Managing Java Applications with JMX

Java™ Management Extensions

O'REILLY®

J. Steven Perry

Java™ Management Extensions

J. Steven Perry

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 2

Standard MBeans

By far the most straightforward type of MBean to create, standard MBeans are a logical starting place for our voyage into the world of JMX. In this chapter, we will begin by defining a management interface, then we will look at the design patterns we must use when building standard MBeans. Next, we will discuss some of the issues involved when using inheritance among standard MBean classes. Then we will look at some common pitfalls of using standard MBeans that might leave you scratching your head, wondering why your MBeans aren't working as expected. Finally, we will discuss some advanced topics and things to consider when creating standard MBeans. This chapter includes several examples from the sample application that is used throughout this book. When you have completed this chapter, you should be able to create standard MBeans and understand the issues involved in doing so.

What Is a Management Interface?

The idea of a management interface revolves around the notion of an application resource, which is any abstraction within an application that provides value. A resource can be, for example, a business component of the application, an infrastructure component such as a cache or queue, or even the application itself. With respect to JMX (and more broadly, system management), only those resources that must be managed are significant. Each resource that is to be managed must provide a management interface, which consists of the attributes and operations it exposes so that it can be monitored and controlled by a management application.

For example, suppose a resource we wish to manage in our application is a queue. A queue is generally used to temporarily store a logical unit of work provided by a supplier until that unit of work is removed from the queue by a consumer for further processing. This is typically done asynchronously, via a multithreaded design, so the queue must also be thread-safe. Let's suppose there is a single supplier thread in the application and a single consumer thread, and that the queue is thread-safe, so that the supplier will wait to add an item to the queue if it is full (and conversely, the consumer will wait on the queue to remove an item if the queue is empty). In monitoring the queue, we want to know two things:

- How long has the supplier waited (i.e., the accumulated wait time) to add an item to the queue because the queue is full?
- How long has the consumer waited to remove an item from the queue because the queue is empty?

Because these two pieces of information are important to us for monitoring the application, it makes sense that we expose them as two attributes on the management interface of the queue. Of course, the queue must be written so that this information is captured, so that it can be exposed in the first place! But that is very straightforward to do and will be handled by our application's `Queue` class.

> I won't spend a lot of time explaining all of the attributes and operations exposed on the management interface of the queue, because the goal of this chapter is to illustrate the mechanics and theory of standard MBeans, not how to design queues.

One other thing to consider is whether the values of these attributes are in milliseconds (the clock tick count, on most systems). If so, and if our application is long-running, we should probably use a `long` as the data type for these attributes to allow them to contain very large values.

Now that we have decided to expose these two attributes, we must decide whether to make them read-only, write-only, or read/write when we expose them on the management interface. It doesn't make much sense for these attributes to be write-only (because then we could only set them and not look at their values at any particular point in time), so we rule that option out right away. Making these attributes readable makes sense, but should we allow the management application to set their values? We could allow the management application to reset both of these values to zero by exposing an operation to handle this action, thus preventing a user of the management application from setting these values to something unreasonable.

The management interface for our queue now has the following:

- A read-only attribute whose value is the total accumulated time spent by the supplier waiting to add a unit of work to the queue because the queue is full
- A read-only attribute whose value is the total accumulated time spent by the consumer waiting to remove a unit of work from the queue because the queue is empty
- An operation that resets both attributes to zero

We may also want to be able to manage the size of the queue. To allow this, we can define an attribute on the queue and expose that attribute on the management interface as a read/write attribute. This allows us to view the current size of the queue and to modify that value to tweak the queue for maximum performance at runtime. In addition, we might be interested in the total number of work units processed by the queue, so that we can get an idea of the throughput of the application with respect to any particular instance of our queue. As long as it makes sense and fits in with the design of the queue, the sky's the limit on what we can expose on our queue's management interface. Table 2-1 summarizes the attributes we will expose on our `Queue` resource.

Table 2-1. Attributes exposed for management on the Queue class

Name	Data type	Read/write
Add Wait Time	`long`	Read-only
Remove Wait Time	`long`	Read-only
Queue Size	`int`	Read/write
Number of Items Processed	`long`	Read-only
Queue Full	`boolean`	Read-only
Queue Empty	`boolean`	Read-only
Suspended	`boolean`	Read-only
Number of Suppliers	`int`	Read-only
Number of Consumers	`int`	Read-only

These attributes will be discussed in detail in the next section, where we will actually implement the queue's management interface as a standard MBean.

Next, let's consider the operations to expose on the management interface of the `Queue` class. We touched briefly on the reset operation earlier in this chapter. Other operations we may want to include offer the management application the ability to suspend and resume activity in the queue. This allows the management application to halt processing so that, for example, an operator can look at a "snapshot" of what is happening inside the queue. It may also be helpful for the operator to be able to turn on and off tracing. Table 2-2 summarizes the operations on our queue's management interface.

Table 2-2. Operations exposed for management on the Queue class

Name	Purpose
Reset	Resets the state of the queue
Suspend	Suspends activity in the queue; all suppliers and consumers sleep until Resume is called
Resume	Signals to sleeping suppliers and consumers that activity may continue
Enable Tracing	Turns on any tracing done by the queue
Disable Tracing	Turns off tracing

Now that we have defined the management interface for our `Queue` class, it's time to see how to instrument our class as a standard MBean using the design patterns in the JMX specification.

How Do Standard MBeans Work?

In this section, we will learn how to instrument a Java class as a standard MBean. We will first look at how to describe the management interface according to the JMX design patterns for standard MBeans. Then we will look at how to implement the MBean interface on the `Queue` class touched on earlier in this chapter. Many examples will be provided. It is here that we will examine all of the classes that make up the application, showing inheritance patterns and other cool standard MBean miscellany. We will also look at the `Controller` class's *main()* routine, which is what drives the application, and we will discuss how to register MBeans with the MBean server, how to register and use the HTML Adaptor server, and how to build and run the example.

Describing the Management Interface

JMX provides us with a set of patterns to follow when instrumenting our application resources as standard MBeans. If we follow these patterns exactly as they are set out in the specification, our standard MBeans are said to be *compliant*. If we don't correctly follow the patterns, the MBean server (part of the reference implementation; we'll discuss the MBean server later in this chapter) will declare our MBean as *non-compliant* by throwing a `javax.management.NotCompliantMBeanException` at the agent that attempts to register the MBean. However, it is possible for us to correctly follow the patterns but still not expose the correct management interface on our standard MBean. We will also look at that case in this section.

There are three patterns you must follow when instrumenting your resources as standard MBeans:

- The management interface of the resource must have the same name as the resource's Java class, followed by "MBean"; it must be defined as a Java interface; and it must be implemented by the resource to be managed using the `implements` keyword.
- The implementing class must contain at least one public constructor.
- Getters and setters for attributes on the management interface must follow strict naming conventions.

Each of these patterns is discussed in detail in this section.

Pattern #1: Defining, naming, and implementing the MBean interface

The management interface must be defined using the Java `interface` keyword, it must have `public` visibility, and it must be strictly named. Earlier in this chapter, we looked at the thought process we might go through to define a management interface for a queue. Suppose the name of this class is `Queue`. Its standard MBean management interface must be defined as:

```
public interface QueueMBean {
// management interface goes here. . .
}
```

The `Queue` class, in turn, must implement the `QueueMBean` interface using the Java `implements` keyword:

```
public class Queue implements QueueMBean {
// implementation of QueueMBean
// and other stuff here. . .
}
```

The name of the MBean interface is case-sensitive. For example, `QueueMbean` is not the same as `QueueMBean`. Of course, the compiler will help you if you "fat-finger" the spelling of the interface in either the interface definition or the implementation. However, if you use the same misspelling in both, the compiler will chug merrily along and produce perfectly runnable bytecode. Only when you attempt to register your MBean will you receive a `NotCompliantMBeanException` exception!

