4 MBeans for stable resources

- Understanding common development rules for all MBeans
- Examining rules specific to developing Standard MBeans
- Using the Standard MBean development patterns
- Exploring Standard MBean examples

In this chapter, we will discuss the simplest type of MBean: the Standard MBean. You created this type of MBean in chapter 2. Standard MBeans are intended for resources that have well-known, stable interfaces. This chapter shows how you can use Standard MBeans to configure application resources (a log utility and application properties) and to break applications into components. If you need the quickest way to implement a resource, the Standard MBean is for you. Standard MBeans expose a resource with an explicitly declared management interface that is unchanging.

In addition, because this is the first of three chapters on MBean types, we also discuss some of the common construction rules of all types of MBeans. After completing this chapter, you will know much more about Standard MBeans, including how to write them and when to use them.

4.1 Laying the MBean groundwork

As we just hinted, before diving into a discussion about writing Standard MBeans, we first need to describe certain traits that are required across all MBean types. Whether you are coding a Standard or Dynamic MBean, you must follow certain rules. After covering these common rules, we will explore the unique traits of a Standard MBean.

4.1.1 Common coding rules for all MBeans

When developing any MBean, you must adhere to the following rules:

- An MBean must be a concrete Java class. A *concrete* class is a Java class that is not abstract, and can therefore be instantiated. Remember from chapter 2 that you dynamically loaded the `HelloWorld` MBean into your simple JMX agent using the HTML adapter. For the agent to successfully create the MBean using reflection, the class name you used had to correspond to a concrete class.
- An MBean must have a public constructor. No additional rules apply to the constructor other than that it must be public. It can have arguments—and the class can contain as many constructors as needed.
- An MBean must implement either its own MBean interface or the `javax.management.DynamicMBean` interface. An MBean interface is any interface that follows a naming scheme `ClassNameMBean`. We will cover MBean interfaces thoroughly in this chapter. MBeans using an MBean interface are Standard MBeans.

A Standard MBean is an MBean that implements its own MBean interface. As mentioned earlier, the `HelloWorld` MBean from chapter 2 was a Standard MBean. It was a concrete class, had a public constructor, and implemented an interface named `HelloWorldMBean`.

Enabling notifications

In addition to following the three rules we just listed, all MBeans can optionally implement the `javax.management.NotificationBroadcaster` interface. This interface allows MBeans to send notifications to interested listeners. Notifications are Java objects sent from JMX components to other objects that have registered as notification listeners. MBeans that implement the `NotificationBroadcaster` interface gain methods that allow objects to register with them to receive notifications. The notification delivery mechanism is very similar to the Java event model, and is covered in chapter 6.

4.1.2 Using Standard MBeans

Now that you know the rules that all MBeans must follow, let's more closely examine the Standard MBean. The Standard MBean is an MBean that uses an explicitly declared management interface to interact with a manageable resource. A *management interface* is the set of methods and attributes exposed by an MBean that management applications can use to manage a resource (via an MBean).

Standard MBean attributes are class members exposed for management by the use of *getter* and *setter* methods. Standard MBean operations are the public class methods in addition to the getters and setters. Operations and attributes are discovered by introspection at the JMX agent level.

Once created, the Standard MBean's management interface does not change. In addition, more than any other MBean type, it embodies one of the major benefits of using JMX: it is simple. Standard MBeans should be used when the interface to a managed resource is well defined or unlikely to change.

For example, Standard MBeans are good for resources that are still being developed, because the resource will have a well-known interface and the management interface can be written explicitly. If you plan to use an MBean to expose part of a new application in development, you should use a Standard MBean. The MBean is simple to develop, and you can create it concurrently with your application. In chapter 5, you will learn about writing MBeans that are not as straightforward as the Standard MBean. In the following sections, you will see that the Standard MBean is the simplest way to expose a resource for management.

4.2 Composing the standard management interface

All information about an MBean must be gathered from its management interface. In the previous section, you read that an MBean must implement an MBean interface or the `javax.management.DynamicMBean` interface. Standard MBeans are MBeans that implement a user-developed MBean interface.

An MBean interface is any interface that follows the naming convention `XMBean`, where *X* is some implementing class name (for example, `PrinterMBean`). An MBean interface declares methods that expose the attributes and operations of a manageable resource.

NOTE One item of importance is that your MBean interfaces must be in the same package as your implementation class. For example, if the `PrinterMBean` interface was in the package `jmxbook.ch6` and the `Printer` class was in `jmxbook.ch4`, then you would not have a valid MBean.

Remember that a management interface includes the set of attributes and operations exposed by an MBean, allowing management applications to use the MBean. In addition, a management interface includes an MBean's constructors and notifications. The following section covers the composition of an MBean's management interface.

4.2.1 Components of the management interface

The management interface of an MBean is composed of the four following items:

- Its public constructors
- Its attributes
- Its operations
- Its notifications

The next few sections cover the details of each of the four components of the management interface of an MBean. The description of the management interface pertains to all types of MBeans. MBeans differ in the way the management interface is exposed, but all management interfaces are composed of the same four parts.

Public constructors

As you witnessed in chapter 2 when using the HTML adapter via a web browser, MBeans can be dynamically loaded into JMX agents. Agents do this using any of

the public constructors exposed by the MBean. Constructors are included in the definition of the management interface because a particular constructor could define specific behavior over the life of the MBean object. For instance, one constructor may tell the MBean to log all of its actions, and another may make it silent. Any way of altering the behavior of an MBean is included as part of its management interface. For Standard MBeans, agents must use introspection to discover the public constructors.

Attributes

Attributes are a vital part of the management interface of an MBean. The attributes describe the manageable resource. Remember, a manageable resource is some application or resource exposed for management by an MBean. For instance, an MBean managing a device such as a printer might have attributes for the number of paper trays, job counts, and so forth.

With Standard MBeans, you expose attributes by declaring getter and setter methods. For an attribute `JobCount` of a printer MBean, there would be a method `getJobCount()`. Recall from chapter 2 that getter methods define read access to an attribute, and setter methods define write access. If both a setter and a getter exist, the MBean grants read/write access to that attribute.

Operations

Operations correspond to actions that can be initiated on the manageable resource. For Standard MBeans, exposed operations are simply the remaining operations that are not getters or setters. Staying with the printer example, in an MBean managing a printer, you might find an operation like `cancelPrintJob()`. Operations are methods like any other; they can have multiple parameters and optionally return a value.

