field name="getMethod" value="getServ
            <field name="descriptorType" value="at
```

```
     </descriptor>
</attribute>
<constructor
     name="jnjmx.ch4.ApacheServer"
     >
     <parameter name="apacheURL"
         description="URL of Apache Server to M
         type="java.lang.String"/>
     <descriptor>
         <field name="name" value="jnjmx.ch4.Ap
         <field name="descriptorType" value="op
         <field name="role" value="constructor"
     </descriptor>
</constructor>

<notification
     name="jmx.ModelMBean.General"
     description="Apache Server Down"
     >
     <notification-type>
     jmx.ModelMBean.General.Apache.Down
     </notification-type>
     <descriptor>
         <field name="name" value="jmx.ModelMBe
         <field name="severity" value="6"/>
         <field name="descriptorType" value="no
     </descriptor>
</notification>

<operation
     name="getValue"
     description="getValue(): get an apache sta
     returnType="java.lang.String"
```

```
      impact="2"
      >
      <parameter name="FieldName"
          description="Apache status field name"
          type="java.lang.String"/>
      <descriptor>
          <field name="name" value="getValue"/>

          <field name="class" value="ApacheServe
          <field name="descriptorType" value="op
          <field name="role" value="operation"/>
      </descriptor>
</operation>
<operation
      name="getState"
      description="getState(): current status of
      returnType="java.lang.String"
      impact="2"
      >
      <descriptor>
          <field name="name" value="getState"/>
          <field name="class" value="ApacheServe
          <field name="descriptorType" value="op
          <field name="role" value="operation"/>
      </descriptor>
</operation>
<operation
      name="getServer"
      description="getServer(): URL of apache se
      returnType="java.lang.Integer"
      impact="2"
      >
      <descriptor>
```

```
            <field name="name" value="getServer"/>
            <field name="class" value="ApacheServe
            <field name="descriptorType" value="op
            <field name="role" value="operation"/>
        </descriptor>
</operation>
<operation
        name="setServer"
        description="getServer(): URL of apache se
        returnType="java.lang.String"
        impact="1"
        >
        <parameter name="url" description="Apache
            type="java.lang.String"/>
        <descriptor>
            <field name="name" value="setServer"/>
            <field name="descriptorType" value="op
            <field name="class" value="ApacheServe
            <field name="role" value="operation"/>
        </descriptor>
</operation>

<operation
        name="start"
        description="start(): start apache server"
        returnType="java.lang.Integer"
        impact="1"
        >
        <descriptor>
            <field name="name" value="start"/>
            <field name="class" value="ApacheServe
            <field name="descriptorType" value="op
            <field name="role" value="operation"/>
```

```
        </descriptor>
    </operation>
    <operation
        name="stop"
        description="stop(): start apache server"
        returnType="java.lang.Integer"
        impact="1"
        >
        <descriptor>
            <field name="name" value="stop"/>
            <field name="class" value="ApacheServe
            <field name="descriptorType" value="op
            <field name="role" value="operation"/>
        </descriptor>
    </operation>
    <operation
        name="getBusyWorkers"
        description="number of busy threads"
        returnType="int"
        impact="2"
        >
        <descriptor>
            <field name="name" value="getBusyWorke
            <field name="class" value="ApacheServe
            <field name="descriptorType" value="op
            <field name="role" value="operation"/>
        </descriptor>
    </operation>
    <operation
        name="getBytesPerSec"
        description="number of bytes per second pr
        returnType="int"
        impact="2"
```

```
      >
      <descriptor>
          <field name="name" value="getBytesPerS
          <field name="class" value="ApacheServe
          <field name="descriptorType" value="op
          <field name="role" value="operation"/>
      </descriptor>
</operation>
<operation
      name="getBytesPerReq"
      description="number of bytes per request p
      returnType="int"
      impact="2"
      >
      <descriptor>
          <field name="name" value="getBytesPerR
          <field name="class" value="ApacheServe
          <field name="descriptorType" value="op
          <field name="role" value="operation"/>
      </descriptor>
</operation>
<operation
      name="getCpuLoad"
      description="current load of cpu"
      returnType="float"
      impact="2"
      >
      <descriptor>
          <field name="name" value="getCpuLoad"/
          <field name="class" value="ApacheServe
          <field name="descriptorType" value="op
          <field name="role" value="operation"/>
      </descriptor>
```

