

# *“Hello World,” the JMX way*

---

- Writing your first MBean
- Writing a simple JMX agent
- Introducing object names
- Using the HTML adapter from Sun

Imagine that you decide to buy a new stereo. You go to the store, pick one out, and bring it home. Are you the type of person who carefully unwraps everything, checks the parts list, and follows the setup instructions step by step? Or do you open everything and start figuring out all the connections on your own? This chapter is written for those of you in the second group. If you fall into the first group, please be sure to read chapter 1; it presents the need for the JMX framework, as well as JMX’s overall architecture (which is only recapped here in chapter 2).

This purpose of this chapter is to familiarize you with the JMX Reference Implementation (RI) provided by Sun Microsystems. After completing this chapter, you will have managed your first resource, created a simple JMX agent, and communicated with the agent from a web browser. In other words, you will create an MBean, use the MBean server, and manage your MBean using the HTML adapter provided by Sun in the JMX RI.

---

**NOTE** The remainder of the book assumes that you already have the JDK 1.3 (at minimum) installed on your machine and that you have it included in your `PATH`. If necessary, you can download it from <http://www.javasoft.com>.

## 2.1 Getting started

---

Before we get too far along, let’s have a quick architectural review and create a development environment.

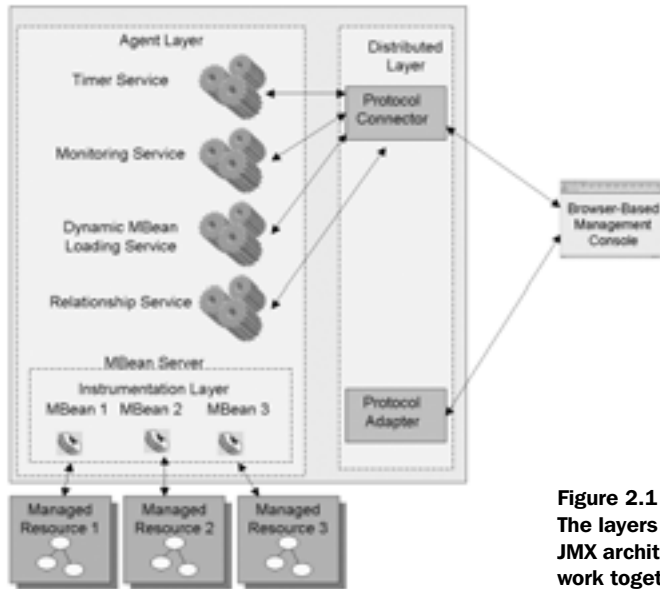
### 2.1.1 A JMX architecture refresher

Chapter 1 detailed the JMX architecture and discussed how it provides a management solution. However, to ensure you get the most of this chapter, let’s have a brief refresher. The JMX architecture lays out a Java framework consisting of three main parts, or layers, that work together to provide a Java management solution. Table 2.1 lists the three layers of the JMX architecture.

**Table 2.1** The three JMX component layers

Layer	Description
Instrumentation layer	Contains MBeans and their manageable resources
Agent layer	Contains the JMX agents used to expose the MBeans
Distributed layer	Contains components that enable management applications to communicate with JMX agents

Figure 2.1 illustrates how the layers work together.



**Figure 2.1**  
The layers of the  
JMX architecture  
work together.

In this chapter, you will interact with components from each layer. From the instrumentation layer, you will be using an MBean. MBeans are Java objects that encapsulate a resource and expose it for management. From the agent layer, you will use a JMX agent. Actually, you will write your own agent to contain your MBean. And finally, from the distributed layer, you will use the HTML adapter, which is a Java object that allows management applications to communicate with your agent over HTML as a communication protocol. Management applications are any applications that are interested in accessing, configuring, or manipulating manageable resources.

### 2.1.2 Setting up the development environment

If you don't already have the JMX RI from Sun Microsystems, download it from <http://www.java.sun.com>. Download the 1.0 version of JMX. Once you have downloaded the zip file, extract it to your hard drive. The extracted zip file produces a JMX parent directory containing the following directories:

- *contrib*—Contains unsupported contributions from Sun Microsystems. For example, Sun provides a Java RMI connector, which, like the adapters, allows management applications to communicate with JMX agents.
- *jmx*—Contains the JMX RI, examples, and documentation.

For the remainder of this book, you will keep your Java source files in a folder called JMXBook. In addition, you will use a setup batch file to set your `PATH` and `CLASSPATH` for compiling and running the examples from the JMXBook directory. The batch file contains the following lines:

```
set CLASSPATH=c:\JMXBook;C:\jmx-1_0_1-ribin\jmx\lib\jmxri.jar;  
C:\jmx-1_0_1-ribin\jmx\lib\jmxtools.jar;  
C:\jmx-1_0_1-ri-bin\contrib\remoting\jar\jmx_remoting.jar;  
set PATH=c:\jdk1.3\bin
```

The setup batch file is used to set up the JMX environment for compiling and running the examples in a Windows environment. If you are using Unix, you will need to modify the script accordingly. As you can see from the `PATH` environment variable, we are using the JDK version 1.3, but every example should work with any Java 2 Platform Standard Edition. When working from the command line, you will invoke this file (`setup.bat` in our case) before doing anything else. After running the setup script, you can test your `CLASSPATH` by typing `java javax.management.ObjectName`. You should get an error indicating that the class does not contain a `main()` method.

