Chapter 12
NOI and Java Beans

You've probably heard about Java Beans by now, but unless you're actually developing some, you're also probably not quite sure precisely what a Java Bean is. That's not too surprising, since tracking down an authoritative definition isn't easy. Beans are not like Applets or Servlets: A given Java class is an Applet if it extends java.applet.Applet, or it's a Servlet if it extends javax.servlet.Servlet. There is no prescribed base class for Beans.

If you download and install a recent copy of the Java Development Kit (JDK) from Sun's Web site (http://java.sun.com), you'll find that there is in fact a class called java.beans.Beans; however, a Java class that wants to be a Bean would probably never extend this class. There are several characteristics that seem to be included in most people's definitions of Java Beans. If your Java class includes these, it can be considered a Bean:

1. Must have a default constructor (one that takes no arguments).
2. Method names must follow the Beans naming conventions.
3. Classes should be serializable, but it's not required. *Serializable* means that objects of the class support having their content being serialized on a data stream. More on this in the following section.
4. Class can optionally implement events in a standardized way.
5. Classes can optionally support a java.beans.BeanInfo interface.
6. Class can optionally support a java.beans.Visibility interface.

The Java Beans specification is freely available from the JavaSoft Web site, and goes into each of these points in some detail, so I won't try to duplicate that here. What follows in this chapter is a brief examination of where the Java NOI conforms to these characteristics and where it departs from them, and why.

Background: What About Beans?

One way to look at what Java Beans are about is to think of them as Java's answer to Microsoft's OCX and Active/X component technology. People who early on jumped on the downloadable Java Applet bandwagon found that their development efforts suffered from the lack of standardization around code reuse and event propagation. Tool builders in particular found themselves blocked in their ability to develop high level development environments for Java Applets by the fact that it was relatively hard to read any random .class file and figure out what the code did.

JavaSoft answered these concerns in their Java 1.1 release by incorporating several new technologies directly in the language: a component model, an event model, standardization of method naming conventions, serialization and "introspection" capabilities, among others.

The current level of Bean technology in Java has largely been driven by the needs of the tool builders: the Borlands and Symantecs and IBMs of the world who have been building sophisticated development environments for other languages (C, C++, even SmallTalk) for years are now doing the same for Java. Their early efforts were hampered because it was so hard to write a tool that could figure out what a given Java class was all about just from its compiled byte codes (not impossible, just hard). The common goal of these tool builders was to make the language support the right set of features, to enable them to create development tools that could, for example, allow a user to drag icons representing individual Java classes onto a palette, and "wire" them together into a complete working Applet, all without making the user write a line of Java code.

There are now a number of tools which in fact support this level of capability: IBM's Bean Machine, Borland's JBuilder, and Symantec's Visual Cafe (among others) are all aimed at this kind of functionality, and all make heavy use of the Beans technologies

listed above. This focus has tended (in my opinion) to skew the discussions and developmental efforts around Java Beans toward an Applet-centric and user-interface-centric view of the Java component world. The Java Notes Object Interface, on the other hand, is (at least in its Domino 4.6 incarnation) very much oriented toward development of server-based applications and Agents. That's not to say that the currently available visual builder tools are not appropriate for use with NOI, only that the strengths of each do not necessarily mesh well (yet).

Let's delve briefly into the important new Beans-oriented technologies in the Java 1.1 release, and also consider how NOI deals (or doesn't deal) with each.

Java Beans Technologies

Method Naming Conventions

Although Java makes no explicit distinction between the methods and properties of a class, other languages do, and people often find tools that present these distinctions more approachably. If you associate properties with object attributes (blue, bold, *x* pixels wide by *y* pixels high), and methods with object behaviors (start, stop, move to a new location on the screen), you have a nice division of the object's interface.