```
</operation>
<operation
     name="getIdleWorkers"
     description="number of idle threads"
     returnType="int"
     impact="2"
     >
     <descriptor>
         <field name="name" value="getIdleWorke
         <field name="class" value="ApacheServe
         <field name="descriptorType" value="op
         <field name="role" value="operation"/>
     </descriptor>
</operation>
<operation
     name="getReqPerSec"
     description="number of bytes per second pr
     returnType="int"
     impact="2"
     >
     <descriptor>
         <field name="name" value="getReqPerSec
         <field name="descriptorType" value="op
          <field name="class" value="ApacheServ
         <field name="role" value="operation"/>
     </descriptor>
</operation>
<operation
     name="getScoreboard"
     description="gets apache scoreboard "
     returnType="java.lang.String"
     impact="2"
     >
```

```
        <descriptor>
            <field name="name" value="getScoreboar
            <field name="class" value="ApacheServe
            <field name="descriptorType" value="op
            <field name="role" value="operation"/>
        </descriptor>
</operation>
<operation
        name="getTotalAccesses"
        description="number of bytes per second pr
        returnType="int"
        impact="2"
        >
        <descriptor>
            <field name="name" value="getTotalAcce
            <field name="class" value="ApacheServe
            <field name="descriptorType" value="op
            <field name="role" value="operation"/>
        </descriptor>
</operation>
<operation
        name="getTotalKBytes"
        description="number of Kilo bytes processe
        returnType="long"
        impact="2"
        >

        <descriptor>
            <field name="name" value="getTotalKByt
            <field name="class" value="ApacheServe
            <field name="descriptorType" value="op
            <field name="role" value="operation"/>
        </descriptor>
```

```
</operation>
<operation
     name="getUptime"
     description="number of bytes per second pr
     returnType="long"
     impact="2"
     >
     <descriptor>
         <field name="name" value="getUptime"/>
         <field name="class" value="ApacheServe
         <field name="descriptorType" value="op
         <field name="role" value="operation"/>
     </descriptor>
</operation>
<operation
     name="getCpuLoad"
     description="number CPU Load"
     returnType="float"
     impact="2"
     >
     <descriptor>
         <field name="name" value="getCpuLoad"/
         <field name="class" value="ApacheServe
         <field name="descriptorType" value="op
         <field name="role" value="operation"/>
     </descriptor>
</operation>
<descriptor>
  <field name="descriptorType" value="mbean"/>
  <field name="logfile" value="jmxmain.log"/>
  <field name="name" value="ApacheServer"/>
  <field name="visibility" value="1"/>
  <field name="log" value="true"/>
```