---

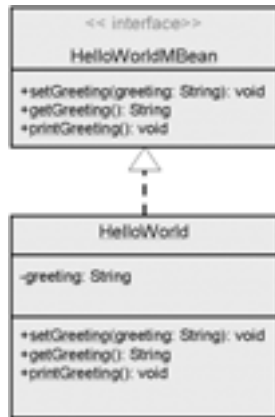
**NOTE** Using Ant: For those of you familiar with Ant (or willing to find out more), we included an Ant build system setup in appendix B. In the appendix you will find the Ant XML build doc and information for setting up your environment.

## 2.2 Managing your first resource

---

So far we have refreshed what you already know about the major JMX components, and ensured that you have a good working environment. Now you are ready to create your first MBean. Keep in mind that this chapter is intended only as an introduction to MBeans; more complex examples are presented in later chapters.

For the first example, you'll create a simple `HelloWorld` MBean. The `HelloWorld` MBean exposes a `java.lang.String` object, its only member variable, as a manageable resource. We will use this example as a tool to introduce you to working with the entirety of JMX, including the MBean server and the HTML adapter. Remember, a manageable resource is any resource that can be accessed and configured via an MBean. (For this chapter, don't worry about the coding standards of MBeans. However, remember from chapter 1 that this book covers three types of MBeans: Standard, Dynamic, and Model. You will create only a Standard MBean in this chapter; the exact rules for developing MBeans are presented in detail in later chapters.) Figure 2.2 shows the UML diagram for the `HelloWorld` MBean.



**Figure 2.2**  
The `HelloWorld` class

The next section discusses writing both the interface and implementing class shown in the figure.

### 2.2.1 Writing the `HelloWorld` MBean

The first step in developing the `HelloWorld` MBean is to write its Java interface. The interface declares three methods: one *getter*, one *setter*, and an additional method for printing the `HelloWorld` MBean's greeting. Normally, you might not write an interface for a simple `HelloWorld` example class like this one. However, as you will learn in chapters 4 and 5, JMX uses interfaces to describe the exposed attributes and operations of an MBean.

Recall that a getter method is a class method with a name in the form of `getMember()`, and a setter method is a class method with a name in the form of `setMember()`. Think of the methods in a Standard MBean interface as the description of the implementation class. Put simply, you should be able to understand the purpose of the methods by their names. In addition, the getter and setter methods define the member variable access granted to objects that use the MBean. By creating a getter method for a member variable, you grant *read* access to it. A setter method grants *write* access. As you can see from the following interface, this MBean is quite simple:

```
package jmxbook.ch2;
public interface HelloWorldMBean
{
    public void setGreeting( String greeting );
    public String getGreeting();
    public void printGreeting();
}
```

An important item to notice is the package statement. All examples in this chapter are in the package `jmxbook.ch2`, and each chapter will package its examples accordingly (for example, `jmxbook.ch3` for chapter 3).

The `HelloWorldMBean` interface declares a getter (`getGreeting()`) and setter (`setGreeting()`), as well as a `printGreeting()` method. You’ll use the `printGreeting()` method later to display the MBean’s greeting value.

Listing 2.1 shows the implementation of the interface.

**Listing 2.1** `HelloWorld.java`

```
package jmxbook.ch2;

public class HelloWorld implements HelloWorldMBean
{
    private String greeting = null;

    public HelloWorld()
    {
        this.greeting = "Hello World! I am a Standard MBean";
    }


    public HelloWorld( String greeting )
    {
        this.greeting = greeting;
    }

    public void setGreeting( String greeting )
    {
        this.greeting = greeting;
    }

    public String getGreeting()
    {
        return greeting;
    }

    public void printGreeting()
    {
        System.out.println( greeting );
    }
}

```



And with that, you have created your first MBean. Now, in order to test the MBean, you need to create a JMX agent to contain it. The next section discusses the creation of the `HelloAgent` class. After creating your agent, you can begin using the MBean.

## 2.3 Creating a JMX agent

Now that you have your first MBean, you need to make it available for use. To do so, you must register it in a JMX agent. Therefore, you need to create the `HelloAgent` class, which is a simple JMX agent.

As described in chapter 1, JMX agents are JMX components in the agent layer of JMX and are the containers for MBeans. Part 3 of the book covers JMX agents in detail.

The `HelloAgent` class performs three important tasks:

- It creates an `MBeanServer` instance to contain MBeans.
- It creates an HTML adapter to handle connections from HTML clients.
- It registers a new instance of the `HelloWorld` MBean.

As you are about to see, the `HelloAgent` class is probably the simplest agent you will ever write, but it is still quite powerful. This fact again highlights one of the benefits of using JMX: it is simple and yet useful. In a matter of moments, you have developed the `HelloWorld` MBean. You can now easily manage this MBean by writing a simple agent that uses the HTML adapter provided by Sun Microsystems.

Using any web browser, the adapter allows you to interact with the agent to view all registered MBeans and their attributes. Figure 2.3 depicts this interaction.