Notifications

Notifications allow MBeans to communicate with registered listeners. You encountered them in chapter 2 when you added a notification to the `HelloWorld` example.

In order to emit notifications, an MBean must implement the `javax.management.NotificationBroadcaster` interface. This interface provides methods for sending notifications, as well as methods for other objects to register as listeners on the implementing MBean. We will skip notifications for now, but we cover them in detail in chapter 6.

4.2.2 Example: a printer MBean interface

Now that you understand the four major parts of a management interface, let's look at an example of an MBean interface. Recall that a user-defined MBean interface indicates that an MBean is a Standard MBean.

The following is the MBean interface for an MBean managing a printer. Look through it before reading further, and try to determine the attributes and operations it exposes:

```
public interface PrinterMBean
{
   public int    getPrintJobCount();
   public String getPrinterName();
   public String getPrintQuality();
   public void   setPrintQuality( int value );
   public void   cancelPrintJobs();
   public void   performSelfCheck();
}
```

The `PrinterMBean` interface exposes three attributes and two operations. You can tell this by visually examining the interface. JMX agents use a process called *introspection* to read the interface. Introspection uses Java reflection to examine the MBean interface to determine its attributes and operations. After discovering all the public methods in this interface, the agent uses a small set of rules to determine what the MBean has exposed as part of its management interface.

To find attributes, a JMX agent looks for any method following the `getAttributeName()` or `setAttributeName()` naming scheme. In addition to the `getAttributeName()` pattern, you can optionally use the form `isAttributeName()`, which must return a `boolean` value. However, if an attribute is exposed with a getter method, it cannot also have an *is* method. Setter methods also have a unique rule: they cannot be overloaded. For example, this interface would be invalid if the method `setPrintQuality( String value )` was added, because it would imply that the attribute `PrintQuality` has two different types: `String` and `int`.

WARNING When you're exposing attributes in a Standard MBean, remember that Java is case sensitive. For example, the method `setPrintQuality()` exposes an attribute `PrintQuality`, whereas `setprintQuality()` exposes a different attribute: `printQuality`.

Table 4.1 breaks down the `PrinterMBean` interface into the parts of the management interface it exposes.

Table 4.1 The exposed attributes and operations of the `PrinterMBean` interface. Attributes are defined by the getter and setter methods. The operations are the methods that are not attributes.

Declared method	Exposed part of management interface
`getPrintJobCount()`	Attribute `PrintJobCount` with read access
`getPrinterName()`	Attribute `PrinterName` with read access
`getPrintQuality()`	Attribute `PrintQuality` with read access
`setPrintQuality( int value )`	Attribute `PrintQuality` with write access
`cancelPrintJobs()`	Operation `cancelPrintJobs`
`performSelfCheck()`	Operation `performSelfCheck`

The two parts of a management interface that an MBean interface does not describe are the public constructors and optional notifications. Notifications are described by a separate interface, `NotificationBroadcaster`, and will be covered later (in chapter 6). Public constructors are found in the class that implements the MBean interface and are discovered by introspection at the agent level.

The MBean interface makes an MBean a Standard MBean. It follows the normal interface rules of the Java language with respect to inheritance and so forth. However, depending on the level at which the interface is implemented, a different management interface may be created for an MBean. The following section describes the different inheritance schemes you can use to create a Standard MBean.

4.3 Standard MBean inheritance patterns

As we've repeatedly mentioned, a Standard MBean is an MBean that implements its own MBean interface. However, you need to be able to recognize the design patterns associated with a Standard MBean. Most likely you have experience with Java, and you understand the ability of Java classes and interfaces to extend other classes and interfaces. However, what effect does subclassing have on the management interface of a Standard MBean?

This section breaks down all the possible inheritance scenarios and explains how each affects the management interface of a Standard MBean. The inheritance patterns are presented in this book because they can affect the management interface you are trying to create.

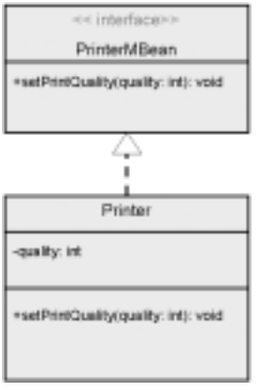


Figure 4.1
The simplest case: direct implementation of an MBean interface. The resulting management interface is the methods contained in the `PrinterMBean` interface.

4.3.1 Direct implementation of an MBean interface

The first scenario deals with an MBean that manages a printer. The pattern is described by figure 4.1, which shows the `Printer` class implementing the `PrinterMBean` interface described earlier.

This is the simplest scenario: a Standard MBean is created by implementing its own MBean interface. The management interface for the `Printer` class contains the methods and attributes exposed in the interface `PrinterMBean`. In this pattern, the `PrinterMBean` interface exposes only one attribute, `PrintQuality` (it is write only).

4.3.2 Inheriting the management interface

Similar to the previous case, a valid MBean can be created by extending another valid Standard MBean. Figure 4.2 depicts the `CopierPrinter` MBean.

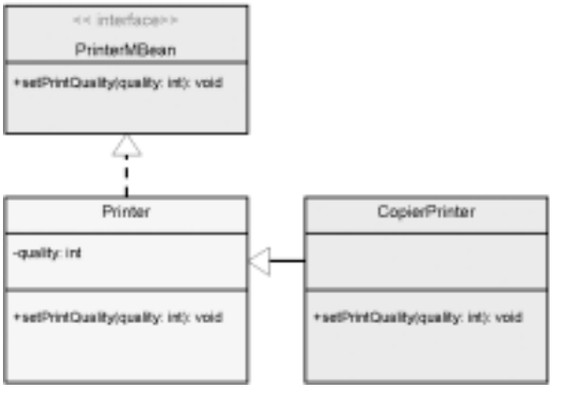


Figure 4.2
Inheriting a management interface by extending another Standard MBean. The `CopierPrinter` MBean will have a management interface identical to that of the `Printer` MBean.

In this case, the class `CopierPrinter` does not directly implement an MBean interface. However, its super class, `Printer`, does implement the `PrinterMBean` interface. The `CopierPrinter` class therefore is a `PrinterMBean`: it inherits the management interface of its super class.