The management interface is contained in its own *.java* file and must have the same name as its corresponding interface. Thus, every standard MBean requires at least two source code files: one for the interface and one for the class that implements the interface.

Another example from the application we use throughout this book is the `Worker` class. Its management interface is defined as:

```
public interface WorkerMBean {
// . . .
}
```

The `Worker` class, in turn, implements this interface as:

```
public class Worker implements WorkerMBean {
// . . .
}
```

The JMX specification states that the class that implements the MBean interface must be declared `public` and be a concrete (i.e., not abstract) class. However, using the JMX 1.0 RI, I was able to instantiate, register, and manage an MBean with only package-level visibility. This is most likely an oversight in the RI. You should not count on being able to do this in future versions of the RI, or in other JMX implementations, because this behavior is not supported by the specification.

Pattern #2: Provide at least one public constructor

The class that implements the MBean interface must have at least one constructor declared with `public` visibility. This class may have any number of public constructors, but it must have *at least* one. If you do not provide a constructor, the compiler will generate a no-argument constructor with `public` visibility. This will work fine for your MBeans, but I recommend that you explicitly declare a no-argument constructor for these cases, as your code will follow the rule and be more readable as well. Continuing with the code snippets from earlier, then, our Queue class would look like:

```
public class Queue implements QueueMBean {
  public Queue( ) {
    // do something here. . .
  }
  // other class methods and management interface
  // implementation. . .
}
```

However, the Queue class might not have a no-argument constructor at all:

```
public class Queue implements QueueMBean {
  // no no-arg constructor provided, that's okay. . .
  public Queue(int queueSize) {
    // do something custom here. . .
  }
  // other class methods and management interface
  // implementation. . .
}
```

and still be a compliant MBean, because it provides a public constructor.

Pattern #3: Attributes and how to name their getters and setters

When defining an attribute on the management interface, you must follow strict naming standards. If the attribute is readable, it must be declared on the interface (and subsequently implemented) as *getAttributeName()*, where *AttributeName* is the name of the attribute you want to expose, and take no parameters. This method is called a *getter*. Table 2-1 showed some of the attributes we plan to expose on the Queue class. As an example, we would define the Add Wait Time attribute on the management interface as:

```
public interface QueueMBean {
  public long getAddWaitTime( );
  // . . .
}
```

Notice the use of "camel case" in the naming of our attribute. If an attribute's name consists of multiple words, the words are placed together and the first letter of each word is capitalized. This is a fairly common practice and will be used throughout this book.

For boolean values, preceding the attribute name with "is" is a common idiom and one that is acceptable according to the JMX standard MBean design patterns. From Table 2-1, notice that we have a boolean attribute called Suspended. We would define this attribute on the management interface as:

```
public interface QueueMBean {
  public long getAddWaitTime( );
  // . . .
  public boolean isSuspended( );
  // . . .
}
```

If an attribute is writable, the naming pattern is similar to that for readable attributes, only the word "get" is replaced with "set," and the attribute takes a single parameter whose type is that of the attribute to be set. This method is called a *setter*. For example, Table 2-1 shows a readable and writable attribute called QueueSize. We would define this attribute on the management interface as:

```
public interface QueueMBean {
  public long getAddWaitTime( );
  // . . .
  public boolean isSuspended( );
  // . . .
  public int getQueueSize( );
  public void setQueueSize(int value);
  // . . .
}
```

There are two rules about setters:

- The setter can take only a single parameter. If you unintentionally provide a second parameter to what you thought you were coding as a setter, the MBean server will expose your "setter" as an operation.
- The parameter types must be the same for read/write attributes, or your management interface will not be what you expect. In fact, if you have a read/write attribute where the getter returns a different data type than the setter takes as a parameter, the setter controls. For example, suppose that I mistakenly coded the setter for QueueSize to take a short data type. My management interface would then look like:

  ```
  public interface QueueMBean {
    public long getAddWaitTime( );
    // . . .
    public boolean isSuspended( );
    // . . .
    public int getQueueSize( );
    public void setQueueSize(short value);
    // . . .
  }
  ```

  Strangely enough, what I have actually exposed is a single write-only attribute called QueueSize, of type short! Clearly, that is not what I intended. Of course, remember that with standard MBeans, the Java compiler can catch some of these mistakes for you. Let's say that I made this particular mistake on the interface definition, but on the implementing class I used the proper int type on my setter. The compiler would tell me that I should declare the implementing class abstract, because it doesn't define the setter that takes the short! That is one advantage of standard MBeans over other MBean types—the compiler can help you find mistakes before they turn into nasty bugs.

Using the information from Tables 2-1 and 2-2, the management interface is shown in Example 2-1.

Example 2-1. The QueueMBean interface

```
public interface QueueMBean {
  // attributes
  public long getAddWaitTime( );
  public long getRemoveWaitTime( );
  public int getQueueSize( );
  public void setQueueSize(int value);
  public long getNumberOfItemsProcessed( );
  public boolean isQueueFull( );
  public boolean isQueueEmpty( );
  public boolean isSuspended( );
  public int getNumberOfSuppliers( );
  public int getNumberOfConsumers( );
  // operations
  public void reset( );
  public void suspend( );
  public void resume( );
  public void enableTracing( );
  public void disableTracing( );
}
```

A word about introspection

Introspection literally means to "look inside" and is performed by the MBean server to ensure compliance on the part of your MBeans when they are registered. Because it is possible to write Java code that cleanly compiles and executes but does not follow the standard MBean design patterns we discussed earlier, the MBean server looks inside your MBean to make sure you followed the patterns correctly.

When your MBean is registered by the agent, the MBean server uses Java's reflection API to crawl around inside the MBean and make sure that the three design patterns we discussed earlier were followed. If they were, your MBean is compliant and its registration proceeds. If not, the MBean server throws an exception at the agent.

Introspection takes place only when your MBean is registered by the agent. Depending on the code paths your application takes when instantiating your MBean classes, the notification (via an exception) that one of your MBeans is not compliant will appear only when the MBean is registered.

Standard MBean Inheritance Patterns

As you are probably aware, inheritance in Java is achieved through the use of the extends keyword. When it comes to exposing a management interface, the MBean server's introspection enforces certain rules. There are some fundamental differences between Java inheritance and management interface inheritance. This section will spell out those differences.

With respect to inheritance, certain patterns are enforced by the MBean server at introspection time. If you are to successfully expose the intended management interface on your MBeans, it is important that you understand these patterns. While an MBean may inherit the public (and protected) attributes and operations of its parent class, it will not necessarily inherit its management interface.

There are five basic patterns of MBean inheritance. We will discuss each of them in this section. We will also introduce and explain the application MBean interfaces in this section, starting with the top of the inheritance hierarchy, `BasicMBean`. We will use UML diagrams to reduce ambiguity.

`BasicMBean` is the management interface that all MBeans in the inheritance graph will expose and in this section we will see exactly how to go about doing that. Along the way, I'll point out some areas to watch out for and offer some tips for avoiding potential mistakes. Example 2-2 shows the source listing for `BasicMBean`.

