Chapter 2

NOI Part 1: Session, DbDirectory, Database, ACL, ACLEntry

This is where we start to dive deep into all the Notes Object Interface (NOI) classes. This chapter covers the top layer of the containment hierarchy, and subsequent chapters go into each of the 23 classes of interest to the Java programmer. Refer to Appendix A for a diagram of the NOI containment hierarchy and a list of all the classes.

Overview of NOI

The Notes object hierarchy does not make much use of class inheritance, but it does enforce a strict containment model, especially in the Java binding. There are no cases where using the *new* operator will result in a valid object instance, even for the top-level object, lotus.notes.Session. The main reason for this is that each Java object instance tightly wrappers a corresponding C++ object in the Notes LSX module (the library that actually implements all the object behaviors). The Java interface is really just a thin layer on top of a bunch of "native methods," which are implemented in C and C++. This allows Notes to use the LotusScript eXtension (LSX) architecture to present multiple language bindings to programmers, all based on exactly the same set of C++ code. When a new method or property (or even class) is added to the product, it can be exposed in all language bindings with only a very small incremental effort. Because each Java object instance is closely tied to an internal object, the objects' contexts must be strictly maintained. It makes no sense (at the Notes API layer, which was used to implement all this stuff), for example, to instantiate a free-floating Document object. Documents *must* have a database context in which to operate. The same is true for all the other NOI classes—each must exist only in the context of a container.

Another implication of strict containment is that if a container is closed (or destroyed), then all of that container's child objects are also destroyed. By "destroyed" I mean only that the in memory programmatic object is destroyed, and its resources are released. The actual object represented by the in memory object (the *real* database, document, or whatever) is not affected. Any modifications cached in the in memory object that have not been committed to disk are lost when the object is destroyed. Several of the objects have explicit save() calls on them to perform the commit operation.

If you're already familiar with the LotusScript binding of NOI (especially the back-end classes), you'll find the Java interface very familiar, except for the differences imposed by the differing syntaxes of the two languages. Of course, two different programming languages do impose some constraints on the mapping of identical functionality from one to the other, and not just syntactically (check out the September/October 1997 issue of the bimonthly publication on Notes technologies called *The View* (published by Wellesley Information Services), Vol. 3, number 5. I had a detailed article on this called "An Introduction to the New Notes Object Interface (NOI) for Java"). The following table summarizes the differences.

Table 2.1 Java/LotusScript Differences

	Java	**LotusScript**
Derived from	C++	BASIC
Function calls	Methods only	Methods and properties
Typing	Strong	Not strong
Threading	Multithreaded	Single threaded
UI Programmability	Fully featured	Minimal
Network programmability	High-level socket, URL, and TCP classes	Notes RPC only
Naming	package lotus.notes	NotesXXX

In general, in the Notes/Domino environment, you can do anything with Java applications and Agents that you can do with LotusScript Agents (there's no way to write LotusScript applications, as LotusScript is an embedded language), and Java even provides functionality lacking (so far) in LotusScript, such as multithreaded programming and high-level network object libraries. Some beloved features of LotusScript are, however, lacking in the Java NOI. Two examples are as follows:

- **Variants**. The Variant data type in LotusScript is a wonderful, and frequently used, feature of NOI. Variants can contain any data type, including object instances and arrays. Thus, they are often used in NOI when a method or property returns a data value, which might be of any type, or an array of object instances. Variants have no place in a strongly typed language such as Java, however. Instead we used method overloading (to handle input arguments of many kinds) and the object type (for single-instance return values and object instances) in the Java NOI. In cases where we needed to return an array of values or objects, we used the java.util.Vector class, all of whose elements are of type Object (or some derivative).
- **Expanded class syntax**. The LotusScript Document class (named NotesDocument) was created as an "expanded" class. This means that when you write your LotusScript program you can specify any arbitrary property name on either the left- or right-hand side of an assignment operator. The property name you use, if not an actual registered property of the Document class, is interpreted to mean "an item of the given name belonging to the referenced Document instance." Thus, if you code something like *doc.Subject* = *"hello",* it means that you want to assign the string "hello" to the item named Subject in the document, since Subject is not a registered property for that class. Likewise, if you were to use *doc.Subject* on the right-hand side of the assignment, it would mean that you wanted to get the value of the item named Subject. Java doesn't allow either of these uses. Instead we just added more accessor and value setting methods to the Document class.