This technique is useful if you want to change the behavior of an MBean but keep the interface unchanged. When you inherit the interface, you cannot add to it, but you can override methods in order to provide a new implementation in the subclass.

Keep in mind that the `CopierPrinter` class must still follow the other MBean rules. Thus it must not be an abstract class, and it must provide a public constructor because constructors cannot be inherited.

4.3.3 Overriding the management interface

The previous case mentioned overriding methods from an inherited MBean interface implementation. This scenario shows you how to override a management interface entirely with a new one (see figure 4.3).

Remember that one of the rules for writing an MBean is that it can only implement a single MBean interface. However, notice in this case that both the `Printer` and `CopierPrinter` classes implement an MBean interface. In this scenario, the `CopierPrinterMBean` interface replaces the management interface of the `PrinterMBean` interface. The `CopierPrinter` class still inherits the methods and implementation from its super class, but JMX agents will not recognize those methods as part of the `CopierPrinter` MBean's management interface. Only the

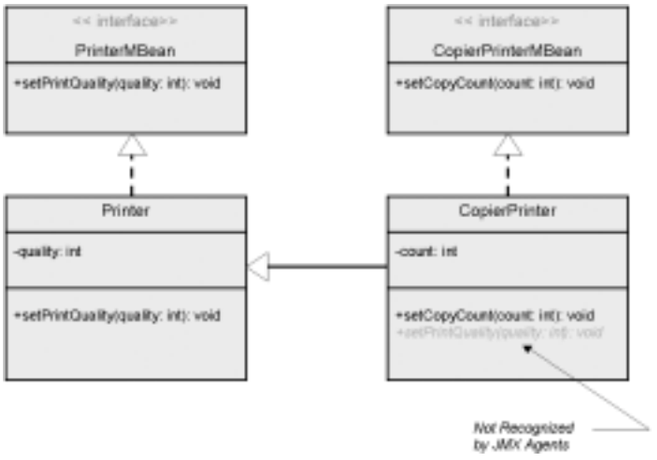


Figure 4.3 Overriding a management interface inherited from a super class. The management interface of the `CopierPrinter` MBean is declared by the `CopierPrinterMBean` interface. Nothing from the super class or the `PrinterMBean` interface is used.

methods and attributes exposed by the `CopierPrinterMBean` interface will be considered part of the management interface.

4.3.4 Extending the management interface

As you know, Java interfaces can extend other Java interfaces. This scenario presents the case when an MBean interface extends another MBean interface. The resulting management interface exposed is the combined methods of both interfaces. Figure 4.4 illustrates this concept with the `Printer` MBean.

Just as in the previous scenario, the `Printer` MBean can implement only a single MBean interface. However, the `PrinterMBean` interface can extend other MBean interfaces, adding more management capability. In this scenario, the `Printer` class is a `PrinterMBean`, not a `DeviceMBean`. This MBean's management interface includes the methods and attributes exposed by both the `DeviceMBean` and `PrinterMBean` interfaces, because a JMX agent will consider them part of the single MBean interface, `PrinterMBean`, implemented by the `Printer` class.

4.3.5 Combination of extending and overriding

By combining the last few scenarios, you can create the case shown in figure 4.5.

Looking at the figure, you can see that the `CopierPrinter` class inherits its management interface and implements its own MBean interface. In addition, the `CopierPrinterMBean` interface extends the `PrinterMBean` interface. The resulting management interface in this case is the same as the previous scenario. The management interface is always taken from the most closely related MBean interface. This means that if an MBean interface is implemented directly, it takes precedence over an inherited one. However, because `CopierPrinterMBean` extends

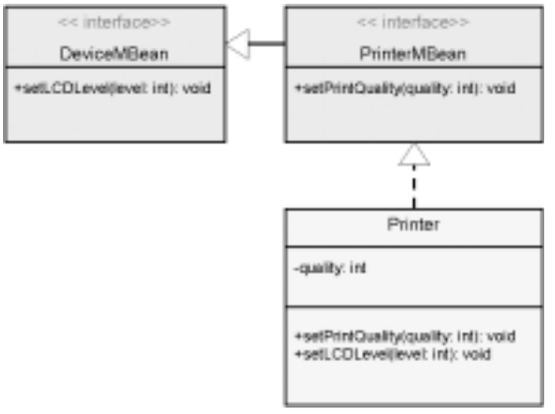


Figure 4.4
Creating an MBean interface by extending an existing MBean interface. The `Printer` MBean management interface is composed of the methods from both the `DeviceMBean` and `PrinterMBean` interfaces.

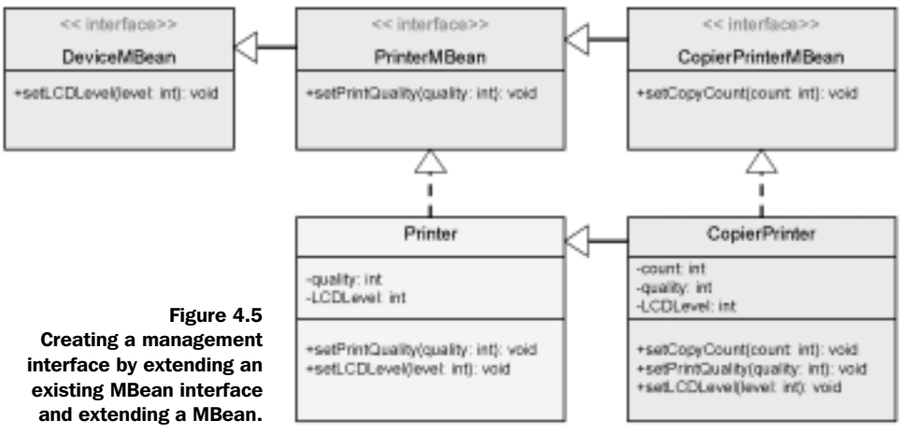


Figure 4.5
Creating a management interface by extending an existing MBean interface and extending a MBean.

`PrinterMBean`, the management interface of the `CopierPrinter` class includes the attributes and operations from both MBean interfaces.

4.3.6 Extending a non-MBean interface

As we showed in section 4.3.4, the MBean interface can extend another MBean interface. It can also extend a non-MBean interface. Figure 4.6 depicts such a scenario.