Example 2-2. The BasicMBean interface

```
package sample.standard;

public interface BasicMBean {
  // attributes
  public boolean isTraceOn( );
  public boolean isDebugOn( );
  public int getNumberOfResets( );
  // operations
  public void enableTracing( );
  public void disableTracing( );
  public void enableDebugging( );
  public void disableDebugging( );
  public void reset( );
}
```

Pattern #1: Basic inheritance

In the basic inheritance pattern, a class implements an MBean interface. This pattern is shown in UML notation in Figure 2-1.

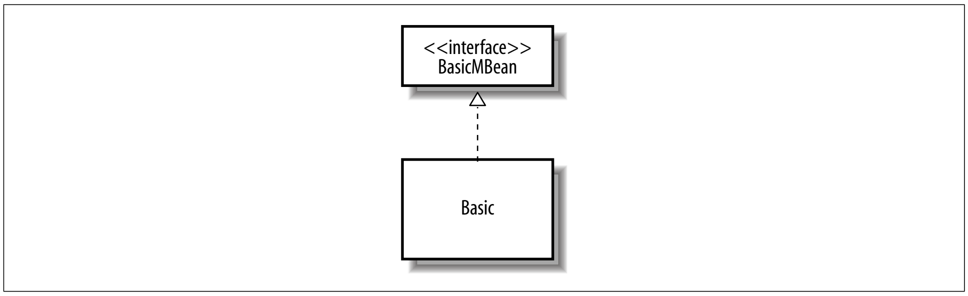


Figure 2-1. UML notation for pattern #1

In source code, this pattern is implemented using the `implements` keyword:

```
public class Basic implements BasicMBean {
// implementation of BasicMBean and other stuff. . .
}
```

Use of the `implements` keyword was explained in the previous section.

Pattern #2: Simple inheritance

With simple inheritance, one class extends another class that implements an MBean interface. This relationship is shown in Figure 2-2.

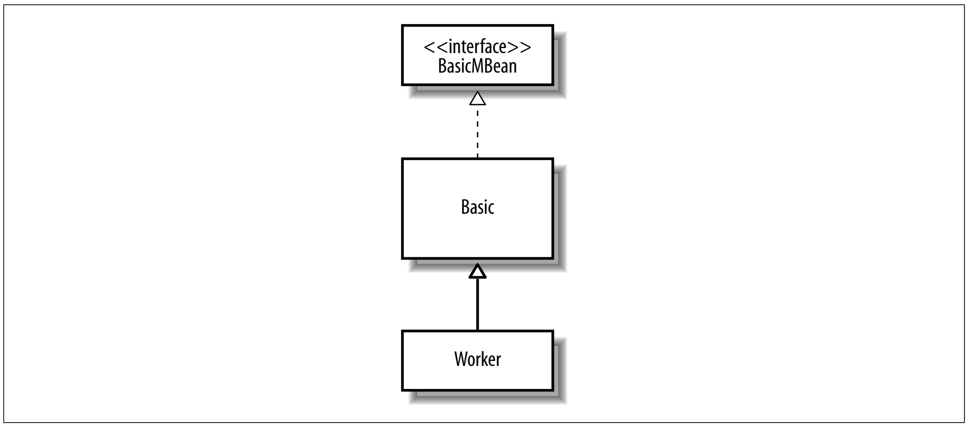


Figure 2-2. UML notation for pattern #2

In source code, this pattern is implemented using the `extends` keyword:

```
public class Worker extends Basic {
// implementation of Worker here. . .
}
```

In this pattern, the management interface exposed by `Worker` is `BasicMBean`. To a management application, `Worker` will appear to be a `BasicMBean`, complete with all of its attributes and operations. In other words, the management interface of `Worker` is the same as that of `Basic`.

Pattern #3: Simple inheritance with child class implementing an MBean interface

Of course, `Worker` could implement its own MBean interface and still extend `Basic`. The `WorkerMBean` interface is shown in Example 2-3.

Example 2-3. WorkerMBean management interface definition

```
package sample.standard;

public interface WorkerMBean {
```

Example 2-3. WorkerMBean management interface definition (continued)

```
  // attributes
  public String getWorkerName( );
  public void setWorkerName(String name);
  public int getWorkFactor( );
  public long getNumberOfUnitsProcessed( );
  public float getAverageUnitProcessingTime( );
  public boolean isSuspended( );
  // operations
  public void stop( );
  public void suspend( );
  public void resume( );
}
```

According to this pattern, `Worker` would continue to extend `Basic` but would now explicitly expose its own MBean interface:

```
public class Worker extends Basic implements WorkerMBean {
// implementation of WorkerMBean. . .
}
```

This pattern is shown in UML notation in Figure 2-3.

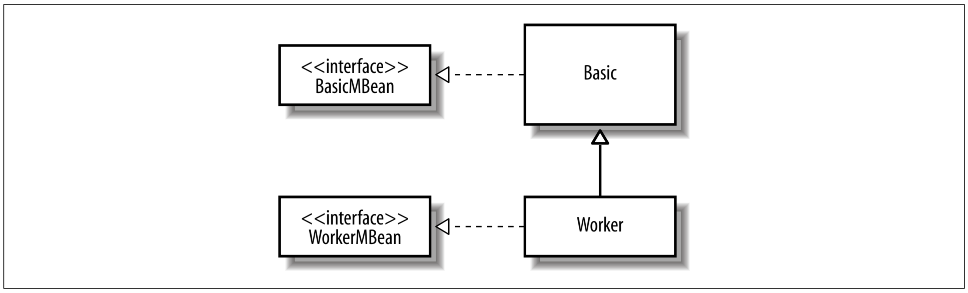


Figure 2-3. UML notation for pattern #3

After looking at pattern #3, you may think that the management interface exposed by `Worker` is the union of `BasicMBean` and `WorkerMBean`. However, this is not the case. When this pattern is used, the introspection performed by the MBean server proceeds up the MBean interface inheritance graph, not the implementing inheritance graph. In Example 2-3, we see that `WorkerMBean` stands alone at the top of the inheritance graph. Regardless of the fact that `Worker` extends `Basic` (which implements `BasicMBean`), the management interface exposed proceeds no further than `WorkerMBean`. Thus, when this pattern is used, the management interface exposed by `Worker` is that shown in Example 2-3. However, should a reference to `Worker` be obtained, methods and attribute getters and setters inherited from `Basic` can be invoked (Java inheritance still works!). These inherited methods and attributes are not available to a management application, though, because they are not on the management interface.

Note also that any time a class implements an MBean interface that is at the top of an MBean hierarchy, the management interface exposed by that class is that MBean interface, regardless of any attributes and methods available to that class through Java inheritance. For example, suppose I have a child class of `Worker` called `Supplier` and `Supplier` implements `SupplierMBean`, which extends nothing:

```
public interface SupplierMBean {
// management interface here. . .
}
.
.
public class Supplier extends Worker implements SupplierMBean {
}
```

Again, you might think that the management interface exposed by `Supplier` is the union of `BasicMBean`, `WorkerMBean`, and `SupplierMBean`. However, this is not the case. Recall that in the earlier example where `Worker` extended `Basic` but implemented `WorkerMBean`, the management interface exposed was `WorkerMBean`. Similarly, in this case, the management interface exposed by `Supplier` is `SupplierMBean`. Java inheritance still works, and a reference to `Supplier` will give the holder of that reference access all the way up the inheritance graph, but a management application will have access only to those methods on the management interface exposed by `Supplier`.