If you're not already a LotusScript programmer and you care only about Java, none of this is relevant to you, really. All you need to know is that the Java binding of NOI exposes all the functionality of the LotusScript binding (one way or another), and that you're not shortchanging yourself by using Java. In fact, as we'll see, Java offers some functionality you can't get with LotusScript.

NOI Containment Hierarchy

When you write a class library, you have to be concerned with inheritance hierarchies. They serve mainly to make the developer's job easier, because they allow you to reuse methods conveniently. But when you go to write a real-life application using someone's class library as a tool kit, you could (I claim) mostly care less about inheritance. What really matters is how you navigate from object to object in a *containment* hierarchy, especially when the classes are strictly contained, as they are with NOI.

In developing the Java binding for NOI for Domino Release 4.6, we had to make a trade-off between, on the one hand, "Java-ness" and on the other, "Notes-ness." Chapter 12 goes into more detail on how the Java NOI relates to Java Beans, and we'll explain in all its goryness why this particular trade-off had to be made, how it was done, and how it might be made better in the future. In any event, the containment hierarchy for NOI is both strict, and worth understanding, if you ever intend to use it. The diagram in Appendix B (and on the CD) serves as a road map to the detailed class by class descriptions that ensue.

Introduction to the Class Descriptions

What follows in this and the next few chapters is a blow by blow, class by class description of NOI. Each class is shown with its *properties* (attributes of objects) and *methods* (behaviors), and most class descriptions include an example or two of how to use them. All descriptions are of the Java binding of NOI; no LotusScript examples are given, except to illustrate an important difference between the LotusScript and Java bindings. All examples are reproduced on the enclosed CD, together with any sample Notes databases that are necessary to run the samples. All the samples were created using the shipping build of Domino 4.6. You can find a complete description formatted for HTML by the Javadoc utility on the companion CD, in the docs directory.

For those of you unfamiliar with LotusScript, a word about properties and methods is in order. LotusScript makes a clear distinction between the two: properties are attributes of objects, while methods are behaviors. Properties can be read/write or read only, and typically (at least in the Notes hierarchy) take no arguments. Methods are what you'd expect from Java or C++: subroutine calls that might or might not take arguments or return a value of some kind.

Java, on the other hand, doesn't really have the notion of properties. It does allow for public member variables on a class, but it is rare that you'd use these for real work. For one thing, if setting an object's attribute has side effects (it usually does in a system of interesting size), then you need some code to run in order to deal with that. For another thing, it's fairly rare that you'd want to let someone set an object's attribute without at least range checking the value. It was recognized soon after Java 1.0 was released that some sort of property get/set scheme was highly desirable, and some features were added to the Java Beans specification to address this (primarily for the benefit of Bean builder tools, but we all gain by it). The Beans spec essentially just lays out a method naming convention, from which a set of properties can be induced.

For example, in the LotusScript binding of NOI there's a property on the Item class called Text. It's a read/write property, so I can both get the value of the property, and set it, like this:

```
Dim x As String
Dim i As NotesItem
x = i.Text
i.Text = "A new value"
```

The Java naming convention (which is followed by the Java binding of NOI) says that a property retrieval call starts with *get*, and a property setting call starts with *set*. If the property retrieval call returns a boolean value, you can optionally use *is* instead of *get*. So, the Item's Text "property" would be coded in Java as the following two calls:

```
String getText();
void setText(String s);
```

You use these methods just the same as any other in Java. The point of it all is that the new visual builder tools for Java Beans can "introspect" the methods of a Java class and figure out that when there's a get/set pattern conforming to the spec, as above, then it can represent that pair as a single property (String property Text, read/write in this case).