The Java Beans spec gives you some standard patterns for naming the methods on a class so that pairs of set/get methods can be collected into single properties by a process of induction. Take, for example, these two methods from the lotus.notes.View class (see Chapter 3 for the details):

```
java.util.Vector getReaders()
void setReaders(java.util.Vector names)
```

This pair of methods is expressible as a single read/write property: Readers, and a good builder tool can present just the one property name in a list of properties for the class,

hiding the fact that there are really two methods that implement the property. The pattern here is expressible as follows:

```
<DataType> get<PropertyName>()
void set<PropertyName>(<DataType>)
```

Another example of a property pattern is:

```
boolean is<PropertyName>()
void set<PropertyName>(boolean)
```

Any pair of methods on a class that fit one of these two patterns qualifies as a Java Beans read/write property. If only the "get" call of the pair exists, then the property is read only. You can see why this is nice for builder tools: They can present an object's properties in a nice user interface and allow users to specify at design time that certain "event triggers" should cause certain properties to be set to certain values. This allows the tool to take on the work of writing the actual Java code to implement that logic.

Methods are a bit more eccentric in this world—they can have any number of arguments and return any type of result (or none at all). Whereas properties lend themselves to visual "wiring" (*get* properties take no arguments and return a single value; *set* properties return nothing and take only a single value), methods are trickier, and typically require the user to deal with Java syntax a bit more. Furthermore (again this is my own opinion), a heavy reliance on exposing the functionality of a class as properties is much more consistent with UI objects than with server objects. Users have no trouble visualizing what will happen when an event trigger causes a message to be sent to an object on the builder's palette causing that object to change some UI characteristic (size, color, position on the screen, and so on). It's a bit more difficult to figure out how to represent in a nice visual fashion how an event should cause a *method* requiring two or three arguments (a Database open, for example) to get executed. We'll see some of the implications of this below.

The Java NOI classes all conform to the Beans naming conventions, as described in Chapter 2.

Event Model

Before the release of Java 1.1, events were confined to Java's Abstract Windowing Toolkit (AWT) subsystem. The Java Beans specification says that *any* class can be an event source or an event handler. Without going into all the details of all the various ways events can be implemented between classes, you can use the new Beans classes to rather flexibly implement any set of events for a class that makes sense to you. A class might be only a source (emitter) of events, only a handler (catcher) of events, or it might do both. Classes that receive event notifications might act on those notifications by changing one or more of their attributes, by emitting additional events of their own, or by ignoring them. A class might *source* an event which constitutes a notification to the world that it is about to do something, and then listen for a response to see if any objects out there want to veto the action.

Parenthetically we should note that Lotus's own InfoBus technology, which lets Java Beans embedded in an Applet page "find" each other and exchange event notifications and data, has been adopted by JavaSoft and will soon appear as part of the language specification.

Currently, NOI does not support events. The NOI classes neither source nor listen for any Java events. This is one of the areas of what you might refer to as the *cognitive dissonance* between the very UI/Client-oriented view of Java Beans and the very "invisible"/Server orientation of NOI for Domino 4.6. There simply is no server analog (at this time, anyway) to the Applet "page" on which multiple Beans might be embedded, and within which they might want to communicate with each other.

We can envision such a setup, of course, perhaps in terms of collections of Agents or Servlets that watch for events of various kinds and use InfoBus (or something) to talk

to each other. That technology is not yet here, though. For one thing, both the Domino Agent framework and the usual HTTP Servlet architecture assume a predefined triggering of Agents and Servlets: You specify a schedule or pick one of a limited set of events to trigger an Agent, or you invoke a Servlet via a URL. Servlets and Agents are loaded into memory and run when invoked. Neither architecture currently supports the idea of an Agent or Servlet that sits in memory all the time, "listening" for one or more event notifications, upon receipt of which it does something.

Given the current state of the art, though, it made no sense for the Domino 4.6 NOI to support event handling on the server, so none was implemented. Chapter 14 comes back to this topic in the context of future directions for NOI.

Introspection and BeanInfo

Introspection is the ability of a program (typically a builder tool, but it could be any program) to glean from a Java class the exact methods, properties, and events that the class implements. You can see why a builder tool would need that information: It has to present a class's interface in a nice way to a user who may not know how to write a line of Java code. The best and most reliable way to receive that information is to somehow get it from the class in question itself.

Before Java 1.1 was released, the only way to introspect a Java class was to read the Java byte codes from the .class file and parse them yourself. Of course you'd also have to look for and parse the class's superclasses, if there were any.