Pattern #4: Simple inheritance with MBean inheritance

If `WorkerMBean` were to extend `BasicMBean`, pattern #3 would become pattern #4, and no further work would be required on the part of `Worker` to implement any methods from `BasicMBean`. This relationship is shown in Figure 2-4.

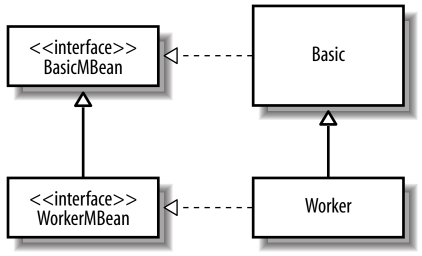


Figure 2-4. UML notation for pattern #4

The MBean interface shown in Example 2-3 would simply need to be defined as:

```
public interface WorkerMBean extends BasicMBean {
// . . .
}
```

in order to implement this pattern. No code changes to `Worker` are required. Now, however, the management interface exposed by `Worker` is the union of `BasicMBean` and `WorkerMBean`.

Suppose that the `Supplier` class from pattern #3 has an MBean interface that extends `WorkerMBean`:

```
public interface SupplierMBean extends WorkerMBean {
// . . .
}
```

This relationship is shown in UML notation in Figure 2-5.

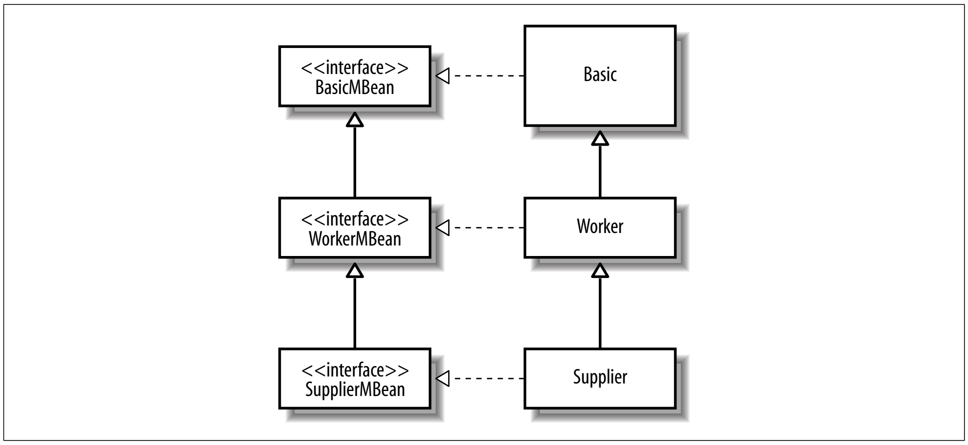


Figure 2-5. UML notation for a more complicated derivative of pattern #4

When the MBean server performs introspection, it will expose as the management interface all of the MBean interfaces until the top of the MBean inheritance graph is reached. This means that the management interface exposed by `Supplier` is the union of `BasicMBean`, `WorkerMBean`, and `SupplierMBean`.

The key to using this pattern is to remember that the MBean server uses MBean inheritance in addition to Java inheritance to determine the management interface. In other words, if you want your MBean interface to expose the management interface, you simply need to use the `extends` keyword when defining your MBean interface. Then your MBean interface will take advantage of Java inheritance as well.

Pattern #5: Compiler-enforced management interface inheritance

Suppose that, in pattern #4, `Worker` implements `WorkerMBean` but does not extend `Basic`. What then? Figure 2-6 shows this pattern in UML notation.

Clearly, the compiler will not allow `Worker` to inherit an interface without implementing it. In pattern #4, because `Worker` extended `Basic`, `BasicMBean` came along for free through Java inheritance. But if `Worker` does not extend `Basic`, it is forced to implement `BasicMBean`, because `WorkerMBean` extends `BasicMBean`.

This may sound like an oversight at first, but it really isn't. This pattern allows you to customize the implementation of management interfaces while keeping the definitions of those interfaces semantically consistent. Carrying the `Worker` example further,

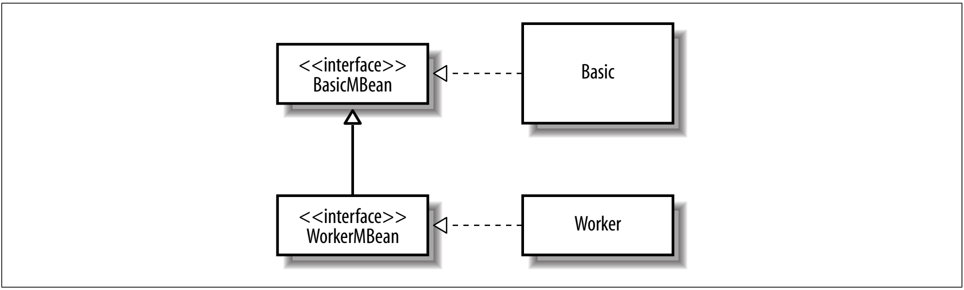


Figure 2-6. UML notation for pattern #5

suppose that you want the `WorkerMBean` management interface to be semantically identical to the union of `WorkerMBean` and `BasicMBean`. Logically, `WorkerMBean` should extend `BasicMBean`. However, suppose that you want to vary the implementation of `BasicMBean` provided by `Basic`. Using this pattern allows you to do this. And if you forget to implement anything from `BasicMBean`, the compiler will tell you!

Common Mistakes Not Caught by Introspection

The MBean server will catch and report most mistakes that you make in applying the standard MBean design patterns. However, there are mistakes you can make when applying the patterns that result in an MBean that is technically compliant but does not function as you intended. In this section, we will look at some of the most common mistakes you can make when instrumenting your application resources as standard MBeans that are not caught and reported by the MBean server. These mistakes are not caught by the Java compiler, either; your MBean simply will not work correctly. This section should aid you in troubleshooting the problem.

Mistake #1: MBean interface not given public visibility

This mistake is not caught by the compiler or reported by the MBean server. Suppose we mistakenly left off the `public` keyword from the interface definition for `BasicMBean`, using:

```
interface BasicMBean {
// . . .
}
```

instead of:

```
public interface BasicMBean {
// . . .
}
```

The `Basic` class implements `BasicMBean` as usual, and is registered by the agent:

```
public class Basic implements BasicMBean {
// . . .
}
```

To the MBean server, this MBean is compliant and is registered with no exceptions thrown. However, when the MBean server is asked to return the state of the MBean through its management interface, none of the attributes or operations can be accessed or invoked, respectively. A `javax.management.ReflectionException` is thrown by the MBean server at that time.

Mistake #2: Wrong return value type

The return value type of a getter must be never be `void`. Suppose we have a getter that is defined as:

```
public interface QueueMBean {
// . . .
  public void getQueueSize( );
  public void setQueueSize(int value);
// . . .
}
```

In this example, the intended getter method is not considered by the MBean server to be a getter at all and instead is exposed as an operation on the management interface. A proper getter must return the type of its attribute.