So, with that in mind, let's dive into the first NOI class. I've separated the set of methods logically into methods and properties, and now you know what that really means.

The lotus.notes.NotesThread Class

One "supporting" class needs to be talked about briefly before we start in on the actual Notes classes. NotesThread is not a real Domino object class, like the Database or Document class, but nonetheless you need to know about it to use any of the other classes. NotesThread extends (inherits from) java.lang.Thread, and must be used whenever you want to manipulate any of the Notes objects.

The reason NotesThread is required is simple: It does the necessary per-thread initialization and teardown of the Notes back end code (which, as you'll remember, is implemented in C and C++, not Java). Other than that, NotesThread is exactly like Thread, and you use it in exactly the same ways. Chapters 7 and 8 will dwell at length on how to write multi-threaded Java applications and Agents using NotesThread. For now, we'll just leave it that you need to run all of the Notes objects on a thread that's been initialized for Notes. There are three ways to do that (subsequent references to Notes classes will generally omit the "lotus.notes" package prefix):

1. Write a class that extends NotesThread, invoke the start() method. Your class's runNotes() method will be called from the new thread.
2. Write a class that implements the java.lang.Runnable interface. Create an instance of NotesThread using *new*, passing your class instance to the NotesThread constructor. Call start() on the NotesThread instance, and your class's run() method will be called.
3. If you can't do either of the previous two techniques, maybe because you're working on a UI where you have some event handlers that are invoked on an AWT (Abstract Windowing Toolkit, the Java UI class library) thread over which you have no control, then use the static calls on NotesThread instead. When you're in a situation where you need to initialize Notes for the current thread, and where the current thread is not an instance of NotesThread, then you can call NotesThread.sinitThread(), a static method (meaning, you don't need an instance of NotesThread to invoke it). WARNING: you must be absolutely sure, if you employ this technique, that you also call NotesThread.stermThread() exactly one time for each sinitThread() call on the thread. Making unbalanced calls to these two methods will most likely cause your program to throw an exception (if you're lucky), to crash (if you're not), or to hang on exit. Use the try/catch/finally mechanism to be sure to initialize and terminate the correct number of times per thread.

Exceptions

Many of the packages I've seen for Java (including the libraries distributed with Java itself) use a unique exception class for each kind of runtime error that could occur. Each exception has a different name, and there are inheritance hierarchies of them. This scheme did not map well onto Domino/Notes, where the C API and the LotusScript classes both use a system of error codes and associated text.

Instead, we created a single exception class (NotesException) to handle all of the package's error conditions. NotesException extends java.lang.Exception and provides one additional method: getErrorCode(). Any Notes call that throws an exception will throw an instance of NotesException containing the message and relevant error code.

All the error codes that are generated by the Notes classes are defined as *public static final int*s (the Java equivalent to C++ #defines) in the NotesException class. You can use the base class methods (on the Throwable class, from which Exception inherits) getMessage(), getLocalizedMessage(), toString(), and printStackTrace() to extract the message text and/or send a stack trace for debugging purposes to the standard output stream.
Okay, let's get into the real stuff.

The lotus.notes.Session Class

The Session class is the root of the NOI containment hierarchy. You can't do much of anything unless you have a Session instance handed to you, or unless you create one. If you're writing an Agent, then Domino creates a Session instance for you (see Chapter 5); otherwise, use the static method newInstance to create one. Why a static method, instead of just using the *new* operator? The main reason is that a static method can return *null* if the system hasn't been initialized for some reason, or if your process is out of memory. It's also easier for a static method to raise an exception if something is wrong. *New* must pretty much always return an object reference, which doesn't give you much flexibility in your constructor to do validity checking, and so on.