Java 1.1 has a new class that handles all this for you, called java.beans.Introspector. It reads a Java class file from disk and figures out who its superclasses are (if any), and what methods, properties, and events those classes implement. It returns an instance of a descriptor class, called java.beans.BeanInfo, which you can interrogate to get the exact signatures of all public methods, properties, and events.

Sometimes, though, a Bean developer wants to hide certain aspects of the Bean. Perhaps a method was declared public because it gets called by other classes with which the Bean has a special relationship, but the method shouldn't be called by the builder tool, or by the end user at all. Or perhaps there's more than one form of some method or property, and only one of the variants is the one that should be used. In such cases the developer of a Bean is free to implement her or his own BeanInfo class (java.beans.BeanInfo is an interface, so any class can implement it). The way you associate a BeanInfo class with the class which it describes is simply by name: If your class is named lotus.notes.Session, for example, then the associated BeanInfo class must be named lotus.notes.SessionBeanInfo.

When the java.beans.Introspector class is asked to parse a Java class, it first looks for a BeanInfo for that class. If it finds one, it has only to instantiate it and invoke the descriptor methods to get all the information about that class which the developer wants known. If no BeanInfo class is found, then the Introspector actually reads the byte codes for the target class to get the information (this process is called *low level reflection*).

Because the Java NOI for Domino has no hidden methods or properties, there are no explicit BeanInfo classes for it. One could argue that certain methods that appear in the output generated by the javadoc utility should either not have been *public* or should be hidden by use of a BeanInfo class. The GetCppObject() call (present in all of the NOI classes) is one example. It is used internally to get the handle of the C++ object for which the target Java instance serves as the wrapper. It probably should either have been declared *protected* instead of public, or it should have been hidden with a BeanInfo. Fortunately the call is pretty harmless.

Visibility

There's another Beans interface called java.beans.Visibility. This interface is optional for Java Beans, and was added late in the development cycle of Java Beans 1.0. It was, in

fact, added at the explicit request of Lotus and Iris to handle the fact that the NOI classes present no user interface whatsoever (JavaSoft readily agreed that it was useful).

The Visibility interface is meant to handle two different situations:

- A smart builder tool needs to know whether or not any given Bean requires a user interface.
- A Java Bean needs to know at any given time whether or not it is okay to present a user interface.

In the first case, the tool can inquire of any Bean that implements the java.beans.Visibility interface whether it requires a user-interface capability via the needsGui() call. In the second case, a tool (or an Applet, more likely) can tell a Bean either that it's okay or not okay to put up a user interface via the dontUseGui() and okToUseGui() calls.

All the NOI classes implement the Visibility interface, and all return *false* for the needsGui() call. Unfortunately, I suspect that most builders (and Applets) pretty much ignore this interface. But I could be wrong about that; there are tools that deal just fine with *invisible* Beans, such as those that play sound clips only. Still, there's an important difference between "invisible" and "has no UI". Watch out for subtle problems in this area when using builder tools together with the NOI classes.

Serialization

Serialization is the process by which an object instance is written to an output stream for persistent storage. The reverse of serialization is called deserialization, which is the process by which an object previously serialized is brought back into memory and re-instantiated with the same state it had before.

In order for a class to be serializable, it must implement the interface java.io.Serializable. This interface has no methods at all—it simply "marks" a class as one which can be serialized (and, implicitly, deserialized).

Why is this particularly relevant to Java Beans? The answer is that Beans, as components in a larger Applet or application, might very well be given a state at design time, and be expected to know about that state later, at run time. Suppose, for example, that you had a Java Bean which simply displayed a colored square on the screen, let's call it BobsShape. BobsShape has (for the purposes of this example) the following properties:

- Height (in pixels)
- Width (in pixels)
- Location (x/y coordinate of the upper-left corner)
- Color

You might go through the following steps when embedding that Bean in an Applet you were constructing:

1. Start the builder tool (JBuilder, Bean Machine, whatever), and create your basic layout for the Applet.
2. Explore a palette of available Beans, including BobsShape. Each available Bean presents a design time icon to the builder, and that's the icon you see for it.
3. You select the icon for BobsShape, and drag it onto the palette at a specific position. You now see a square of default size and color.
4. Behind the scenes, the builder tool creates an in memory instance of the BobsShape class, and sets the Location property of that instance to the point to which you dragged it.
5. You then stretch the lower-right corner to enlarge the square. The tool sets the Width and Height properties appropriately.
6. You use the tool's UI to select a different color for the square, the tool sets the Color property of the BobsShape instance.