The return value of a setter, however, *must* be `void`. This mistake is not caught by the compiler or reported by the MBean server, and it produces some pretty strange results. Example 2-1 showed the management interface for the `Queue` class. Notice the read/write attribute `QueueSize`:

```
public interface QueueMBean {
// . . .
  public int getQueueSize( );
  public void setQueueSize(int);
// . . .
}
```

Suppose we provided a return value for *setQueueSize()*:

```
public interface QueueMBean {
// . . .
  public int getQueueSize( );
  public int setQueueSize(int value);
// . . .
}
```

When the MBean server performs its introspection, it sees the *getQueueSize()* method, indicating a readable attribute called `QueueSize`. It also notices a *setQueueSize()* method that takes the correct number of parameters for a setter but also provides a return value. Because a setter cannot return a value, the MBean server interprets the method as a management interface operation instead of as an attribute setter. Thus, in our example, a read-only attribute called `QueueSize` and an operation called *setQueueSize()* that takes an `int` parameter and returns an `int` would be exposed. (Note that the choice of `int` as the return value in this example was purely arbitrary. This mistake would happen if we had used any other type as well.)

If the management interface for your MBean does not show up as you expected, check the return values of all of the setter methods on your MBean interface. Any setter methods that return a value of any type other than `void` will be exposed as operations called *setSomething()*, where *Something* is the name of the attribute. Remember, a proper setter must return `void`!

What if the return value type of the getter is different from the parameter type of the setter? This is probably the least common mistake, as it is usually the result of mistyping the declaration of either the getter or the setter. Because the declaration must be typed twice (unless you copy the declaration from the interface and paste it into the implementing class), this mistake is not likely to occur often. However, because it has baffling results, I wanted to mention it here. Suppose you mistakenly define the management interface as:

```
public interface QueueMBean {
// . . .
  public int getQueueSize( );
  public void setQueueSize(long value);
// . . .
}
```

Do you see the problem? Notice that the parameter type to the setter is a `long`. This interface definition for the `QueueSize` attribute certainly follows the rules: the getter takes no parameters and returns a value, while the setter returns `void` and takes only one parameter.

So what do you suppose is exposed as the management interface? In the JMX 1.0 RI, the management interface exposed is a write-only attribute of type `long`. That's it—there is no getter. Because of the conflicting types, the MBean server has to choose what is exposed, and the setter wins. Exactly what is exposed in the JMX you use depends on how the MBean server's introspection is implemented. However, the point is the same: make sure the parameter type of the setter and the return type of the getter match!

Mistake #3: Wrong number of parameters

Suppose we define a getter that takes an argument on the management interface:

```
public interface QueueMBean {
// . . .
  public int getQueueSize(int value);
  public void setQueueSize(int value);
// . . .
}
```

When the MBean server performs its introspection, it will detect that *getQueueSize()* takes a parameter and will expose it as a management operation instead of as the getter for the `QueueSize` attribute. A proper getter must take *zero* arguments.

A setter must take only one argument, and that argument must be of the type of the attribute it is to set. Suppose that the management interface is defined as:

```
public interface QueueMBean {
// . . .
  public int getQueueSize( );
  public void setQueueSize(int value, char someOtherValue);
// . . .
}
```

When the MBean server performs its introspection, it exposes a read-only attribute called `QueueSize` and an operation called *setQueueSize()* that takes an `int` and a `char` parameter and returns `void`.

If the management interface for your MBean does not appear as you expected, check the number of arguments to all of the setter methods on the MBean interface. Remember, a proper setter must take only one parameter!

Implementing the MBean Interface

In this section, we will see how to implement MBean interfaces and will take a look at the application that is used throughout this book. Implementing the MBean interface is actually very straightforward, as we'll see.

The classes we will use in this chapter and their relationships to one another are shown in UML notation in Figure 2-7.

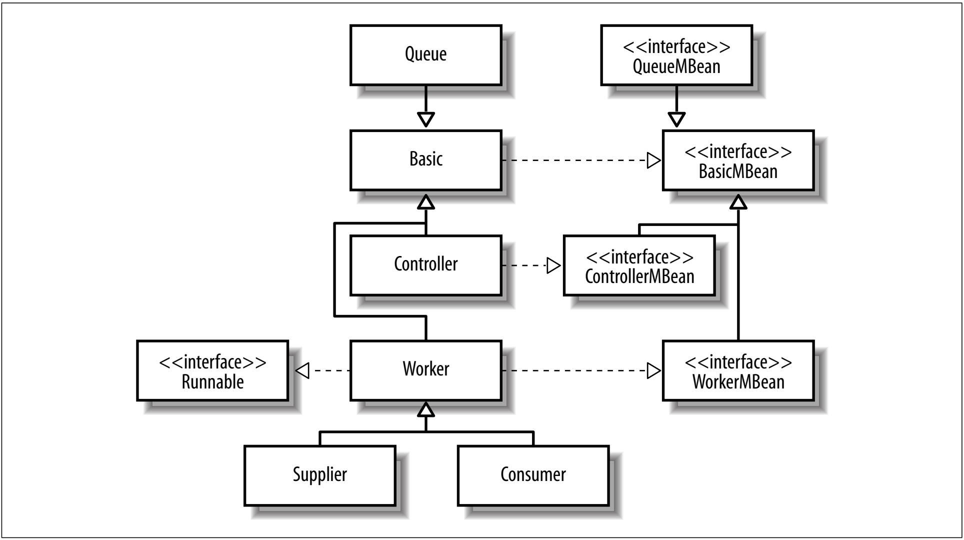


Figure 2-7. UML notation for the application classes used in this chapter

As we saw in design pattern #1, we must implement the MBean interface on the appropriately named class using the `implements` keyword. For the sake of review, let's take a quick look at how to do this. Example 2-2 showed the `BasicMBean` management interface definition. The syntax to implement this interface on the `Basic` class is:

```
public class Basic implements BasicMBean {
// . . .
}
```

Notice that this interface has only getters (i.e., all attributes are read-only). Because the implementation of the getters will be hidden behind the MBean interface, we are free to implement them however we choose, as long as the implicit contract provided by the interface is obeyed. However, the most common way to implement a getter is to declare a private instance variable on the class and simply return the value of that member variable when the getter is invoked:

```
public class Basic implements BasicMBean {
// . . .
  private boolean _traceOn;
  private int _numberOfResets;

  public boolean isTraceOn( ) {
    return _traceOn;
  }
// . . .
  public int getNumberOfResets( ) {
    return _numberOfResets;
  }
// . . .
}
```

Implementing an operation is equally straightforward:

```
public class Basic implements BasicMBean {
// . . .
  public enableTracing( ) {
    _traceOn = true;
  }
// . . .
}
```

When implementing the operation, we simply write code that makes the operation do something. Notice that the *enableTracing()* operation shown above resembles a setter. Why didn't we simply provide a setter for the `TraceOn` attribute? Notice that *enableTracing()* acts as a setter—it sets the value of the attribute—but with one important difference: it only sets the attribute to `true`. A setter can set the value to any value (within language limits, of course) that is acceptable for the data type of the attribute.

The full source listing of the `Basic` class is shown in Example 2-4.

Example 2-4. Full source listing for Basic class

```
package sample.standard;

public abstract class Basic implements BasicMBean {
  // backing stores
  private boolean _traceOn;
  private boolean _debugOn;
  private int _numberOfResets;
  // not on management interface
  public void setNumberOfResets(int value) {
    _numberOfResets = value;
  }
  // attributes on management interface
  public boolean isTraceOn( ) {
    return _traceOn;
  }
  public boolean isDebugOn( ) {
    return _debugOn;
  }
  public int getNumberOfResets( ) {
    return _numberOfResets;
  }
  // operations on management interface
  public void enableTracing( ) {
    _traceOn = true;
  }
  public void disableTracing( ) {
    _traceOn = false;
  }
  public void enableDebugging( ) {
    _debugOn = true;
  }
  public void disableDebugging( ) {
    _debugOn = false;
  }
  public abstract void reset( );
}
```

Each child class of `Basic` must provide its own implementation of *reset()*. For this reason, there is a public setter (this setter also could have been protected or friendly) for the `NumberOfResets` attribute. Notice, however, that *setNumberOfResets()* is not included on the `BasicMBean` interface and hence is not part of the management interface of the MBean.