The public methods of the Session class can be divided into the following categories:

1. Initializers
2. Properties
3. Child object creation
4. All others

Session Initializers

There's only one initializer that you'd ever use: the static newInstance() method we mentioned above. There are actually two public versions of this call, one with no arguments and one with an *int* argument. The second version—the one with an *int* argument—is meant for internal use only. The argument is a "magic cookie" that the Agent subsystem uses to pass agent context to a new Session instance. For your programs, just use Session.newInstance(). It returns a Session reference, or throws a NotesException instance.

Session Properties

java.util.Vector getAddressBooks()

Read only. This property returns a Vector containing a lotus.Notes.Database instance for each of the Public Address Books known to the system. If you're running the program on a workstation, this will typically be just your local names.nsf. If you're running it on a Domino server, it is often a series of databases, as most servers make use of the address book chaining feature.

One important difference between this method and the getDatabase() method, discussed below, is that getAddressBooks() does not open the databases that are returned. Any Database property or method that you use on an address book instance that hasn't been opened yet will either return *null* or throw an exception. To open an address book instance explicitly, use the open() method.

The example in Listing 2.1 gets the current list of address books and prints out the file name for each.

Listing 2.1 Address Books Example (Ex21AddrBooks.java)

```
import java.lang.*;
import java.util.*;
import lotus.notes.*;
public class Ex21AddrBooks extends NotesThread {
    public static void main(String argv[])
         {
        try {
             Ex21AddrBooks e21 = new Ex21AddrBooks();
```

```
                e21.start();
                e21.join();
                }
            catch (Exception e) {e.printStackTrace();}
            }
        public void runNotes()
            {
            try {
                Session s = Session.newInstance();
                java.util.Vector v = s.getAddressBooks();
                if (v != null)
                    {
                    Enumeration e = v.elements();
                    while (e.hasMoreElements())
                        {
                        Database db = (Database)e.nextElement();
                        if (db != null)
                            {
                            db.open();
                            System.out.println(db.getFileName()
    + " / " +
                                                 db.getTitle());
                            }
                         }
                    }
                 }   // end try
           catch (NotesException e) {e.printStackTrace();}
            }
    }    // end class
```

Because this is the first full example in the book, I'll point out a couple of things that have nothing to do with the getAddressBooks() call. First, you have to include the import statement for the lotus.notes package, and the Notes/Domino executable directory (the one where all the executable files are installed) must be on the path, so that the proper libraries can be loaded. Furthermore, your CLASSPATH must include the notes.jar file.

Second, note that the first method in the class is a static one named *main*. This is required in order to run the program from the command line. You'd invoke this program with the command **java Ex21AddrBooks**, and the Java interpreter will start your program at the main() function. Because main() is static, no instance of the class has yet been created when the program starts. That's why the first thing main() does is create an instance of Ex21AddrBooks. Once that instance exists, we just call start() on it. That causes our runNotes() method to be invoked on a new thread. Main() then calls join() on the new thread, to wait for it to complete before exiting. While not strictly necessary in this simple example, waiting for all child threads to exit before terminating the mainline program is good practice, and it *is* required when you're writing an Agent, as we'll see in Chapter 8. Did we have to use another thread to run this? No, certainly not. We could easily have just put all the calls in main(). But then you wouldn't be as hip as you are now to the total coolness of Java and threads.

The runNotes() method is where we put the real logic of the program. It creates a new Session instance and gets a Vector containing a list of databases. Each Database instance in the Vector is an unopened address book. We iterate over the elements in the Vector using the Enumeration interface in a simple while loop, printing out each database's file name and title. We have to open each database explicitly before we can access the title (but not the file name).

When I run the program from the command line (in my case from a DOS window on my NT system), the first thing I see is a password prompt. That's because my Java program is accessing the Notes back-end, just like an API program would do, and my user id has a password on it. When I type in my password, the program continues. Then I see the names of the address books known to my system (both local address books and the ones on my default server).