Specifically, the adapter lets you:

- View the *readable* MBean attributes
- Update the *writable* attributes
- Invoke the other remaining methods

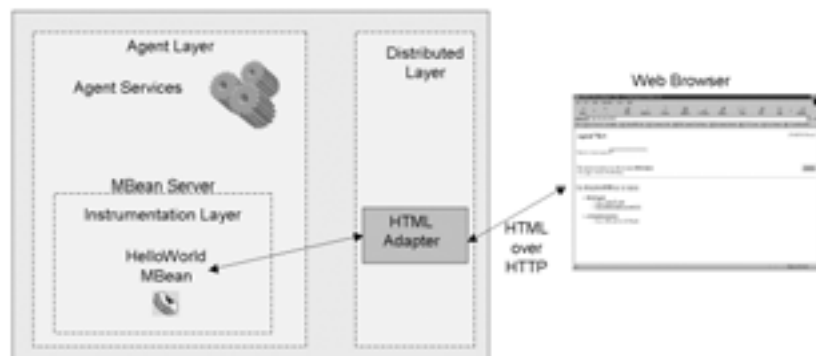


Figure 2.3 Using a Web browser to contact the HTML adapter present in the MBean server

Not only that, the adapter gives you a quick method of dynamically creating and registering additional MBeans. Essentially, the HTML adapter provides you a simple management tool for working with MBeans. The HTML adapter returns a protocol (HTML) that your web browser renders as a useable application. But let's not get ahead of ourselves; first you need to create the simple JMX agent.

### 2.3.1 Writing the HelloAgent class

Listing 2.2 presents the `HelloAgent` class. Don't worry if you don't understand or recognize what is going on in the listing; you will learn more about agents as the book progresses (part 3). For now, you need only a basic understanding of the agent code. Boiled down to the simplest steps, listing 2.2 does the following:

- 1 Creates the MBean server and HTML adapter
- 2 Registers (and thus enables you to manage) the MBean
- 3 Uniquely identifies the MBean
- 4 Registers and starts the HTML adapter

Listing 2.2 HelloAgent.java

```
package jmxbook.ch2;

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

public class HelloAgent
{
    private MBeanServer mbs = null;

    public HelloAgent()
    {
        mbs = MBeanServerFactory.createMBeanServer( "HelloAgent" );
        HtmlAdaptorServer adapter = new HtmlAdaptorServer();
        HelloWorld hw = new HelloWorld();
        ObjectName adapterName = null;
        ObjectName helloWorldName = null;

        try
        {
            helloWorldName =
                new ObjectName( "HelloAgent:name=helloWorld1" );
            mbs.registerMBean( hw, helloWorldName );

            adapterName =
                new ObjectName( "HelloAgent:name=htmladapter,port=9092" );
        }
    }
}
```

**1** Creates HTML adapter

**2** Creates HelloWorld MBean instance

**3** Creates ObjectName instance; registers HelloWorld MBean

**4**



group of MBeans; the domain uniquely differentiates this `MBeanServer` from any other. Each `MBeanServer` contains a supplied domain name, allowing you to group MBeans in a meaningful way. If you invoke the factory `create()` method again with an identical domain name parameter, it will simply return the previously created `HelloAgent MBeanServer` instance. (You will learn more about the `MBeanServer` class later in chapter 8. For now, keep in mind that an `MBeanServer` object acts as a registry, enabling storage, lookup, and manipulation of MBeans.)

- 3 The next step is to create some way for management applications to contact the `HelloAgent`. Recall that agents open themselves up to management applications by constructing protocol adapters and connectors. Adapters and connectors are a major reason why JMX is so powerful and versatile. These components are Java objects that allow management applications to use a specific protocol to contact JMX agents. (You will learn more about them, and create more complex examples, as the book continues.) This chapter only makes use of the HTML adapter; to create the adapter, you just need to invoke its default constructor, as seen in the `HelloAgent` class.

### **Registering and managing the MBean**

- 4 Once the adapter has been created, you need to register it on the `MBeanServer`. This brings up an interesting point about adapters (and connectors): they are also MBeans. Thus the Java classes that make up the adapters and connectors are written to conform to one of the MBean types defined by the JMX specification. Because they are MBeans, they can be managed during runtime like other MBeans. However, before you can register an MBean, you need to make sure you can identify and find it again; this is where the `javax.management.ObjectName` class comes into play.

### **Uniquely identifying MBeans**

So far, the agent has created the `MBeanServer` and registered an HTML adapter with it. Now it is time to examine how the `MBeanServer` keeps track of objects registered with it. Look back at the code that registers the `HelloWorld MBean` instance on the `MBeanServer` 3. When registering an MBean, you need to be able to distinguish it from every other MBean that might be registered.

To do this, you must create an instance of the `javax.management.ObjectName` class. The `ObjectName` class is a JMX class that provides a naming system for MBeans, allowing unique identification of MBeans registered in the `MBean` server. Each `ObjectName` consists of two parts:

- *A domain name*—The domain name usually coincides with the domain name of the `MBeanServer` in which the MBean wants to register. When it does not, it is usually meant to segregate one MBean from the others.
- *A key=value property list*—Property name/value pairs are used to uniquely identify MBeans, and also to provide information about the MBean. The object name may be the first representation a user will see of your MBean. You can supply information such as names, port values, locations, and purposes with a few property values.

In this case, the `ObjectName` for the `HelloWorld` MBean looks like this:

```
"HelloAgent:name=helloWorld1"
```

Now that you have an `ObjectName` instance, you will be able to identify and find the MBean once it is registered.

### **Registering and starting the HTML adapter**

- ④ As previously mentioned, the agent creates an `ObjectName` for the adapter and registers the adapter in the `MBeanServer` object. Because they are MBeans, each adapter can choose to expose as many attributes as necessary for configuration by a management application.

Even though at this point in the code the adapter has been created and registered, management applications still cannot contact it. For clients to make use of the HTML adapter, it must be started. To start the adapter, you call its `start()` method. The `start()` method tells the adapter MBean to begin listening for HTTP clients on the default port of 9092. The `HelloAgent` is now ready to receive client calls.