While simply returning the value of a member variable of the same type as the attribute for which the getter is provided is the most common way of writing a getter, it is not the only way. For example, without changing the interface, we might implement `NumberOfResets` as:

```
    public class Basic implements BasicMBean {
    // . . .
```

```
  private Integer _numberOfResets = new Integer(0);
// . . .
  public void setNumberOfResets(int value) {
    _numberOfResets = new Integer(value);
  }
// . . .
  public int getNumberOfResets( ) {
    return _numberOfResets.intValue( );
  }
// . . .
}
```

Notice that in this code snippet, the backing store for the NumberOfResets attribute is a java.lang.Integer object. When *getNumberOfResets()* is called, we simply return *intValue()* on the Integer object. Again, the MBean interface serves as the contract between the MBean server, a management application, and your MBean. As long as your MBean implementation obeys the MBean interface, you are free to implement the interface however you choose.

A getter doesn't have to be that simple, however. For example, a getter can also be a calculated quantity. Consider the declaration of the WorkerMBean interface in Example 2-3. Notice the read-only attribute AverageUnitProcessingTime, which is of type float:

```
public class Worker extends Basic implements WorkerMBean {
// . . .
  private long _totalProcessingTime;
  private long _numberOfUnitsProcessed;
  public getNumberOfUnitsProcessed( ) {
    return _numberOfUnitsProcessed;
  }
  public float getAverageUnitProcessingTime( ) {
    return (_numberOfUnitsProcessed > 0)
      ? (float)_totalProcessingTime / (float)_numberOfUnitsProcessed
      : 0.0f;
  }
    // . . .
}
```

In designing the sample code, I decided to calculate the elapsed system time required to process a work unit and accumulate it in a private instance variable of type long called _totalProcessingTime. Then, when *getAverageUnitProcessingTime()* is called, the average is calculated by dividing _totalProcessingTime by the number of units processed so far (taking care not to divide by zero if no units have been processed).

Implementing a setter is equally straightforward. Consider the WorkerName attribute on WorkerMBean (see Example 2-3):

```
public class Worker extends Basic implements WorkerMBean {
// . . .
  private String _workerName;
  public String getWorkerName( ) {
    return _workerName;
  }
  public void setWorkerName(String value) {
    _workerName = value;
  }
// . . .
}
```

Consider the needs of your application before implementing your setters. Depending on how robust I want to make the implementation, the implementation of *setWorkerName()* shown above may be sufficient. However, I might want to set the value only if the new value is not a `null` reference, in which case I would make the following modification:

```
public class Worker extends Basic implements WorkerMBean {
// . . .
  private String _workerName;
  public String getWorkerName( ) {
    return _workerName;
  }
  public void setWorkerName(String value) {
    if (value != null)
      _workerName = value;
  }
// . . .
}
```

An example of a more complicated setter is *setQueueSize()*, for the `Queue` class. This setter allows a management application to dynamically alter the size of the queue, so, as you can imagine, it is not as straightforward as simply setting an attribute value. Here is the code for *setQueueSize()*:

```
public class Queue extends Basic implements QueueMBean {
// . . .
  public synchronized void setQueueSize(int value) {
    if (!_suspended) {
      if (value > _backingStore.length) {
        Object[] newStore = new Object[value];
        System.arraycopy(_backingStore, 0, newStore, 0, _backingStore.length);
      }
    }
    notifyAll( );
  }
// . . .
}
```

This code allows the queue to grow but not to shrink. Essentially, what this setter does is this: if activity in the queue is not currently suspended, and if the new queue size is greater than the current size, a new `Object` array is allocated and copied, then any threads in the wait state are signaled to become runnable. It's not too complicated, but it's certainly not as simple as just setting an instance variable's value.

Throwing Exceptions from Your MBeans

There will be times when you need to throw an exception from your MBeans—for example, when a setter needs to report that a bad value has been passed. Suppose that the setter for the `QueueSize` attribute on the `Queue` class needs to report when an attempt is made to shrink the queue (remember, the queue is allowed only to grow). If I want to throw such an exception, I have to change the declaration on the MBean interface:

```
public interface QueueMBean extends BasicMBean {
// . . .
  public setQueueSize(int value) throws Exception;
// . . .
}
```

as well as the implementing class:

```
// . . .
import sample.exception.*;
// . . .
public class Queue extends Basic implements QueueMBean {
// . . .
  public synchronized setQueueSize(int value) throws Exception {
    if (!_suspended) {
      if (value > _backingStore.length) {
        Object[] newStore = new Object[value];
        System.arraycopy(_backingStore, 0, newStore, 0, _backingStore.length);
      }
      else {
        throw new GenericException("Queue.setQueueSize( ): ERROR: " +
          "Queue size may not be set less than the current size of " +
          this.getQueueSize( ) + ". The value of " + value + " is invalid.");
      }
    }
    notifyAll( );
  }
}
```

If we attempt to set the value of the queue to be less than its current size (i.e., to shrink the queue), an exception containing a message describing the mistake will be thrown to the management application.

It is perfectly fine to use a user-defined exception. In this example, I used one called `GenericException`, located in the `sample.exception` package:

```
package sample.exception;

public class GenericException extends Exception {
  public GenericException(String message) {
    super(message);
  }
}
```

The Driver Program: Controller.main( )

As its name implies, `Controller` is the class that contains the *main()* method that drives the application and controls the activities that occur within it. This class has a number of interesting features. Recall that there are three levels to the JMX architecture: instrumentation, agent, and distributed services. So far, we have been concerned with only the instrumentation level. However, instrumentation by itself isn't very interesting. `Controller` is part of the agent level, and it performs a few duties that allow the other standard MBeans (e.g., `Queue` and `Worker`) to be plugged into the MBean server. In this section, we will discuss some of the duties of this agent program that are unrelated to standard MBeans per se but that are important for understanding JMX.

The ObjectName class

The `ObjectName` class is provided by the RI and is crucial to the MBean registration process. Every MBean must be represented by an `ObjectName` in the MBean server and no two MBeans may be represented by the same `ObjectName`. Each `ObjectName` contains a string made up of two components: the domain name and the key property list. The combination of domain name and key property list must be unique for any given MBean and has the format:

```
domain-name:key1=value1[,key2=value2,...,keyN=valueN]
```

where `domain-name` is the *domain name*, followed by a colon (no spaces), followed by at least one *key property*. Think of a domain name as JMX's namespace mechanism. A key property is just a name/value pair, where each property name must be unique. For example, the object name used by the `Queue` instance into which the `Supplier` places its work units is:

```
DefaultDomain:name=Queue
```

Notice the domain name. Every compliant JMX implementation must provide a default domain name. For the JMX 1.0 RI, that name is `DefaultDomain`, but you can't depend on this to be the case all of the time. The MBean server provides a method called *getDefaultDomain()* that returns the name of the default domain.

> As a convenience, the JMX 1.0 RI allows you to pass an empty string for the domain name if you want to use the default domain. However, the domain name you pass may never be `null`, or a `MalformedObjectNameException` will be thrown.