When an MBean interface extends another interface that is not an MBean interface, the resulting exposed attributes and operations are determined by both interfaces because the MBean interface inherits all the methods of its parent. It does not matter that the `Device` interface does not follow the MBean interface naming pattern.

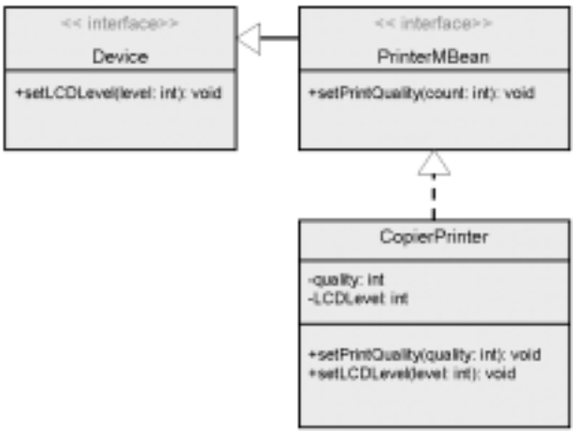


Figure 4.6
Creating an MBean interface by extending a non-MBean interface. The `CopierPrinter` management interface will be composed of methods from both the `Device` and `PrinterMBean` interfaces.

4.4 Standard MBeans in action

In the previous section, you learned what it takes to write Standard MBeans. You know they must be concrete classes, have at least one public constructor, and follow certain inheritance patterns. You also learned the components of the management interface of a Standard MBean. At this point, you should be ready and eager to see some code examples.

The Standard MBean is straightforward and simple: you don't need to create complex data structures or algorithms to create a Standard MBean. Therefore, you should not have any problems understanding the examples presented in the next few sections. These examples are intended to help you understand how MBeans can be used in your own applications.

For this section, consider the application that contains its own instance of the `MBeanServer` class, or embedded JMX agent. Remember, a JMX agent is a Java class that acts as the container of MBeans. Agents have a small footprint and can be included easily into an application. When you include a JMX agent, the application can use MBeans for many purposes. This section describes using Standard MBeans to make your applications configurable and componentized. Figure 4.7 illustrates this concept.

You won't see any UML diagrams for the remaining examples in this chapter. Each MBean example implements its own MBean interface as described in the inheritance patterns in the previous section.

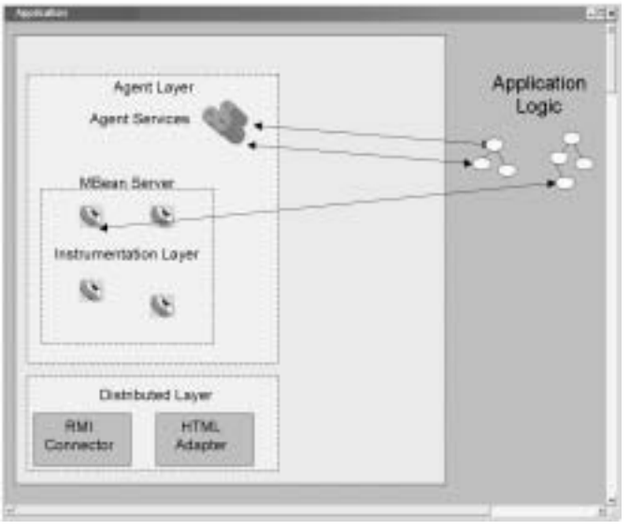


Figure 4.7
Embedding a JMX agent in an application. The application contains its own MBean server, which it can use to contain components of functionality. Doing so allows it to use and register its own MBeans.

4.4.1 Making applications easily configurable

In chapter 1, you read that one of the benefits of using JMX in your applications is that it can make them more configurable. With JMX, you can expose APIs from your application for management. By exposing certain operations, you can use MBeans to expose the behavior of your application at runtime. In other words, you can use MBeans to expose an API that configures your application. The configuration ability will give your applications more flexibility and can possibly save you downtime. The next section describes using MBeans to encapsulate your application properties.

Managing application properties

Many applications are configured by loading a set of properties from the file system. Unless the application chooses to monitor and reload the properties file, it can only be refreshed by being restarted. For many applications, it's not feasible to stop and start just for a minor reconfiguration. Applications that have the ability to be reconfigured during runtime are more flexible, powerful, and long lasting.

The `PropertyManager` Standard MBean example manages a set of properties. An application can acquire its configuration by using this MBean, and users can update the configuration by connecting to the embedded JMX agent. The first step in creating this MBean is to define its MBean interface:

```
package jmxbook.ch4;

import java.util.*;

public interface PropertyManagerMBean
{
  public String getProperty( String key );
  public void setProperty( String key, String value );
  public Enumeration keys();
  public void setSource( String path );
}
```

Now that you know how an MBean interface describes a Standard MBean's management interface, look at the `PropertyManagerMBean` interface to determine the management interface it describes. Judging by the fact that there is a `getProperty()` method and a `setProperty()` method, you might think the interface exposes a readable and writable attribute. However, the get method in this case is an operation, not an exposed readable attribute; a getter method cannot accept arguments. Likewise, the `setProperty()` method is not really a setter method—setter methods can take only a single argument. Therefore, although acceptable, this

interface is misleading to a human reader. By definition, this interface exposes only one (writable) attribute: `Source`. All other methods are exposed operations.

Listing 4.1 shows the implementation of the `PropertyManagerMBean` interface.

Listing 4.1 PropertyManager.java

```
package jmxbook.ch4;

import java.util.*;
import java.io.*;

public class PropertyManager implements PropertyManagerMBean
{
  private Properties props = null;

  public PropertyManager( String path )
  {
    try
    {
      //load supplied property file
      props = new Properties();
      FileInputStream f =  new FileInputStream( path );
      props.load( f );
      f.close();
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }

  public String getProperty( String key )
  {
    return props.getProperty( key );
  }

  public void setProperty( String key, String value )
  {
    props.setProperty( key, value );
  }

  public Enumeration keys()
  {
    return props.keys();
  }

  public void setSource( String path )
  {
    try
    {
      props = new Properties();
      FileInputStream f =  new FileInputStream( path );
```

```
      props.load( f );
      f.close();
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }

  }
}//class
```

This MBean is straightforward and exposes only the methods present in the `java.util.Properties` class. The only attribute, `Source`, is used to reset the entire properties set with a new properties file.