### **2.3.2 More about object names**

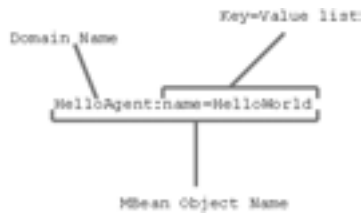
We briefly described an `ObjectName` value in the previous section. Having completed the code examination, let's return our focus to the `ObjectName` class. As you noticed, object names have a specific structure that must be followed when constructing a value.

Figure 2.4 shows the structure of an `ObjectName` value.

#### **Domain names**

Domain names provides context for the agent in relation to other agents. For example, an agent might be created to contain MBeans managing resources on a particular computer. In this case, the domain could be the computer's hostname. A domain name does not have to be a meaningful value like a computer's

hostname, but as a rule of thumb, you should try to provide some meaning in the name. That way, you will be able to look at an `ObjectName` value and possibly understand something about its `MBean`.



**Figure 2.4** The structure of an `ObjectName` value

`MBeans` of a certain domain can be registered on an `MBeanServer` containing a different domain name. This situation is acceptable because domain names do not impose any rules or constraints on which `MBeans` can be registered on an `MBeanServer` object.

### **Key/value property list**

The key/value list portion of the object name is a set of comma-separated property values that provide the mechanism for uniquely identifying `MBeans` within an `MBeanServer`. The properties do not have to be actual `MBean` attributes; the only requirement is that they are unique when compared to other instances of `ObjectName`. In each `ObjectName`, you must specify at least one property value that makes it distinct from all other `ObjectName` instances in an `MBeanServer`.

The `ObjectName` class provides three constructors that build the name `String` with various parameters. In the `HelloAgent` class, you create an `ObjectName` as follows:

```
helloWorldName = new ObjectName( "HelloAgent:name=helloWorld1" );
```

This `ObjectName` uniquely identifies the `HelloWorld` instance by giving it an attribute of `name` and value `helloWorld1`. If you register any other `MBeans`, you cannot use this property value on its own again; instead, you'll need an additional property combined with it.

### **Registering object name conflicts**

You can think of the registry function like a more complex `Hashtable`. You put objects into the table and associated them with a key. The key in this case is the `ObjectName` object. To register the `MBean`, the `HelloAgent` class invokes the

As you can tell from figure 2.4, the domain name does not even have to be specified. If it's left blank, the `MBeanServer` provides a default domain name. The same is true for the `MBeanServerFactory` class. If you use `createMBeanServer()` without a domain name parameter, the factory will provide you with an `MBeanServer` with a default domain.

Right about now, you may have noticed that both `MBeanServer` objects and `MBeans` (through the `ObjectName`) are associated with a domain. In fact,

`registerMBean()` method of the `MBeanServer` object. If the `ObjectName` is not unique, the `MBeanServer` will throw a `javax.management.InstanceAlreadyExistsException` exception, indicating that an `MBean` was already registered with an identical `ObjectName`. The `MBean` server does not compare actual `MBean` object values for equality—only their associated object names.

## 2.4 Running the agent

---

Let's review what you have accomplished so far. First, you created your first `MBean`, contained in the `HelloWorld` class. It exposes a single attribute—its greeting—as a manageable resource. (Recall that a manageable resource is any resource that can be encapsulated by an `MBean` to provide access and/or configuration.)

Next, you created the `HelloAgent` class, which is a simple `JMX` agent. The agent will contain your `MBean` and provide you with a mechanism for managing it. That leaves you with one more task to do: compile, run, and contact this agent.

### 2.4.1 Compiling the agent

To get the agent started, you need to compile your Java source code and execute the `HelloAgent` class. To compile your classes, execute the following command after ensuring your environment is set up correctly (`CLASSPATH` and so forth):

```
javac jmxbook\ch2\*.java
```

### 2.4.2 Running the agent

The following command will run the `HelloAgent`:

```
java jmxbook.ch2>HelloAgent
```

After executing these commands, your agent should be running. The command prompt will not return, because the `HelloAgent` process does not exit. However, you should see the output “`HelloAgent is running`”, indicating that the agent has started.

### 2.4.3 Contacting the agent

To contact the running `HelloAgent`, you need to use an `HTML` client. Any web browser will do the trick. For this example, the `HTML` adapter of the `HelloAgent` defaults to listening on port 9092. If you do not have that port available, go back to the `HelloAgent` code, add the following line after the adapter's constructor is called, and add your own port value:

```
adapter.setPort( [port value] );
```

Make the port a value that is available for use. For the remainder of this book, we will use the HTML adapter on port 9092.

Once you have a valid port value, open your browser and point it to `http://localhost:9092`. (If you have specified a different port, be sure to use that one instead of 9092.)

The HTML adapter running in your agent is now listening on the specified port for HTTP requests. When you open your browser to that address, the adapter responds by sending back HTML. You can now manage your agent by interacting with the adapter via this HTML.

Before continuing, you should congratulate yourself: you have successfully created your first MBean and agent. The next section will walk you through communicating with your agent using the HTML adapter.

## 2.5 Working with the HTML adapter

---

Now that you have the HTML adapter up and running on an agent that contains an MBean, it is time to connect and see what it provides for you. The HTML adapter provides access to a JMX agent through an HTML client (any web browser will do). It contains three main pages:

- *Agent View*—The Agent View is the first page you will see; it provides a summary of the MBeans contained within the agent. From this page you can filter the MBean list to provide more refined views.
- *MBean View*—This page provides details about a specific MBean. From this page you can set and get MBean attributes and invoke MBean operations.
- *Admin View*—This page enables you to register new MBeans on the agent.