There is only one restriction on domain names: you cannot use `JMImplementation` as the domain name for your MBeans. This domain name is reserved for the implementation (hence the name) and contains a single metadata MBean that provides information about the implementation, such as its name, version, and vendor.

To create an ObjectName instance, use one of the three constructors provided. The simplest constructor to use takes a single String parameter that contains the full object name string, as described above:

```
// . . .
try {
  String myObjName = "UserDomain:Name=Worker,Role=Supplier";
  ObjectName = new ObjectName(myObjName);
} catch (MalformedObjectNameException e) {
// . . .
}
```

In this example, you can also leave off the domain name preceding the colon if you want to use the default domain:

```
// . . .
try {
  String myObjName = ":Name=Worker,Role=Supplier";
  ObjectName = new ObjectName(myObjName);
} catch (MalformedObjectNameException e) {
// . . .
}
```

The second constructor is provided as a convenience when you want to provide only one key property. It takes three String arguments: the domain name, the key property name, and the key property value.

```
// . . .
try {
  // String objName = "UserDomain:Name=Controller";
  ObjectName = new ObjectName("UserDomain", "Name", "Controller");
} catch (MalformedObjectNameException e) {
// . . .
}
```

The third constructor is used when you want to use the contents of a Hashtable to set the key property list. It takes two arguments: the domain name and a Hashtable reference containing the name/value pairs that make up the key property list.

```
// . . .
try {
  Hashtable table = new Hashtable( );
  table.put("Name", "Worker");
  table.put("Role", "Supplier");
  ObjectName = new ObjectName("UserDomain", table);
} catch (MalformedObjectNameException e) {
// . . .
}
```

Once the ObjectName instance for your MBean has been created successfully, you can use that ObjectName to register the MBean.

Registering the MBean with the MBean server

Without an `ObjectName` instance, an MBean cannot be registered with the MBean server. In fact, the `ObjectName` is critical to doing anything meaningful with the MBean server. In the previous section, we saw how to create an `ObjectName` instance using one of the three constructors provided by `ObjectName`. In this section, we will see how to use that `ObjectName` to register an MBean.

The first step in using the MBean server is to obtain a reference to it. Every compliant JMX implementation must provide an `MBeanServerFactory` class that contains several methods that allow you to gain access to the MBean server (these will be discussed in more detail in Chapter 6). The easiest method to use is *createMBeanServer()*, which takes no arguments and returns a reference to a newly created MBean server:

```
// . . .
MBeanServer server = MBeanServerFactory.createMBeanServer( );
// now do something with the MBean server
// . . .
```

Now that we have a reference to the MBean server, we can register our MBean. The following example shows how to create an `ObjectName`, obtain a reference to the MBean server, and register the `Controller` MBean:

```
// . . .
try {
  MBeanServer server = MBeanServerFactory.createMBeanServer( );
  ObjectName objName = new ObjectName("UserDomain:Name=Controller");
  Controller controller = new Controller( );
  server.registerMBean(controller, objName);
} catch (MalformedObjectNameException e) {
// . . .
}
// . . .
```

There are several ways to register an MBean. In the previous example, the MBean object was created explicitly using the `new` keyword, and then a reference to that object was passed to the *registerMBean()* method of the MBean server. However, those two steps could have been combined into one, allowing the MBean server to create the MBean object:

```
// . . .
try {
  MBeanServer server = MBeanServerFactory.createMBeanServer( );
  ObjectName objName = new ObjectName("UserDomain:Name=Controller");
  server.createMBean("sample.standard.Controller", objName);
} catch (MalformedObjectNameException e) {
// . . .
}
// . . .
```

The MBean server also provides an overloaded version of *createMBean()* that allows you to specify constructor parameters for your MBean. The various ways to create and register MBeans will be covered in more detail in Chapter 6.

Once the MBean is registered with the MBean server, it is available for management. The mechanisms used by a management application to plug into and manage an MBean server are part of the distributed services level of the JMX architecture and are not fully specified in the JMX 1.0 RI. However, provided with the RI is a class called `HTMLAdaptorServer`, which is used throughout this book and is described in the next section.

The HTMLAdaptorServer Class

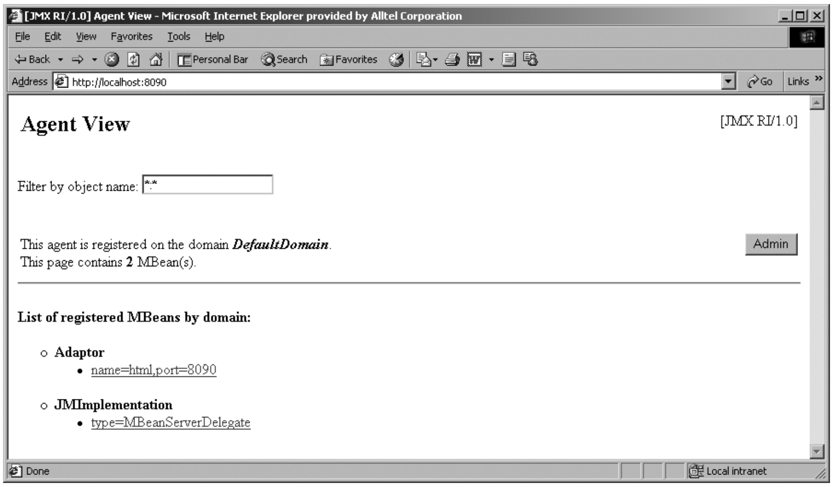
The `HTMLAdaptorServer` class is located in the `com.sun.jdmk.comm` package, which is distributed as part of the RI in *jmxtools.jar*. This handy class allows us to manage an MBean server through a web browser. `HTMLAdaptorServer` ("Adaptor" for short) is itself an MBean, and as such it must have an `ObjectName` and be registered with the MBean server. This class is essentially an HTTP server that listens on a specified port and generates HTML forms that are sent to the web browser. It is through these forms that you can manage and monitor your MBeans.

To use the `HTMLAdaptorServer` class, you must create an `ObjectName` for it and register it with the MBean server, as you would any other MBean:

```
// . . .
  MBeanServer server = MBeanServerFactory.createMBeanServer( );
  int portNumber = 8090;
  HtmlAdaptorServer html = new HtmlAdaptorServer(portNumber);
  ObjectName html_name = null;
  try {
    html_name = new ObjectName("Adaptor:name=html,port=" + portNumber);
    server.registerMBean(html, html_name);
  } catch (Exception e) {
    System.out.println("Error creating the HTML adaptor. . .");
    e.printStackTrace( );
    return;
  }
  html.start( );
// . . .
```

In this example, the Adaptor will be listening for HTTP requests on port 8090 of the machine on which it is running. A new instance of the `HTMLAdaptorServer` class is created, passing the specified port number to its constructor. Then an `ObjectName` is created for the Adaptor, and it is registered with the MBean server. Finally, the Adaptor is started. `HTMLAdaptorServer` implements the `Runnable` interface (actually, its parent class, `CommunitorServer`, does), so it runs on its own thread. Once the thread is started (by calling the *start()* method), the Adaptor is running and awaiting HTTP requests.

Now that the Adaptor is running, all you need to do is point your browser to the machine that contains the JVM in which the MBean server is running. Assuming that the browser and the MBean server are running on the same machine, simply point your browser to *http://localhost:8090*. Figure 2-8 shows a screen shot of the form that will be displayed.