AgentContext getAgentContext()

If your program is an Agent, then this call returns the context object for the current Agent. Otherwise it returns *null.* From the AgentContext class you find out all sorts of things about how the Agent is being run (current database, current user name, and so on). See Chapter 5 for details.

String getUserName()

lotus.notes.Name getUserNameObject()

String getCommonUserName()

These three calls return different versions of the user's name, as found in the current id file. The first one, getUserName() returns the fully qualified "distinguished name," for example, "CN=Bob Balaban/O=Looseleaf." The getCommonUserName() method returns only the "common" part of the hierarchical name (e.g., "Bob Balaban"), and getUserNameObject() returns the distinguished name instantiated in a lotus.notes.Name object instance (see Chapter 5 for details on this class).

lotus.notes.International getInternational()

Returns an instance of the International class, which contains a bunch of read-only properties exposing many of the international settings on your system. These include: AM/PM Strings, decimal point character, the localized word for "today," and so on. See Chapter 5 for details. There is only one instance of the International class per machine.

String getNotesVersion()

Obtains a string representing the id of the version of Notes that you have installed. The string is localized for the country and language version of the product, and usually contains the date of the release as well.

String getPlatform()

Returns the name of the operating system on which the current version of Notes is running.

lotus.notes.Database getURLDatabase()

If you have your current location record set up to refer to a Domino Web server database, whether local or remote, this call will return an instance of that Database. You can then use that Database instance to retrieve pages off the Web and convert them to Notes documents (see the write up on the Database class, later in this chapter).

boolean isOnServer()

Returns *true* if the current Agent program is running in a Domino server process. This property will be *true* for any Agent run in the background by the Agent Manager, or for any Agent invoked by the HTTP server. Any other program will return *false* for this property, even if the program is being run on a server machine, if it's being run from the workstation console or from the command line.

Session Child Object Creation

These methods are similar to the properties that return other objects belonging to NOI, but these calls are not properties, because in some cases they require input arguments, and in other cases they return objects which are not (semantically speaking) attributes of the session.

lotus.notes.DateTime createDateTime(String time)

Notes has its own internal formats for dates and times. This call creates a DateTime instance using an optional date/time string. If you want to create an "empty" DateTime instance and set the value of the object later using one of the DateTime properties, just use "" or *null* as the argument value. See Chapter 4 for details on the DateTime class.

lotus.notes.DateRange createDateRange()

lotus.notes.DateRange createDateRange(lotus.notes.DateTime start, DateTime end)

A DateRange is just a pair of DateTime instances, although the DateRange object does not embed its start and end times, it merely points to them. Thus, if you use the second form of the createDateRange() call, providing starting and ending DateTime instances, you can later change the value of one or the other DateTime instance, and the DateRange will point to the new value. Be careful if you do this, you never want the starting date to be later than the ending date, or you'll have a meaningless range.

lotus.notes.Log createLog(String name)

Returns an instance of the Log class, which can be used to log action and error information to a Notes database, to a mail message, or to disk. See Chapter 6.

lotus.notes.Name createName(String name)

Creates an instance of the Name class, initialized with the provided string, usually a full distinguished name, but it doesn't have to be. If the name you provide is not a full hierarchical name, though, this class has no way of converting a common name to a distinguished name. Full discussion of the Name class is in Chapter 5.

lotus.notes.Newsletter createNewsletter(lotus.notes.DocumentCollection list)

Newsletters are usually used to format a list of Document instances into a message, often containing doclinks to the source documents and some kind of tag or title line. The input argument is a DocumentCollection instance, which contains the Document list. Chapter 3 talks about DocumentCollections, and Chapter 6 discusses Newsletters.

lotus.notes.Registration createRegistration()

The Registration class (new to Domino 4.6) allows you to create and manage user, certifier, and server ids. See Chapter 5.

lotus.notes.RichTextStyle createRichTextStyle()

The RichTextStyle class is also new to Domino 4.6, and its purpose is to allow you to add text to a rich text item using different styles. See Chapter 4 for examples.