### 2.5.1 Agent View

After contacting the agent with your web browser, the Agent View page appears (figure 2.5). Agent View is an HTML page received from the agent’s HTML adapter; it shows you all the registered MBeans in this agent, representing them by presenting their `ObjectName` values. From this page, you can get to the MBean View or Admin View HTML page. Before checking out the other views, notice the Filter by Object Name field at the top of the page. Right now it displays `*:*`, which tells the agent to return a list of all the MBeans it contains. The MBean count on this page displays the total MBeans returned by each search.

This filter form allows you to filter the MBean list by partial or whole object names. For instance, if you type `HelloAgent:name=helloWorld1` into the field,



**Figure 2.5**  
The Agent View page  
presented by the  
HTML adapter

the list will only show your `HelloWorld` MBean. You can enter partial object names by following the rules listed in table 2.2.

**Table 2.2** Agent View filtering rules

MBean filter rule	Example
Use the <code>*</code> character as an alphanumeric wildcard for multiple characters, and as a wildcard for key/value pairs.	<code>*:name=helloWorld1,*</code>
Use the <code>?</code> character as a wildcard for one character.	<code>??Agent:name=helloWorld1</code>
If you do not specify a domain name, the filter assumes you mean the default domain. You must specify at least one key/value pair (or use <code>*</code> ).	<code>*:*</code>
Partial domain names are valid, but you cannot specify partial key/value pairs (for the key or the value).	<code>??Agent:name=helloWorld1</code>
All keys must be matched exactly or use a wildcard.	<code>??Agent:*</code>
Key/value pairs can be specified in any order.	

Try filtering the MBean list on your own; doing so will help you get the hang of the `ObjectName` format. When you are done, return the filter to `*:*` so you can view all MBeans.

Note that the list includes an MBean you did not register in section 2.4: the `MBeanServerDelegate` MBean. The `MBeanServerDelegate` is an MBean created by the `MBeanServer` to handle certain tasks—specifically, sending out notifications for the MBean server. The `MBeanServer` registers this MBean with a different domain in order to keep it separated from any others that will be registered.

Now that all the MBeans are visible, click the `MBeanServerDelegate` MBean link; it will take you to the MBean View presented by the HTML adapter.

### 2.5.2 MBean View

MBean View is another HTML page received from the HTML adapter that shows information about the MBean you clicked. MBean View presents you with all the details of the selected MBean, including the information shown in table 2.3.

**Table 2.3** The elements of MBean View

MBean detail	Description or example
Class name	Main class of the MBean, such as <code>HelloWorld</code> .
Object name	Object name of the MBean, such as <code>HelloAgent:name=helloWorld1</code> .
Description	Description of the MBean. For Standard MBeans, the <code>MBeanServer</code> creates the description.
Attributes table	Lists the exposed attributes of the selected MBean, including the type, access, and value if possible. The attributes table also allows you to change writable attributes.
Exposed operations	List of operations exposed by the MBean. From here you can invoke an operation.
Reload Period	Tells the <code>MBeanServer</code> if it needs to reinstantiate this MBean, and if so, how often.
Unregister button	Tells the <code>MBeanServer</code> to unregister this MBean.

Figure 2.6 depicts the MBean view of the `MBeanServerDelegate` MBean.

Look at the table of MBean attributes in figure 2.6, and notice the values in the Access column. Currently, all rows contain the value RO, which stands for Read Only. Other possible values are WO (Write Only) and RW (Read/Write) access. As you might suspect, RO implies that the MBean’s Java interface has provided only a getter method for an attribute. WO access implies that there is only a setter method, and RW implies that both a setter and a getter exist for this attribute.

The HTML adapter is using the reflection API to examine the method names from the interface. It removes the `get` or `set` of each method name and creates the attribute name from the remaining method name portion. Remaining methods (those without `get` or `set` at the start of their names) go into the Operations section of the MBean View.

By looking back at figure 2.6, you see that the `MBeanServerDelegate` exposes only read-only attributes. These attributes describe the reference implementation being used and which version of the JMX specification it implements.

Let’s go back to the Agent View by clicking the Back to Agent View link. This time, select your `HelloWorld` MBean. Figure 2.7 shows what you should see.

The view of the `HelloWorld` MBean is displayed exactly like that of the `MBeanServerDelegate` MBean, except for two important differences. Remember that

**MBean View** [J2SE4.0Server]

- MBean Name: javax.management.MBeanServerDelegate
- MBean Java Class: javax.management.MBeanServerDelegate

Refresh Period in seconds: 0

[Back to Agent View](#)

**MBean description:**  
Information on the management interface of the MBean.

**List of MBean attributes:**

Name	Type	Access	Value
ImplementationName	java.lang.String	R/O	J2SE 4.0
ImplementationVendor	java.lang.String	R/O	Sun Microsystems
ImplementationVersion	java.lang.String	R/O	1.0
MBeanServerId	java.lang.String	R/O	Agent_0021345T10009
SpecificationName	java.lang.String	R/O	Java Management Extensions
SpecificationVendor	java.lang.String	R/O	Sun Microsystems
SpecificationVersion	java.lang.String	R/O	1.0 Final Release