[JMX RI/1.0] Agent View - Microsoft Internet Explorer provided by Alltel Corporation
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost:8090
Go Links
Admin
Done
Local intranet

Agent View

[JMX RI/1.0]

Filter by object name: *:*

This agent is registered on the domain ***DefaultDomain***.
This page contains **2** MBean(s).

List of registered MBeans by domain:

- **Adaptor**
  - name=html,port=8090
- **JMImplementation**
  - type=MBeanServerDelegate

Figure 2-8. The Adaptor in action

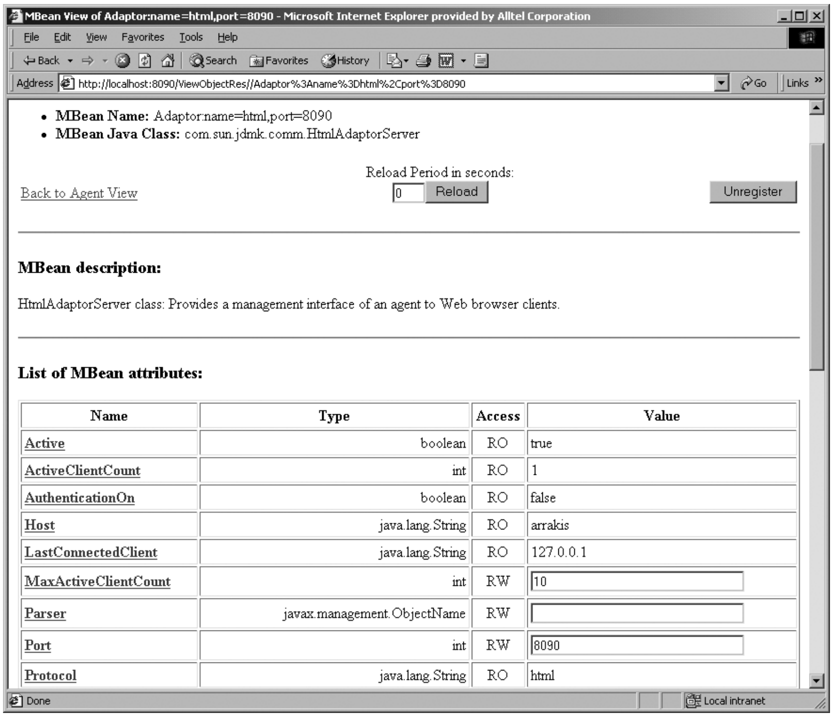
The important elements are:

"Filter by object name:"
: This text box contains a pattern for which MBeans to show under "List of registered MBeans by domain:". The pattern starts with the domain name, followed by a colon, followed by the key property list of the MBeans to show. By default, this is "*:*", which means to show all domains and all MBeans.

"List of registered MBeans by domain:"
: This is a bulleted list of domains that match the filter (see above) and the MBeans within that domain that also match the pattern.

Notice that there are two domains, `Adaptor` and `JMIplementation`. In Figure 2-8, we see that when the `ObjectName` was created for the Adaptor MBean, "Adaptor" was provided as the domain name. The key property list consists of "name=html,port=8090".

If you click on this MBean (the key property list contains a link), you can view the attributes and operations exposed on the Adaptor MBean. The attributes are shown in Figure 2-9, and the operations are shown in Figure 2-10.


MBean View of Adaptor:name=html,port=8090 - Microsoft Internet Explorer provided by Alltel Corporation
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8090/ViewObjectRes//Adaptor%3Aname%3Dhtml%2Cport%3D8090
Go Links
Done
Local intranet

- **MBean Name:** Adaptor:name=html,port=8090
- **MBean Java Class:** com.sun.jdmk.comm.HtmlAdaptorServer

Back to Agent View

Reload Period in seconds: 0 Reload

Unregister

MBean description:

HtmlAdaptorServer class: Provides a management interface of an agent to Web browser clients.

List of MBean attributes:

Name	Type	Access	Value
Active	boolean	RO	true
ActiveClientCount	int	RO	1
AuthenticationOn	boolean	RO	false
Host	java.lang.String	RO	arrakis
LastConnectedClient	java.lang.String	RO	127.0.0.1
MaxActiveClientCount	int	RW	10
Parser	javax.management.ObjectName	RW	
Port	int	RW	8090
Protocol	java.lang.String	RO	html

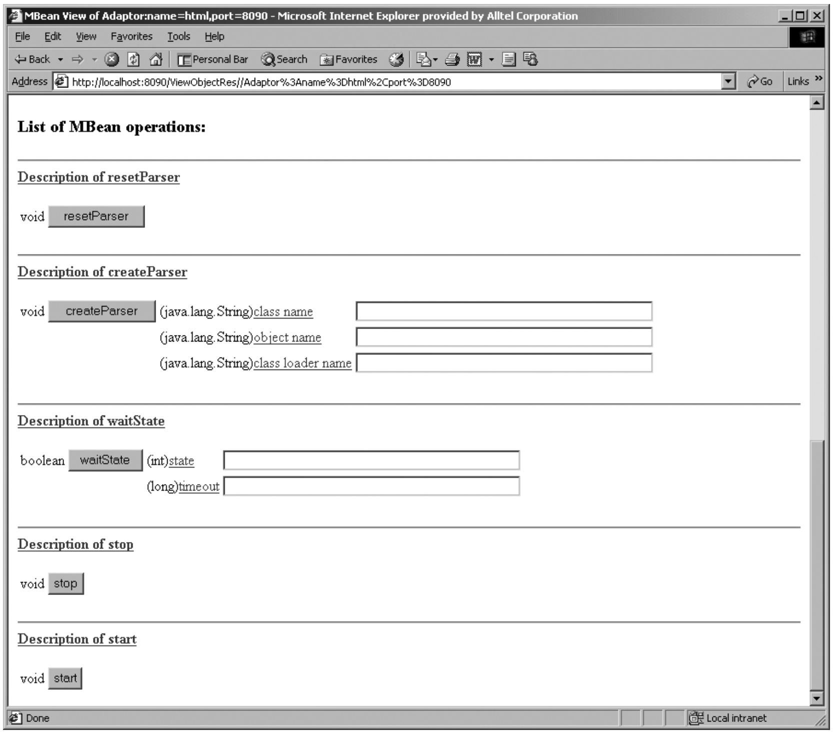
Figure 2-9. Attributes exposed for management on the HTMLAdaptorServer; scroll down to view the operations

We will use the `HTMLAdaptorServer` class throughout this book for managing all of the MBeans in the application.

Downloading and Installing the JMX Reference Implementation

Before you can build and run the application, you must first obtain the JMX RI. The easiest way to do this is to download it from Sun Microsystems at *http://java.sun.com/products/JavaManagement/*. Select either the source code or binary RI under "JMX Deliverables" and follow the instructions.

Once you've downloaded the RI, you should unzip the downloaded file directly into your *c:\* drive for Windows or your home directory for Unix.


MBean View of Adaptor:name=html,port=8090 - Microsoft Internet Explorer provided by Alltel Corporation
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost:8090/ViewObjectRes//Adaptor%3Aname%3Dhtml%2Cport%3D8090
Go Links
Done
Local intranet

List of MBean operations:

Description of resetParser

void resetParser

Description of createParser

void createParser (java.lang.String)class name
(java.lang.String)object name
(java.lang.String)class loader name

Description of waitState

boolean waitState (int)state
(long)timeout

Description of stop

void stop

Description of start

void start

Figure 2-10. Operations exposed for management on the HTMLAdaptorServer