```
  <field name="currencyTimeLimit" value="10"/>
  <field name="type" value="jnjmx.ch4.Apacheerv
  <field name="export" value="false"/>
  <field name="displayName" value="ApacheServer
  <field name="persistPolicy" value="Never"/>
</descriptor>
```

Notes

1. "Java Management Extensions (JMX) Specification," JSR 3, http://www.jcp.org/jsr/detail/3.jsp, which was led by Sun Microsystems to create a management API for Java resources.

2. XML stands for eXtensible Markup Language. XML standards are developed by the Internet Engineering Task Force (IETF, at http://www.ietf.org) and the World Wide Web Consortium (W3C, at http://www.w3.org).

3. J2SE stands for Java 2 Platform, Standard Edition, which is Sun Microsystems' Java platform. More information is available at http://java.sun.com/j2se. Java and all Java-based marks are trademarks of Sun Microsystems, Inc., in the United States and other countries.

4. J2EE stands for Java 2 Platform, Enterprise Edition, which is Sun Microsystems' Java platform. J2EE application servers are vendor products that support the J2EE specification. More information is available at http://java.sun.com/j2ee. Java and all Java-based marks are trademarks of Sun Microsystems, Inc., in the United States and other countries.

5. J2ME stands for Java 2 Platform, Micro Edition, from Sun Microsystems. More information is available at http://java.sun.com/j2me. Java and all Java-based marks are trademarks of Sun Microsystems, Inc., in the United States and other countries.

6. JDBC (Java Database Connectivity) is an API that isolates database clients from database vendors. It is a Sun Microsystems technology.

7. **EJB stands for Enterprise JavaBeans, which is a component model core to the J2EE specification. More information is available at http://java.sun.com/j2ee.**

8. **LDAP stands for Lightweight Directory Access Protocol, a protocol designed to provide access to the X.500 directory while not incurring the resource requirements of the Directory Access Protocol (DAP). More information is available in RFC 1777 (http://www.ietf.org/rfc/rfc1777.txt).**

9. **DB2 stands for Database 2, a relational database management system designed by IBM for large computers. More information is available http://www-3.ibm.com/software/data/db2.**

10. **NFS stands for Network File System, which allows a remote user to read and update a computer file system (see http://searchwin2000.techtarget.com/sDefinition/0,, NFS version 4 is currently under development by the IETF (see http://www.ietf.org/html.charters/nfsv4-charter.html).**

11. **IDL stands for Interface Definition Language, which defined a language agnostic mechanism to define the interface of a CORBA component. More information is available at http://www.omg.org.**

12. **MOF stands for Managed Object Format. This format is used to describe CIM information and is defined by the DMTF (Distributed Management Task Force) in the CIM specification. More information is available at http://www.dmtf.org/standards/cim_schema_v23.ph**

13. **JMX Instrumentation and Agent Specification 1.1, a maintenance release of the JMX specification, reference implementation, and TCK by Sun**

Microsystems. See note 1.

14. IETF stands for Internet Engineering Task Force (http://www.ietf.org).

15. IETF Web Server MIB, an SNMP MIB defined to help manage and represent Web servers (HTTP servers). More information is available in IETF RFC 2594 (http://www.ietf.org/rfc/rfc2594.txt).

16. SNMP stands for Simple Network Management Protocol, which is an IETF standard. More information on SNMP is available at http://www.ietf.org.

17. HTTP MIB, an SNMP MIB defined to help manage and represent Web servers (HTTP servers). More information is available in IETF RFC 2594 (http://www.ietf.org/rfc/rfc2594.txt).

18. CIM object property, a property or attribute of a CIM class or object, as defined by the DTMF CIM model specification (http://www.dmtf.org/standards/cim_schema_v23.p

19. The CIM DTD is part of the CIM/WBEM standard (Common Information Model/Web-Based Enterprise Management). It is defined by the DMTF. More information is available at http://www.dmtf.org/download/spec/xmls/CIM_XM

20. Common Information Model Schema version 2.6 available from http://www.dmtf.org.

21. Jakarta Commons Modeler is an open-source tool to instantiate JMX model MBeans from XML files. More information is available at http://jakarta.apache.org/commons/modeler.html.

Chapter 3. All about MBeans

MBeans are the raw material of JMX-based management. Management systems monitor and control resources: a router, a server, an EJB, or a JVM. To perform their monitoring and control duties, management systems need information about the resources they are responsible for and a means of affecting the operation of those resources. In JMX, MBeans provide the information about, and operations on, resources that management systems require.

The instrumentation layer of the JMX architecture is all about MBeans: standard MBeans, dynamic MBeans, open MBeans, and model MBeans. This chapter describes standard and dynamic MBeans in detail and gives an overview of the principles and interfaces of open MBeans. Model MBeans are the topic of Chapter 4. Here we define what MBeans are, discuss the various aspects of their design, and describe their implementation. Throughout we pay special attention to the management interface that MBeans expose to the rest of the management system.

Chapter 1. Management Concepts