**List of MBean operations:**  
No Operations

Figure 2.6 MBean View presented by the HTML adapter

**MBean View** [J2SE4.0Server]

- MBean Name: HelloAgentNameHelloWorld
- MBean Java Class: javax.management.HelloWorld

Refresh Period in seconds: 0

[Back to Agent View](#)

**MBean description:**  
Information on the management interface of the MBean.

**List of MBean attributes:**

Name	Type	Access	Value
Greeting	java.lang.String	R/W	Hello World I am a Standard MBean

**List of MBean operations:**  
Description of `printGreeting`  
void

Figure 2.7 The HelloWorld MBean View

you wrote the `HelloWorld` MBean to contain a single exposed attribute: its greeting. That means the greeting attribute of the `HelloWorld` MBean is accessible for both reading and writing, and the attribute table of the MBean View includes a text field allowing you to enter a value. Clicking the Apply button commits any changes. Go ahead and change the value for the greeting, and then click Apply. The page reloads, and the text field displays the current value of `Greeting`, which reflects the changes you just made.

Now look at the MBean Operations section, and you will see one available operation. The MBean View constructs each operation as a button labeled with the name of the method. For the `HelloWorld` MBean View, you see a button with `printGreeting` as a label. That is the remaining method from the `HelloWorldMBean` interface. Just before the button, you see `void`, which is the return type of the method. If this method had any input parameters, you would see a text field for each input value.

---

**NOTE** The HTML adapter can provide input only for certain types of parameters. It supports only `Strings`, primitive types, and the standard classes related to the primitive types, such as `java.lang.Integer`.

---

When you click the `printGreeting` button, you will see two things happen. First, the web browser navigates to a page indicating that the method succeeded and did not return a value. Second, if you look at the output of your agent, you should see the update value you entered for the `Greeting` attribute.

Congratulations—you have successfully managed the `HelloWorld` MBean. There is only one more page to examine from the HTML adapter: the Admin View. Go back to the Agent View and click the Admin button to go to the Admin View of the `HelloAgent`.

### 2.5.3 Admin View

Using the first two HTML pages, you can configure and query MBeans registered in the agent. However, what if you want to add additional MBeans to the agent without writing more code? The Admin View is an HTML page presented by the HTML adapter that gives you access to the agent’s MBean server in order to remove or add MBeans. From this page, you can specify an `ObjectName` value and associate it with a Java class that is an MBean. The MBean server will construct and register an MBean corresponding to your input. The Admin View presents four text fields (see figure 2.8):



Figure 2.8 The Admin View presented by the HTML adapter

- *Domain*—The HTML adapter defaults the Domain field to the domain of the agent. Currently, it shows `HelloAgent`, which is your domain. This is the first part of the object name.
- *Keys*—The Keys field requires input in the form `[name=value],*`. This field represents the key/value portion of an object name.
- *Java Class*—This field requires a full Java class name of the class of the MBean you want to create.
- *Class Loader*—The Class Loader field is the only field that is optional. You can specify a class loader for the MBean server to use when attempting to load the Java class specified in the previous field.

To get a good understanding of this page, let's use it to create some more MBeans. As you will see in the next section, you can have many MBeans of the same type in one `MBeanServer`, as long as their object names are unique.

#### 2.5.4 Registering/unregistering MBeans on the HelloAgent

Let's load another instance of the `HelloWorld` MBean into the agent by using the Admin View. Leave the domain value as `HelloAgent`. Type in **name=helloWorld2** for the Keys field. For the Java Class field, type **jmxbook.ch2.HelloWorld**, the implementation of the `HelloWorldMBean` interface.

---

**NOTE** Any class you specify in the Java Class field must be accessible to the `MBeanServer` of the agent. For the `HelloAgent`, this requirement means the input Java Class value must be in its local `CLASSPATH`.

---

Leave the Class Loader field empty, to tell the `HelloAgent` MBean server to use the default class loader to find the Java Class value. The agent will use the values you have entered to create an object name like

```
HelloAgent:name=helloWorld2
```

With all the values in place, you are ready to create this new `HelloWorld` instance. At this point, you have three Action options to choose from in the drop-down list:

- *Create*—Tells the `MBeanServer` to create an MBean using a no-argument constructor
- *Unregister*—Works only if you have specified an `ObjectName` of an existing MBean registered in the agent
- *Constructors*—Loads the specified MBean class and presents you with a list of constructors to use in creation of the MBean

These options are similar to the Operations section of the MBean View. For now, choose the Create option.

Click the Send Request button, and you will see a success message telling you the agent created and registered the new MBean. Go back to the Agent View and verify this using the list of MBeans.

### **Using constructors with arguments**

Remember that your `HelloWorldMBean` implementation class, `HelloWorld`, has two constructors: the default constructor and a constructor that takes an initial value for the `Greeting` attribute. Let’s register a final instance of the `HelloWorld` MBean by using this second constructor. To do so, perform the following steps:

- 1 Go back to the Admin View and enter appropriate values to create a new `HelloWorld` MBean. Be sure to enter a unique Key value (such as **name=helloWorld3**).
- 2 This time, select Constructors from the Action list and click the Send Request button. You will see a list of constructors (in this case, two), one of which displays a text field for its single input parameter.
- 3 Type a value for the `Greeting` attribute, and then click the Create button associated with the constructor. If you entered valid data for the object name and class fields, you will see the “creation successful” message again.
- 4 Go back to the Agent View to verify that the MBean list now contains three instances of the `HelloWorld` MBean.