Now comes the fun part: You want to actually generate the Applet and save it in a jar file so that it can be downloaded to someone's browser, whereupon they'll see a square of correct size and color on the Applet's page in the correct position. There are two different ways the builder tool could handle this:

1. It can write out a bunch of Java source code for the Applet that has hardwired in it the parameters for the Bean: color, size, position. At Applet initialization time, the Java code would create a new instance of BobsShape on the fly, then set its parameters appropriately. The shape would then appear on the page where it was supposed to.
2. A more elegant way to handle this would be to have at Applet generation time each embedded Bean serialize itself to an output stream. The stream would contain, for each embedded Bean, a binary representation of each of the Bean's persistent properties (color, size, etc.). The stream would be written to a .jar file, which would also contain the actual Applet code. When the .jar file got downloaded to someone's browser, the Applet would start up, deserialize all its components, and be off and running.

In the second mechanism, each Bean saves its own state onto some output stream. At run time, the browser or the Applet is responsible for noticing that there is some serialized information in the .jar file. It can then use Java's deserialization mechanism to 9a) construct each object instance that is required, and (b) tell each instance to deserialize itself (load its individual property values) from an input stream.

The default behavior for serialization of an object is that all member variables in the object instance are written to the output stream, unless they are declared *transient*. Transient variables are skipped. Scalar values are simply written out. Object references are recursively serialized. If an object needs control over how one or more of its members is serialized, it can implement a writeObject() and a readObject() call to customize the serialization and deserialization behavior. Still more control is offered by the java.io.Externalization interface.

Serialization is a terrific piece of functionality for a few reasons. First, it is useful in many situations, not just in the Applet builder scenario. Objects can be serialized for persistent storage to disk (or to an object database), or for transmission across a network (when you make a remote call with an object reference as an argument). Serialization has to handle not just the storage of a single object, but must operate recursively as well,

serializing all objects to which a given object refers. While doing that, serialization has to also maintain the integrity of all object references, so that if (for example) two objects both refer to a third, the third object is not serialized more than once. If it were, then when the stream was read back in and deserialized, you'd end up with two instances of a class, whereas you only had one to begin with.

To invoke serialization you simply write an object to a special stream, of class java.io.ObjectOutputStream. The writeObject() method on that class inspects the target object to see if the Serialization or Externalization interfaces are implemented. If neither interface is present, an exception is thrown.

When you deserialize objects, you simply read them off an instance of the class java.io.ObjectInputStream. The readObject() call figures out what the next object in the stream is, invokes its default constructor, and then tells the object (which we already know is serializable) to read its members from the stream itself. The object emerges fully constructed (possibly invoking readObject() recursively, if it has object member variables).

You can see why this approach is attractive: there is much less code for tool developers to write, and it provides a very nice general purpose mechanism for persistent storage of all kinds, that any class can use. There's even a way you can do implicit version control (of a limited sort), so that if you save an object to a stream, modify the object's implementation by adding an additional member variable, and then read the older version of the object back off an input stream, it will work (see the documentation on java.io.ObjectStreamClass.getSerialVersionUID()). Of course, the new member variable's data will be missing in the input stream, and you have to account for that.

Domino's Java NOI, however, does not make use of serialization at this time. There are a couple of reasons for this, both historical, and functional:

- Serialization requires that each class have a default constructor. As we've mentioned before, none of the NOI classes has a default constructor that is really usable, because of the requirements of a strict containment hierarchy. At the time that this particular design decision was locked down (i.e., coded) there were no real Beans-aware builders to work with (the Beans spec had just been released). Thus, there was really no way to test out (in time) whether having a default constructor for each class could be made to work, given the design constraints imposed by the back-end Notes code (especially the one that says you can't have free-floating objects).
- It was never clear that any builders could really make effective use of the NOI classes anyway, partly because the ones that we knew about in those early days were so Applet centric, and partly because they were all very property centric, whereas NOI makes heavy use of methods to do things like instantiate child classes.
- The Java 1.1 software that incorporated robust object serialization wasn't available early enough for us to be able to come to grips with it in time for the 4.6 release.