Properties are the most obvious way to use MBeans to make applications configurable. However, you can also use an MBean to configure a single part of your application, such as database access.

Configuring a DataSource

Many applications need the services of a database. Java applications use the JDBC API to open database connections by creating a `Connection` object or by acquiring `Connection` objects from a `DataSource` object. In both cases, it might be useful to configure the creation of database connections. You can do so by encapsulating the acquisition of database connections inside an MBean.

The following example is a simple Standard MBean that acquires database connections from a `DataSource` object. It gets the `DataSource` object by using a Java Naming and Directory Interface (JNDI) lookup; it could just create the connection directly, as well. The following is the MBean interface for the `DBSource` MBean (to learn more about using JNDI or JDBC, go to http://www.javasoft.com):

```
package jmxbook.ch4;

import java.sql.*;

public interface DBSourceMBean
{
  public void resetDataSource( String name );
  public void setAutoCommit( boolean commit );
  public boolean getAutoCommit( );
  public Connection getConnection();
}
```

As you can see, the `DBSourceMBean` interface appears to expose one read/write attribute, `AutoCommit`, and one readable attribute, `Connection`. It also exposes an operation, `resetDataSource()`. Listing 4.2 shows the `DBSource` class.

Listing 4.2 DBSource.java

```
package jmxbook.ch4;

import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class DBSource
{

  private DataSource ds = null;
  private boolean commit = false;

  public DBSource( String JNDIName )
  {
    try
    {
      //lookup data source using JNDI
      Context ctx = new InitialContext();
      ds = ( DataSource ) ctx.lookup( JNDIName );
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }

  }

  public void resetDataSource( String name )
  {
    try
    {
      Context ctx = new InitialContext();
      ds = ( DataSource ) ctx.lookup( name );
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }

  public Connection getConnection()
  {
    Connection con = null;

    try
    {
      con = ds.getConnection();
      con.setAutoCommit( commit );
      return con;
    }
    catch( Exception e )
    {
```

❶ **Expose getConnection() operation**

```
        e.printStackTrace();
        con = null;
        return null;
      }
    }

    public boolean getAutoCommit( )
    {
      return commit;
    }

    public void setAutoCommit( boolean commit )
    {
      this.commit = commit;

    }
}
```

❶ The `getConnection()` method is unique because the implementation class does not contain a `Connection` attribute. In fact, it really returns a connection from the `DataSource` object. The `getConnection()` method is more like an operation than an exposed attribute. This situation illustrates again the importance of carefully naming methods for an MBean interface—for example, perhaps you should name this method `acquireConnection()`. By naming methods thoughtfully, you can avoid misunderstandings.

Testing the PropertyManager MBean

Before moving to the next section, let's run one of these MBeans in the `JMXBookAgent` agent from chapter 3. (As we've mentioned, you will use this agent class at various times in the book.)

You can register an MBean into your agent two ways: you can use either the HTML adapter or the Remote Method Invocation (RMI) connector. You have already seen how to use the HTML adapter, so let's take this chance to use the RMI connector to register an MBean in the agent. To do so, you need to write a simple setup class that contacts an instance of the `JMXBookAgent` and registers an MBean. Listing 4.3 shows a setup class to create the `PropertyManager` MBean.

Listing 4.3 PropertyManagerSetup.java

```
package jmxbook.ch4;

import javax.management.*;
import com.sun.jdmk.comm.*;
import jmxbook.ch3.*;

public class PropertyManagerSetup
```

```
{
   public PropertyManagerSetup()
   {
     try
     {
        RmiConnectorClient client = RMIClientFactory.getClient();
        ObjectName propertyName = new
                    ObjectName( "JMXBookAgent:name=property");

        client.createMBean( "jmxbook.ch4.PropertyManager",
                            propertyName );

     }
     catch( Exception e )
     {
       ExceptionUtil.printException( e );

     }
   }

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

The setup class uses the `RMIClientFactory` class to acquire an RMI client with which to contact your agent. Using the client, it invokes the `createMBean()` method of the `MBeanServer`. When you used the HTML adapter in chapter 2, you caused the same thing to happen by using the browser.

Before you run the setup class, make sure you have an instance of the `JMXBookAgent` running—use the following command to do so:

```
javac jmxbook.ch4.JMXBookAgent
```

After the agent successfully starts, execute the `PropertyManagerSetup` class to create your `PropertyManager` MBean. Open your web browser to the address of the agent's HTML adapter, and you will see the new MBean registered in the agent.

The `PropertyManager` and `DBSource` MBeans are both good examples of using MBeans to make an application more configurable. The next section deals with making an application componentized.

4.4.2 Breaking applications into components

Chapter 1 explains that it is possible to use JMX to break applications into manageable components. *Componentization* is a development method that defines interfaces between components of an application, allowing their implementations to be changed or even replaced. With Standard MBeans, you can define

unchanging MBean interfaces that an application uses to access certain implementations of functionality it needs. With the MBean interfaces staying the same over time, you can change the MBean implementation as needed, preserving access to the functionality. The next example demonstrates this concept.

Abstracting a data layer

We already showed how an application can encapsulate the creation of database connections. Taking that concept a little further, an application can abstract its entire data access layer by using JMX. Figure 4.8 illustrates the data abstraction concept.

This example is presented as a Standard MBean because it would be developed with a well-known interface. It would be developed along with the application, and its interface could be defined in advance. For this scenario to work, the application needs to send and receive data to the data layer in a form independent of the persistence mechanism. The interface to the data layer is dependent on the application, so a full code example is not too useful. However, you could expect the interface to resemble something like the following:

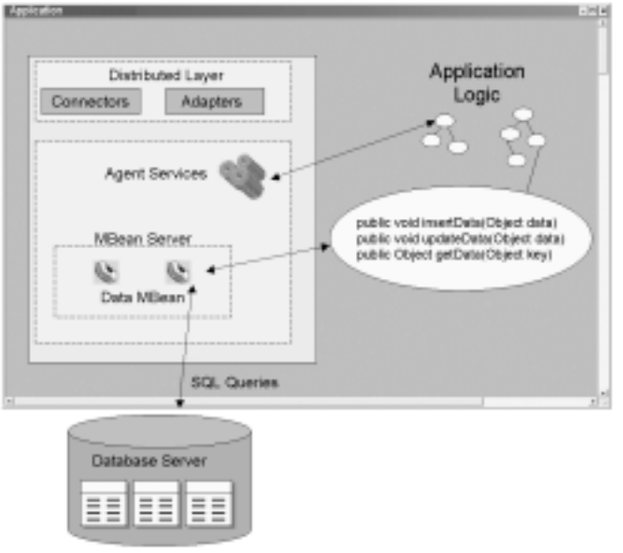


Figure 4.8
Abstracting a data layer using JMX. A Standard MBean shields the application from the actual implementation of the data layer.