## 2.6 Using MBean notifications

After creating and registering your own MBean in the previous section, you already have enough knowledge to start working with JMX. You have learned how to create a Standard MBean, how to add it to a simple JMX agent, and how to manage that agent by using the HTML adapter. However, you are still missing a key ingredient: notifications.

JMX notifications are Java objects used to send information from MBeans and agents to other objects that have registered to receive them (see figure 2.9). Objects interested in receiving events are notification listeners—they implement the `javax.management.NotificationListener` interface.

Notifications are an important piece of JMX because they allow for the transmission of events. JMX events can be anything from the changing of an MBean attribute to the registration of a new MBean on an MBean server.

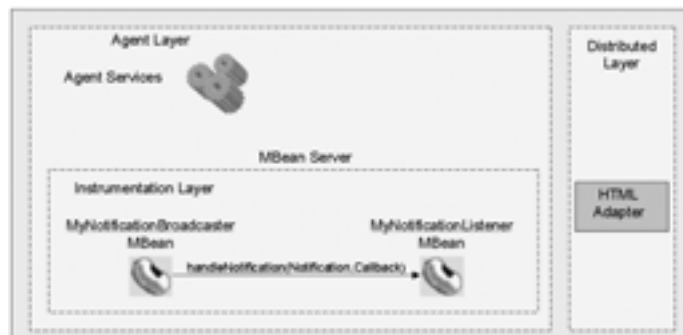
To give you a quick introduction to notifications, you'll add them to the `HelloWorld` MBean in this section. In chapter 6, we'll cover the notification model in depth.

### 2.6.1 Adding notification code to the `HelloWorld` MBean

For the `HelloWorld` MBean to send notifications, it needs to allow objects interested in receiving notifications to register for them. JMX supports two mechanisms for MBeans to provide listeners to register for notifications:

- Implement the `javax.management.NotificationBroadcaster` interface
- Extend the `javax.management.NotificationBroadcasterSupport` class (which in turn implements the `NotificationBroadcaster` interface)

The advantage of implementing the interface is that it frees your class from being tied to a particular super class. The advantage of extending the broadcaster



**Figure 2.9**  
Notification being sent  
to a registered listener  
from an MBean

support class is that you do not have to write code for the broadcaster interface. If your MBean does not need to extend a class, then have it extend the `NotificationBroadcasterSupport` class and reuse that implementation. The `HelloWorld` class does not need any special super class, so you are free to extend the broadcaster support class, `NotificationBroadcasterSupport`, as shown in listing 2.3.

### Listing 2.3 HelloWorld.java

```

package jmxbook.ch2;

import javax.management.*;

public class HelloWorld extends NotificationBroadcasterSupport
implements HelloWorldMBean ❶ Extend NotificationBroadcasterSupport class
{
    public HelloWorld() ❷ Define two public constructors
    {
        this.greeting = "Hello World! I am a Standard MBean";
    }

    public HelloWorld( String greeting ) ❷ Define two public constructors
    {
        this.greeting = greeting;
    }

    public void setGreeting( String greeting )
    {
        this.greeting = greeting;

        Notification notification = new Notification(
            "jmxbook.ch2.helloWorld.test", this, -1,
            System.currentTimeMillis(), greeting ); ❸ Create javax.management.Notification object

        sendNotification( notification ); ← Send notification
    }

    public String getGreeting()
    {
        return greeting;
    }

    public void printGreeting()
    {
        System.out.println( greeting );
    }

    private String greeting;
}
//class

```

- ❶ You change the declaration of the `HelloWorld` class by extending the `NotificationBroadcasterSupport` class. This super class provides the MBean with methods that allow other objects to register as notification listeners and allow the MBean to send notifications. The super class implements the `javax.management.NotificationBroadcaster` interface examined after this code discussion.
- ❷ For this example, you are sending only basic notifications. At this step in the code, you create a `Notification` object with the constructor

```
public Notification(java.lang.String type, java.lang.Object source,
                  long sequenceNumber, long timeStamp,
                  java.lang.String message)
```

The parameters of this constructor are listed in table 2.4.

**Table 2.4 The Notification constructor parameters**

Parameter	Description
<code>java.lang.String type</code>	Dot-separated <code>String</code> value used to identify the notification. Used as a short description of the purpose and meaning for the notification.
<code>java.lang.Object source</code>	The MBean that generated this notification. This will be either the object reference or the <code>ObjectName</code> of an MBean.
<code>long sequenceNumber</code>	A number that identifies this notification in a possible sequence of notifications.
<code>long timeStamp</code>	A timestamp of the creation of the notification.
<code>java.lang.String message</code>	A <code>String</code> value containing a message from the notification source.

- ❸ By extending the `NotificationBroadcasterSupport` class, you not only gain the implementation of the `NotificationBroadcaster` interface, but also inherit the `sendNotification()` method. Your `HelloWorld` MBean can use this convenience method when it needs to send a notification, and the super class will send it to all appropriate listeners. Appropriate listeners are those that have registered with the MBean and whose filter object accepts the particular type of notification.

### **Examination of the NotificationBroadcaster interface**

The previous example used the `NotificationBroadcasterSupport` class. This class provides subclasses with an implementation of the `NotificationBroadcaster` interface. The super class implements the `NotificationBroadcaster` interface shown next:

```
public interface NotificationBroadcaster
{
    public void addNotificationListener(
        NotificationListener listener,
```

```

        NotificationFilter filter,
        Object handback )
        throws IllegalArgumentException;

    public MBeanNotificationInfo[] getNotificationInfo();

    public void removeNotificationListener(
        NotificationListener listener )
        throws ListenerNotFoundException;

}

```

MBeans implementing this interface provide other objects with a mechanism to register for notifications by using the `addNotificationListener()` method. This method accepts a `NotificationListener` object, a `NotificationFilter` object, and a handback object as parameters.