As a result, none of the NOI classes implement either the Serializable or the Externalizable interfaces. Furthermore, all member variables in all the NOI classes are declared *transient*. This means that, at this time anyway, you can't use a builder tool to set an NOI object's initial state at design time, save the object in a .jar file, and have it "remember" that state later at run time.

Furthermore, any builder tool that relies on being able to instantiate an NOI object at design time using its default constructor had better be able to trap a System.exit() call, as that's what the default base class constructor for NOI does (remember, no one is supposed to call it). The developers of IBM's BeanMachine discovered this quirk in time to deal with it, but I don't know if any of the other vendors have done so.

NOI and Builder Tools

I've experimented using two different Java development environment products with the Domino Java NOI: Borland's JBuilder and Symantec's Visual Cafe. I picked these two to try out because they are popular tools from reputable vendors (this isn't a product review or endorsement; I just wanted to share a bit of experience with you about how NOI works in these kinds of environments). There are other tools that are (I'm sure) perfectly good too, I didn't have time to try out every one.

One tool that I know will *not* work with Domino NOI is Microsoft's Visual J++. The problem with VJ++ is that Microsoft never adopted the standard Java Native Interface (JNI) architecture. JNI is the specification which allows Java code to call into C or C++ modules, an important requirement for using the Notes classes.

The choices for Java development tools range from the simple (any text editor for typing in Java code, command line Java Development Kit compiler and interpreter from JavaSoft) to the fully featured (syntax driven editor, source code debugger, performance profiler). For Java Beans, which tend to the, shall we say, "rich" end of the functional spectrum, the full-featured development environments are a big plus, especially when it comes to debugging. If the application or Agent you're writing is small and simple, a few System.out.println() calls and a decent Java Console (as in the Notes client) are about all you need. For an event-driven, embeddable component with heavy user-interface requirements involving hundreds of lines of code, a real debugger is a big win.

Some of the new Java development tools also offer features like source code versioning and archiving. Anyhow, I picked these two products to try out, and was pleased with the results, once I got them set up and operational.

Borland's JBuilder is strong in its handling of Java Beans: Borland has really gone to town with its exploitation of the Beans introspection and event-handling features (no surprise there—they contributed to the original Beans specifications). I was hoping they would figure out (as the Bean Machine developers did) that the Domino NOI classes could really be treated as Beans, but for the lack of default constructors). Unfortunately,

JBuilder refused to recognize all but a few of the NOI classes as Beans when I tried to load them onto the components palette (see Figure 12.1).

Figure 12.1 *JBuilder's reaction to NOI as Beans.*

Symantec's Visual Cafe likewise refused to treat any of the NOI classes as Beans. Both, however, did a fine job of debugging the samples. Figure 12.2 shows Visual Cafe in debug mode on the sample debuggable Agent Ex90Conflict, from Chapter 9, and Figure 12.3 shows approximately the same code in the JBuilder debugger.

Figure 12.2 *Debugging the Ex90Conflict Agent with Visual Cafe.*

Figure 12.3 *Debugging the Ex90Conflict Agent with JBuilder.*

Summary: Are NOI Objects Java Beans?

The answer to whether or not the NOI objects are "real" Beans is probably debatable. If you agree that serialization is an option, not a requirement, then the only reason you wouldn't be able to consider the NOI classes to be true Beans is that they don't support a real default constructor (which could in all likelihood be fixed in a future Domino release). If you feel strongly that serialization is a required trait, then you'll argue that they aren't real Beans at all. If you take the pragmatic approach and say that they're not Beans because two builder tools won't treat them that way, then I really can't disagree too much.

In the end I'm not sure it matters very much one way or the other, at least not right now, given the current UI centric focus of Beans development. When more vendors and tools start focusing on using Java Beans as a component technology aimed at *server* Application development, then I think Domino NOI will have a strong role to play. Besides that, whether the NOI classes are "real" Beans or not doesn't seem to affect our ability to use different development tools to create and debug interesting Java programs that make use of Domino objects, as you've seen with both JBuilder and Visual Cafe.

In the next chapter, we'll take a look at how to use yet another Java API together with the Java NOI to access relational databases.