Session Other Methods

lotus.notes.Database getDatabase(String server, String dbname)

Returns a Database instance, given a server name and database file name. If you want to access a database on your local machine, use "" as the server name.

lotus.notes.DbDirectory getDbDirectory(String name)

DbDirectories are used primarily to iterate over the databases on a particular machine. You create an instance of DbDirectory by using this call and providing the name of the server you want to use ("" for the local machine). See below for details.

String getEnvironmentString(String name)

String getEnvironmentString(String name, boolean issystem)

Object getEnvironmentValue(String name)

Object getEnvironmentValue(String name, boolean issystem)

void setEnvironmentVar(String name, Object value)

void setEnvironmentVar(String name, Object value, boolean issystem)

"Environment variables" are named values, string, DateTime, or numeric. They are stored in your system's notes.ini file. Some environment variables are used internally by Notes; these are called *system variables*. Other environment variables you can make up yourself; these will automatically have a "$" prepended to the name you supply when you set or get their values.

To retrieve the value of an environment variable as a string, use one of the getEnvironmentString calls. If you know that the variable whose value you want is a system variable (i.e., the variable's name in notes.ini is not preceded by a "$"), then you must use the variant where you specify *true* for the second argument. If the variable is not a system variable, you can use either call (if you use the second one, specify *false* for the second argument). Any environment variable can be retrieved as a string.

If you know that the environment variable you want has either a date or a numeric value, use one of the getEnvironmentValue() calls. If the value is numeric, then an Object of subclass Number will be returned. If the value is a date, or date and time, then an instance of lotus.notes.DateTime is returned. You can use the built-in Java operator *instanceof* to determine the kind of object you've retrieved. If the environment variable does not exist, getEnvironmentValue() returns an Integer object whose value is 0. There is no way to tell the difference between a missing variable and a variable whose real value is 0, unfortunately. Again, use the variant with the boolean argument to access system variables.

The setEnvironmentVar call handles all values, String, DateTime, and numeric. You invoke it with the name of the variable you want to set, the value represented as an Object, and (optionally) a boolean indicating whether the name is a system variable or not. Using Object as the value input argument type allows you to pass any of the valid formats, as all object classes extend Object somewhere in their inheritance hierarchy. If you pass in an Object that is not one of the valid formats, an exception is thrown.

The example in Listing 2.2. sets and then gets an environment variable.

Listing 2.2 Setting an Environment Variable Example (Ex22SetEnv.java)

```
import java.lang.*;
import java.util.*;
import lotus.notes.*;
public class Ex22SetEnv {
    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            DateTime dt = s.createDateTime("today");
            s.setEnvironmentVar("Bob'sVar", dt);
            Object o = s.getEnvironmentValue("Bob'sVar");
            if (o == null)
                System.out.println("Didn't get it back,
what's up??");
```

```
                else {
                     if (!(o instanceof DateTime))
                         System.out.println(
                               "Got something, but it ain't a
     date!!");
                     else System.out.println("Got " + o);
                     }
            catch (Exception e) {e.printStackTrace();}
            finally {NotesThread.stermThread();}
            }
     }   // end class
```

This is a simple example, but there are a couple of points worth illuminating. First, notice that all the code is in the **main()** function. We don't really need to start up another thread here, but we do need to initialize the current thread for Notes, since we aren't creating an instance of NotesThread anywhere. We do that using the static init and term methods discussed earlier. Note that the stermThread() call is in a *finally* block to ensure that it gets executed, even if there's an exception.

Second, we create a DateTime object and initialize it with a valid Notes date expression, "today." We could have also used "yesterday," "tomorrow," or any valid date format. After setting and retrieving the environment variable's value, we use *instanceof* to make sure that we got a real DateTime object back. Note that the negating operator "!" is outside a set of parentheses. That's because *instanceof* is of lower operator precedence than !; if we didn't have the parens, then we would get a compiler error saying that "o" is not a boolean type, and therefore, "!o" is invalid.