The term *management* is grossly overused in the computer industry. It is often the responsibility of a *management* system to take care of installation and configuration of a resource into a computer system. After a resource is installed, the *management* system should be able to start it, monitor it to be sure it is performing well, change its behavior through reconfiguration or operations, and stop it. The first thing this book is going to do is define management and the terms to describe management, and then identify the aspects of management that are addressed by Java Management Extensions (JMX)[1].

This chapter gives a short history of management systems and an overview of management architectures and technologies. Building on this basis, we will discuss the management lifecycle, resulting management disciplines, management data and operations, and then how all of these things are combined into management applications and systems.

Most of this chapter is not specific to JMX, and if you have a background in management and management technologies, you may choose to move on to Chapter 2 (Introduction to JMX). If you are new to management, having a basic understanding of management technology will help you understand the terminology used in this book, in the JMX specification, and by management system vendors. It will also give you the background you may need to develop or integrate with a management system.

Resources use JMX to make themselves manageable by a management system. JMX is an isolation and mediation layer between manageable resources and management systems. Why do we need this decoupling of the manager and the managed? A quick look at the history of management systems and technologies will make clear how we came to need JMX for the Java platform.

Chapter 9. Designing with JMX

The authors of this book have approached JMX from different points of view, with different goals, with different skills. These differences have yielded many opportunities to be exposed to a wide variety of applications, problems, and solutions. This chapter looks at how and where to deploy MBeanServers, what options are available to you when you're designing MBeans for your Java resources, and who should create and register your MBeans and when. We also describe and illustrate some of the best practices we've picked up in the course of applying JMX to management problems, including MBean granularity, application managers, management models, and federation.

Notes

1. Source: "Java Management Extensions (JMX) Specification," JSR 3, http://www.jcp.org/jsr/detail/3.jsp, which was led by Sun Microsystems to create a management API for Java resources.

2. 3270 is the model of a line of user terminals made by IBM Corporation, Armonk, NY (http://www.ibm.com).

3. IBM Corporation, Armonk, NY (http://www.ibm.com).

4. Candle Corporation, 201 N. Douglas St., El Segundo, CA 90245 (http://www.candle.com).

5. Computer Associates International, Inc., One Computer Associates Plaza, Islandia, NY 11749 (http://www.cai.com).

6. For information about UNIX, see The Open Group's site at http://www.opengroup.org.

7. Solaris is Sun Microsystems' UNIX-based operating system. More information is available at http://www.sun.com/solaris.

8. Hewlett-Packard's UNIX-based operating system is called HP-UX. More information is available at http://www.hp.com/products1/unix/operating.

9. TCP/IP (Transmission Control Protocol/Internet Protocol) is an Internet protocol that has been defined as a standard by the Internet Engineering Task Force. Additional information is available at

http://www.ietf.org.

10. Tivoli Systems Inc., 9442 Capital of Texas Highway North, Arboretum Plaza One, Austin, TX 78759 (http://www.tivoli.com). Tivoli is a trademark of Tivoli Systems in the United States, other countries, or both.

11. BMC Software, Inc., 2101 City West Blvd., Houston, TX 77042-2827 (http://www.bmc.com).

12. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries.

13. A daemon on a system is a long-running system process that is executing a program.

14. SNMP stands for Simple Network Management Protocol, which is an IETF standard. More information on SNMP is available at http://www.ietf.org.

15. CMIP stands for Common Management Information Protocol and is usually referred to in conjunction with CMIS (Common Management Information Services). This management standard was defined by OSI (Open Systems Interconnection) as an ISO (International Standards Organization) standard: ISO 9595/2 and 9596/2 (http://www.iso.ch). More information on CMIP/CMIS can be found at http://www.iso.ch.

16. CIM/WBEM stands for Common Information Model/Web-Based Enterprise Management. It is defined by the Distributed Management Task Force (DMTF). More information is available at http://www.dmtf.org.

17. SMUX stands for SNMP Multiplexing Protocol (IETF

RFC 1227, http://www.ietf.org/rfc/rfc1227.txt?number=1227) and is used to allow an application to communicate with an SNMP agent for the purposes of satisfying a portion of the MIB. SMUX tends to predominate in UNIX systems.

18. The Distributed Program Interface (DPI) is designed for extending SNMP agents. It is predominant in IBM systems. Source: G. Carpenter and B. Wijnen, "SNMP-DPI: Simple Network Management Protocol Distributed Program Interface," RFC 1228 (May 1991), http://www.ietf.org/rfc/rfc1228.txt?number=1228.

19. AgentX is a standard SNMP agent-to-subagent protocol, "Extensible SNMP Agent" specification from the IETF. Source: M. Daniele, B. Wijnen, M. Ellison, and D. Franciso (Eds.), "Agent Extensibility (AgentX) Protocol Version 1," RFC 2741 (January 2000), http://www.ietf.org/rfc/rfc2741.txt?number2741.