The `NotificationListener` parameter is an object that implements the `NotificationListener` interface, which specifies a `handleNotification()` method. This method will be invoked as a callback when a notification needs to be delivered to a listener.

The `NotificationFilter` parameter is an optional argument that will allow the MBean to filter which notifications to send to the listener based on the listener’s preferences created in the filter. The handback argument is sent back to the client each time a notification is delivered.

Notice the similarity between this notification registration and delivery mechanism and the Java event model of listener registration.

## 2.6.2 Changes to the *HelloAgent* class

In order to send notifications, you need to have something to receive them. For this small notification example, you’ll make your `HelloAgent` class act as a notification listener. You need to modify your code in a few ways. First, the `HelloAgent` class needs to implement the `NotificationListener` interface. The `HelloAgent` will still create and register both the HTML adapter and the `HelloWorld` MBean. After it has created the MBeans, it can now register with the `HelloWorld` MBean as a notification listener interested in receiving notifications. The code changes for the `HelloAgent` class appear in listing 2.4 in bold.

Listing 2.4 `HelloAgent.java`

```

package jmxbook.ch2;

import javax.management.*;
import com.sun.jdmk.comm.HtmlAdaptorServer;

public class HelloAgent implements NotificationListener

```

**Implement  
NotificationListener  
interface** **1**

```

{
    private MBeanServer mbs = null;

    public HelloAgent ( )
    {
        mbs = MBeanServerFactory.createMBeanServer( "HelloAgent" );
        HtmlAdaptorServer adapter = new HtmlAdaptorServer();
        HelloWorld hw = new HelloWorld();

        ObjectName adapterName = null;
        ObjectName helloWorldName = null;

        try
        {
            adapterName = new ObjectName(
                "HelloAgent:name=htmladapter,port=9092" );
            mbs.registerMBean( adapter, adapterName );
            adapter.setPort( 9092 );
            adapter.start();

            helloWorldName = new ObjectName(
                "HelloAgent:name=helloWorld1" );
            mbs.registerMBean( hw, helloWorldName );

            hw.addNotificationListener( this, null, null ); ❷ Register to
                                                                    receive
                                                                    notifications
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }
    //constructor
    public void handleNotification(
        Notification notif, Object handback ) ❸ Implemented
                                                                    from listener
                                                                    interface
    {
        System.out.println( "Receiving notification..." );
        System.out.println( notif.getType() );
        System.out.println( notif.getMessage() );
    }

    public static void main( String args[] )
    {
        HelloAgent agent = new HelloAgent();
    }
}
} //class

```

- ❶ The first addition to the agent is the inclusion of the `NotificationListener` interface. Recall that this interface declares a single method, `handleNotification()`, which will be called when a notification is being delivered from a source the listener has registered with.

- ② After registering the MBean with the MBean server, the `HelloAgent` class adds itself to the `HelloWorld` MBean as a listener. To do this, it passes itself as the `NotificationListener` parameter to the MBean’s `addNotificationListener()` method (inherited from its super class, `NotificationBroadcasterSupport`).
- ③ As mentioned earlier, in order to receive notifications, an object must implement the `NotificationListener` interface. The interface declares a single method, `handleNotification()`, which is a callback method invoked by the sender to deliver notifications to the listener. For this implementation, the `HelloAgent` class just prints out some of the members of the incoming notification.

### Getting results

To test the notification example, you need to compile the `HelloAgent.java` and `HelloWorld.java` files and execute the resulting `HelloAgent` class. (Look back at section 2.5 for a reminder.) Once the agent is running, open your web browser to `http://localhost:9092` to see the Agent View of your `HelloAgent` class. To perform the test and receive a notification, follow these steps:

- 1 Navigate to the MBean View of the `HelloWorld` MBean by clicking on the corresponding object name in the list. Look back at section 2.5.2 for a refresher.
- 2 The `HelloWorld` MBean sends a notification when its greeting is changed, so enter a new value and click Apply. For example, enter **I have changed my greeting**.
- 3 Look at the output window of your `HelloAgent`. You should see the following:

```
Receiving notification...
jmxbook.ch2.helloWorld.test
I have changed my greeting
```

The output contains your printed message, “Receiving notification...”, the notification type, and the message contained in the notification.

## 2.7 Summary

---

This chapter gave you some hands-on experience with much of the JMX framework. In this chapter you developed a manageable resource, and created and ran a simple JMX agent. We discussed how to register an MBean, ensure that it has a unique name, and create an MBean server.

In addition, you worked with the HTML adapter that comes with the Sun reference implementation. By constructing the `HelloWorld` example, you should now understand that MBean development is simple from the JMX point of view.

MBeans expose resources with just a few lines of code. Part 2 of this book covers instrumenting resources by walking you through numerous MBean examples.

Finally, to round out your JMX introduction, we gave you a crash course on JMX notifications. We will discuss notifications in greater detail in chapter 6 and make a stronger case for why notifications are an essential part of managing resources.

In chapter 3, you'll begin to develop a JMX agent that you'll enhance throughout the remainder of the book. This agent will be used for many of the examples in other chapters.