```
public interface DataLayerMBean
{
  public boolean insertData( Object data );
  public boolean updateData( Object data );
  public boolean deleteData( Object data );
  public boolean retrieveData( Object data );
}
```

The data object should contain enough information for the persistence implementation to fulfill its task.

The logging MBean

Up to this point, the examples have pertained to Java resources such as properties and JDBC connections. One example had to do with application configuration, and the other introduced MBeans as components of an application. The final example of this section combines these concepts.

Most applications use logs to keep records of activity and occurrences of errors. In many cases, the log file also records developer debug statements for possible analysis. Like data repositories, logs can be kept in many different forms, such as a flat file or a database. By writing an MBean, you can both componentize the application's logging system and expose the logging system to a management tool. Exposing the logging mechanism allows you to tune it for certain behavior. For instance, using a management tool, you can tell the logging MBean to record only critical errors.

By defining the log system as an MBean, you not only encapsulate its implementation, but also expose it for configuration. The following example creates such an MBean. First, let's look at its MBean interface, `LoggerMBean`. As you know, the interface describes the MBean's exposed attributes and operations:

```
package jmxbook.ch4;

public interface LoggerMBean
{
  public void setLogLevel( int level );
  public int getLogLevel();
  public String retrieveLog( int linesback );
  public void writeLog( String message, int type );
}
```

Table 4.2 describes the management interface exposed by the `LoggerMBean` interface.

Table 4.2 The parts of the `LoggerMBean` interface. Attributes are described by the getter and setter methods, and operations are described by the remaining methods.

Declared method	Part	Description
`setLogLevel()`	Attribute	Declares write access to `LogLevel`
`getLogLevel()`	Attribute	Declares read access to `LogLevel`
`retrieveLog()`	Operation	Declares an exposed operation
`writeLog()`	Operation	Declares an exposed operation

The `Logger` class implements the `LoggerMBean` with a flat-file implementation. Listing 4.4 shows the `Logger` class.

Listing 4.4 Logger.java

```
package jmxbook.ch4;

import javax.management.*;
import java.io.*;
import java.util.*;

public class Logger implements LoggerMBean
{

  public static final int ALL    = 3;
  public static final int ERRORS = 2;
  public static final int NONE   = 1;

  private PrintWriter out = null;
  private int logLevel = Logger.ALL;

  public Logger()
  {
    try
    {
      //open the initial log file
      out = new PrintWriter(
                 new FileOutputStream
                 ( "record.log " ) );
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }

  public void setLogLevel( int level )
  {
    logLevel = level;
  }
```

```
    public int getLogLevel()
    {
      return logLevel;
    }

    public String retrieveLog( int linesback )
    {
      //implementation here
      return null;
    }

    public void writeLog( String message, int type )
    {
      try
      {
        if( type <= logLevel )
          out.println( message );
      }
      catch( Exception e )
      {
        e.printStackTrace();
      }
    }

  }//class
```

Breaking applications into components as Standard MBeans is a valuable development advantage. In this manner, you can keep the interfaces between components stable, and shield the application from the implementation of specific functionality. Once the implementation is hidden, changing it will not impact the application. For example, if an application was using a logging MBean like the previous example, the logging implementation could persist messages to a file or a database. Neither method affects how an application would access its logging functionality.

4.4.3 MBeans using other MBeans

In the past two sections, we have discussed using Standard MBeans to componentize and configure your applications. We presented each of these concepts with a single MBean: you used an MBean to manage a property set that an application could use to access application settings, and you also used an MBean to handle an application's logging functionality.

You have created an application that uses MBeans to handle its configuration and certain components. However, as part of the application, the MBean components should have access to the configuration of the application. In this scenario, your MBean components need access to another MBean. Figure 4.9 illustrates this concept using the `Logger` MBean and `PropertyManager` MBean.

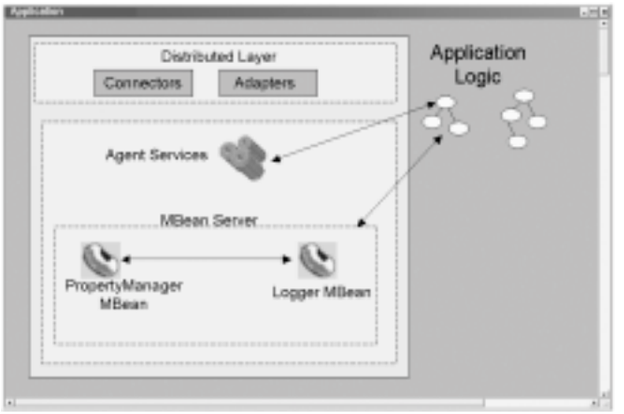


Figure 4.9
The `Logger` MBean accessing the `PropertyManager` MBean. The `Logger` class can use another MBean because it has a reference to the MBean server. It acquires this reference by implementing the `MBeanRegistration` interface.

For one MBean to use another MBean, it must be able to contact the MBean server. You could pass in an MBean as a parameter to another's constructor, but you don't want to create an unnecessary dependency on the MBean. You can implement an MBean to contain a reference to its MBean server two ways:

- Construct the MBean with an MBean server parameter.
- Implement the `MBeanRegistration` interface.

This section discusses the `MBeanRegistration` interface. This interface declares methods that are invoked before and after registration and deregistration on the MBean server. The following is the `MBeanRegistration` interface:

```
package javax.management;

public abstract interface MBeanRegistration
{
  public void postDeregister();
  public void postRegister( Boolean registrationDone );
  public void preDeregister();
  public ObjectName preRegister( MBeanServer server,
                                 ObjectName name );
}
```

This interface contains two methods that are called in conjunction with the MBean's registration on a MBean server, and two methods that are called with deregistration. These methods are invoked by the `MBeanServer` instance that is being asked to perform the registration or deregistration of a particular MBean instance. For example, if the `HelloWorld` MBean from chapter 2 implemented this interface, the MBean server would perform the following tasks when asked by the HTML adapter to create another `HelloWorld` MBean instance:

1. Create the MBean instance using the appropriate constructor.
2. Invoke the `preRegister()` method.
3. Register the MBean instance.
4. Invoke the `postRegister()` method.

The `postRegister()` method is invoked with a `Boolean` value passed as a parameter. This value indicates whether the registration of the MBean was successful. If the value is `true`, registration succeeded. The `preRegister()` method allows the MBean to find and use other MBeans. It takes two parameters: an `MBeanServer` instance and an `ObjectName` instance. If the `ObjectName` parameter is passed as null, the method should return an appropriate `ObjectName` value to use with the registration of this MBean.

Revisiting the `Logger` MBean, listing 4.5 shows how to implement the `MBeanRegistration` interface to provide the `Logger` class with a mechanism to get the initial values for its attributes. The changes from the previous `Logger` MBean class (listing 4.4) are shown in bold.

Listing 4.5 Logger.java

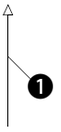
```
package jmxbook.ch4;

import javax.management.*;
import java.io.*;
import java.util.*;

public class Logger implements LoggerMBean, MBeanRegistration
{

  public static final int ALL    = 3;
  public static final int ERRORS = 2;
  public static final int NONE   = 1;

  private PrintWriter out = null;
  private int logLevel = Logger.ALL;
  private MBeanServer server = null;

  public Logger()
  {
    try
    {
      out = new PrintWriter(
                 new FileOutputStream (
                 "record.log" ) );
    }
    catch( Exception e )
    {
      e.printStackTrace();
```

```
  }
}

public void setLogLevel( int level )
{
  logLevel = level;
}

public int getLogLevel()
{
  return logLevel;
}

public String retrieveLog( int linesback )
{
  //implementation here
  return null;
}

public void writeLog( String message, int type )
{
  try
  {
    if( type <= logLevel )
      out.println( message );
  }
  catch( Exception e )
  {
    e.printStackTrace();
  }
}
```

Implement MBeanRegistration interface ❶

```
public void postDeregister() {}
public void postRegister( Boolean registrationDone ) {}
public void preDeregister() {}

public ObjectName preRegister(
               MBeanServer server, ObjectName name )
{
  this.server = server;
  try
  {
    ObjectName name1 = new ObjectName(
          "JMXBookAgent:name=props" );
    Object[] params = { "loglevel" };
    String[] sig = { "java.lang.String" };
    String value = ( String )
         server.invoke( name1, "getProperty", params, sig );
    logLevel =
       Integer.parseInt( value );
  }
  catch( Exception e )
  {
```

```
            e.printStackTrace();
            logLevel = 0;
        }
        return name;
    }
}//class
```



❶ The methods that appear in bold are declared in the `MBeanRegistration` interface. For the first three, the `Logger` MBean did not provide an implementation. However, the `preRegister()` method is implemented to get its initial `logLevel` attribute value from a `PropertyManager` MBean present in the MBean server passed in as an argument to this method.

To find the value for the `logLevel` attribute, the `Logger` MBean must invoke the `getProperty()` method of a registered `PropertyManager` MBean. For this example, the `Logger` MBean assumes that the object name `HelloAgent:name=props` will correspond to a `PropertyManager` MBean. In the code, the appropriate method signature is created to allow the MBean server to invoke the `getProperty()` method. The `Logger` MBean invokes the MBean server's `invoke()` method with the appropriate parameters.

WARNING When you're creating an MBean that depends on the existence of another MBean, you need to implement some default behavior in case the necessary MBean does not exist. For example, the `Logger` MBean must ensure that its attributes have appropriate values if an exception occurs when invoking methods on the `PropertyManager` MBean.

The `MBeanRegistration` interface is useful for acquiring a reference to the containing MBean server. In addition, because it declares methods that are invoked before an MBean is removed from the MBean server, implementing MBeans can be informed when to clean up resources before the MBean is removed from the MBean server.

4.5 Handling MBean errors

In each example in this chapter, there is an opportunity to catch an exception. Each MBean contains a generic `try-catch` statement:

```
try
{
   //code
}
catch( Exception e )
{
   e.printStackTrace();
}
```

You can see the drawbacks of this approach. Imagine you are using management software to configure the `PropertyManager` MBean you created in section 4.3.1. You create that MBean by specifying a path to a properties file. During construction, the MBean attempts to open a file with that path and load it into a properties object. If that file does not exist, an exception is thrown and basically ignored. From the management tool, you would never know what had occurred, and your application's configuration would be in error.

To adequately manage an application, you must know if your management actions succeed or cause errors. Fortunately, JMX provides a way to avoid the situation we just described, by supporting runtime and declared exceptions. Declared exceptions are declared in a `throws` statement. Runtime exceptions are not expected and are not required to be in a `try-catch` statement.

Exceptions in JMX occur in two categories. First, exceptions occur as agent-level components (such as `MBeanServer`) perform operations on an MBean. For example, registration, lookup, and invoking methods on an MBean instance fall into this category. Second, exceptions occur as defined by MBean code. These include Java language exceptions and user-defined exceptions.

JMX supports a mechanism for handling exceptions in a meaningful way. The last sections of this chapter discuss exceptions at the MBean level.

4.5.1 Throwing exceptions

Figure 4.10 depicts the class hierarchy for JMX exceptions. The main class is `JMException`; all others are subclasses of it. It has three subclasses: `OperationsException`, `ReflectionException`, and `MBeanException`. Each of these exception types has subclasses, as well.

You can see that JMX exceptions at the MBean level are broken down into three categories, extending from one of three super classes:

- `OperationsException`—Subclasses define exceptions that occur when invoking operations on an MBean.
- `ReflectionException`—Wraps standard reflection exceptions of the Java language that occur when working with MBean classes.

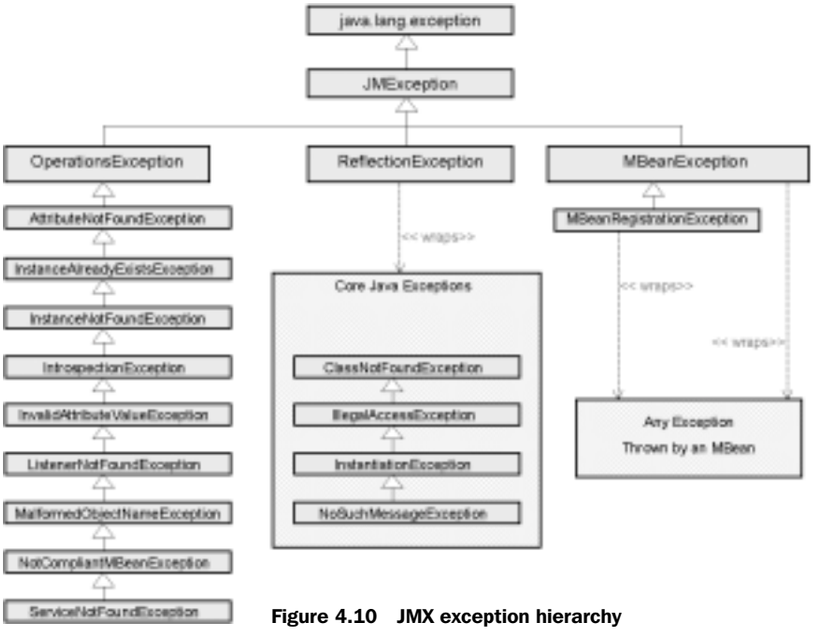


Figure 4.10 JMX exception hierarchy

- `MBeanException`—Wraps any other (user-defined or standard) exceptions thrown from an MBean. The MBean server constructs and throws this exception when an unknown exception is thrown by an MBean.

Table 4.3 describes the remaining exceptions. All these exceptions will propagate from the MBean server, because it is the object that will detect (and possibly wrap) the condition that causes a problem. Again, not all subclasses of `JMException` are listed here.

Table 4.3 JMX exceptions

Exception	Description
`AttributeNotFoundException`	Thrown when a specified attribute cannot be found (does not exist for the MBean specified)
`InvalidAttributeValueException`	Thrown when the specified attribute contains an invalid value for that attribute
`InstrospectionException`	Thrown if an error occurs when the MBean server is examining the management interface of an MBean
`NotCompliantMBeanException`	Occurs when attempting to register an MBean, if the MBean does not follow the applicable rules

Table 4.3 JMX exceptions *(continued)*

Exception	Description
`MBeanRegistrationException`	Wraps exceptions thrown by the `preRegister()` and `preDeregister()` methods of the `MBeanRegistration` interface
`ClassNotFoundException`	Thrown if the MBean server cannot find a specified MBean class when creating an MBean
`InstantiationException`	Thrown by the `newInstance()` method from the `Class` class when trying to create an MBean instance
`IllegalAccessException`	Thrown by the `Class.forName()` method when the MBean server is trying to create an MBean instance
`NoSuchMethodException`	Thrown when trying to invoke a non-existent method on an MBean

4.5.2 Runtime exceptions

Runtime exceptions are handled in the same manner as other exceptions. Operations performed on MBeans occur in a `try-catch` statement inside the JMX agent, allowing the agent to catch any runtime exceptions and wrap them in a JMX exception. The JMX framework defines a subclass of `java.lang.RuntimeException` called `JMRuntimeException`. In JMX, there are subclasses for runtime exceptions at the agent level and the MBean level. Figure 4.11 shows the class

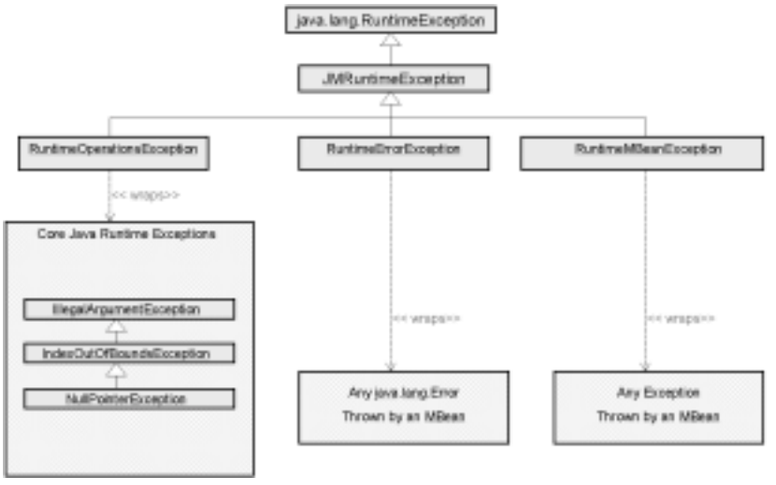


Figure 4.11 JMX runtime exception hierarchy

hierarchy of the JMX runtime exceptions for the MBean level. Note that JMX wraps both runtime errors and runtime exceptions in its own runtime exception.

Like JMX exceptions, JMX runtime exceptions at the MBean level are broken down into three categories:

- `RuntimeOperationsException`—Wraps Java runtime exceptions that occur during operations on an MBean
- `RuntimeErrorException`—Wraps standard runtime errors of the Java language that occur at the MBean level
- `RuntimeMBeanException`—Wraps any other (user-defined or standard) runtime exceptions thrown from an MBean

4.6 Summary

This chapter introduced the Standard MBean. The Standard MBean uses an explicitly declared management interface to interact with a manageable resource. The explicitly declared interface—the MBean interface—makes the Standard MBean a static, unchanging MBean used for well-known or predefined resources.

The examples in this chapter covered the three following topics:

- *Configuration*—Using MBeans to make your applications more configurable. The example was a Standard MBean used to manage a properties object.
- *Componentization*—Using MBeans to break your applications into components, allowing you to alter or replace component implementations. The `Logger` MBean demonstrated this concept.
- *MBeans using MBeans*—Combining both previous concepts, the `Logger` MBean used the `PropertyManager` MBean to initialize one of its member variables.

At the end of this chapter, you got your first look at the exception hierarchy for exceptions that occur when working with MBeans. JMX provides exceptions for many situations that may occur when reflecting upon or invoking MBean objects. In addition, JMX provides other exception classes to wrap core Java exceptions and user-defined exceptions.

Chapter 5 introduces the Dynamic MBean. The Dynamic MBean is used to manage evolving resources in situations where the Standard MBean may not be appropriate.