20. Information on IBM's WebSphere Administration Console is available at http://www.ibm.com/software/webservers/appserv. WebSphere is a trademark of IBM in the United States.

21. Information on IBM's System/390 series is available at http://www.s390.ibm.com. System/390 is a registered trademark of IBM in the United States.

22. Systems Management Server (SMS) is Microsoft's workstation management application. More information is available at http://www.microsoft.com/smsmgmt/default.asp?RLD=263. Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or

both.

23. See note 14.

24. More information is available at http://www.ietf.org.

25. Source: M. Rose and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-Based Internets," RFC 1065 (August 1988), http://www.ietf.org/rfc/rfc1065.txt?number=1065.

26. Source: M. Rose and K. McCloghrie, "Management Information Base for Network Management of TCP/IP-Based Internets," RFC 1066 (August 1988), http://www.ietf.org/rfc/rfc1066.txt?number=1066.

27. More information about Hewlett-Packard's OpenView is available at http://www.openview.hp.com?qt=OpenView, or from Hewlett-Packard, 3000 Hanover St., Palo Alto, CA 94304-1185.

28. Information about Tivoli NetView is available at http://www.tivoli.com/products/index/netview.

29. See note 15.

30. The URL for the International Standards Organization (ISO) is http://www.iso.ch.

31. OSI/TMN stands for Open System Interconnection/Telecommunication Management Network. OSI was defined as an ISO (International Standards Organization) standard: ISO ISO/IEC DIS 10165-1,2,3,4, 9595/2 and 9596/2 (http://www.iso.org).

32. Groupe Bull was the original company name for the representative. This company is now referred to as "Evidian, A Groupe Bull Company"

(http://www.evidian.com).

33. The Distributed Management Task Force (DMTF) is a standards body that is responsible for DMI, CIM, and WBEM management standards. More information is available at http://www.dmtf.org.

34. See note 16.

35. See note 22.

36. WBEMsource, at http://www.wbemsource.org, is a group of open-source WBEM implementations sponsored by The Open Group. Implementations are available in Java and C++.

37. Pegasus is The Open Group's C++ implementation of the DMTF's WBEM and CIM operations specifications, and a CIMOM, for accessing CIM data through HTTP (http://www.opengroup.org/management, http://www.openpegasus.org).

38. JSR 48, "WBEM Services Specification," is Sun's Java implementation of the DMTF's WBEM and CIM operations specifications for accessing CIM data through HTTP. See http://www.jcp.org/jsr/detail/48.jsp.

39. JDMK stands for Java Dynamic Management Kit. JDMK 4.0 is available from Sun Microsystems, Inc., 901 San Antonio Rd., Palo Alto, CA 94303, http://java.sun.com/products/jdmk.

40. TMX4J is an implementation of the JMX specifiation from Tivoli Systems and is available for free on IBM's alphaWorks Web site: http://alphaworks.ibm.com.

41. Information about AIC is available at http://www.opengroup.org/management/aic.htm.

12. The Application Response Measurement (ARM) standard was developed by The Open Group, http://www.opengroup.org/management/arm.htm.

13. You can find more information about IBM WebSphere 4.0 at http://www.ibm.com/websphere. WebSphere is a trademark of IBM in the United States.

14. Tivoli's Application Management Specification is used to define the characteristics of a managed application.

15. MOF file stands for Managed Object Format file. This format is used to describe CIM information and is defined by the DMTF in the CIM specification. More information is available at http://www.dmtf.org/standards/cim_schema_v23.ph

16. Source: J. Farrell and H. Kreger, "Web Services Management Approaches," IBM Systems Journal 41: 2 (March 2002).

17. Marimba, Inc., 440 Clyde Ave., Mountain View, CA 94043. Additional information is available at http://www.marimba.com. Marimba is a trademark of Marimba, Inc., in the United States and other countries.

18. TCP stands for Transmission Control Protocol, UDP for User Datagram Protocol, and ICMP for Internet Control Message Protocol.

19. S. Waldbusser and P. Grillo. "Host Resources MIB," RFC 2790 (March 2000), http://www.ietf.org/rfc/rfc2790.txt?number=2790.

Chapter 2. Introduction to JMX[1]

The Java Management Extensions (JMX) specification[2] defines a Java optional package for J2SE[3] that provides a management architecture and API set that will allow any Java technology–based or accessible resource to be inherently manageable. By using JMX, you can manage Java technology resources. You can also use Java technology and JMX to manage resources that are already managed by other technologies, such as SNMP[4] and CIM/WBEM.[5]

JMX introduces a JavaBeans model for representing the manageability of resources. The core of JMX is the simple, yet sophisticated and extensible, management agent for your Java Virtual Machine (JVM) that can accommodate communication with private or acquired enterprise management systems. JMX also defines a set of services to help manage your resources. JMX is so easy to use and is so suited for the Java development paradigm that it is possible to make an application manageable in three to five lines of code.

Basically, JMX is to management systems what JDBC (Java Database Connectivity)[6] is to databases. JDBC allows applications to access arbitrary databases; JMX allows applications to be managed by arbitrary management systems. JMX is an isolation layer between the applications and arbitrary management systems. So why do we need this layer anyway?