Notice also that we can just pass our DateTime object to System.out.println(), and it will print out today's date. Why does that work? Because, like several other objects in the NOI package, DateTime overrides the implementation of the toString() method, which belongs to the Object class. We can decide that printing the value of a DateTime instance means printing the actual date value. If we didn't override toString(), we'd get some weird object reference stringification from Java, which is pretty useless to us.

java.util.Vector freeTimeSearch(lotus.notes.DateRange startend, int duration, Object namelist, boolean findfirst)

This method allows you to determine the blocks of time that are available on the calendars of the people and/or group(s) you specify. The *startend* argument is a DateRange specifying the window in which you want the search to take place. The starting and ending times can be minutes, hours, days, or years apart. *Duration* is the number of minutes you want to be available for each person. If you're trying to schedule a one-hour meeting, you will enter 60, for example. Next comes the name or names that you want the system to search for. We used Object as the input type here for maximum flexibility: You can enter a single String, or a Vector containing any number of Strings. If you use one or more group names, Notes will expand each group and search for all members of the group. The "findfirst" boolean specifies, if *true*, that you just want the first available time returned. If you specify *false,* all available times within the window are returned.

The return value is a Vector containing zero or more DateRange instances. If you specified that you only wanted the first match, the Vector will have at most one DateRange in it; otherwise, it will have as many DateRanges as there are available blocks of time for all participants whose names you supplied. If no available times were found for all participants, then an empty Vector will be returned.

java.util.Vector evaluate(String expression)

java.util.Vector evaluate(String expression, lotus.notes.Document context)

In LotusScript NOI Evaluate is a language construct, really a built-in global function (not associated with any object instance). The purpose of it is to pass through to the host application a "macro" expression, usually some legacy language that the host product uses. In the case of Notes/Domino, you would use Evaluate to execute an @function formula from LotusScript. (Notes inventor Ray Ozzie once referred to the Evaluate feature as "kind of like coding inline assembler for LotusScript." Shows you where he's coming from.)

Because Java doesn't have global functions (and even if it did, it would be *so* un-objectoriented to use them), we put an Evaluate method on the Session class instead. The two flavors of the Evaluate() method each take as input a String containing the formula you want evaluated. You can optionally supply a Document instance that is taken as the context for the formula. This allows you to specify field names in the formula, and the values for those fields will be taken from the document you supply.

The return value is a Vector containing the results of the formula. A Vector is needed because some formulas return lists of values, while others return scalar values. You have to use the various methods on Vector to determine how many values there are, and what kind. Let's try a simple example.

The example in Listing 2.3 uses the Session.evaluate() call to get the value of a Notes @function formula.

Listing 2.3 Using Evaluate Example (Ex23Eval.java)

```
import java.lang.*;
import java.util.*;
import lotus.notes.*;
public class Ex23Eval {
    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("", "names.nsf");
            View v = db.getView("People");
            Document doc = v.getFirstDocument();
            String formula = "@created";
```

```
                java.util.Vector vec = s.evaluate(formula, doc);
                String result = vec.firstElement().toString();
                System.out.println("Formula result = " +
result);
                }
            catch (Exception e) {e.printStackTrace();}
            finally {NotesThread.stermThread();}
            }
}  // end class
```

In this example, we find the first entry in the People view of the machine's address book. This Document instance serves as the "context" for the formula. Not all formulas need a context, but providing a document lets you use field names and so on that can only apply to a specific document. In this case, we know that the result of the formula will be a single value, so we can just pull the first entry out of the Vector and convert it to a String. Date values are returned by evaluate() as lotus.notes.DateTime instances, and the toString() method is implemented for that class.

The lotus.notes.DbDirectory Class

DbDirectory is the class you use to navigate databases on a machine. You provide a Notes server name when you create the object (using getDbDirectory() on the Session). Then you call the getFirstDatabase() method with a constant indicating what type of file you're interested in, obtaining a Database instance as the return value. Then you call getNextDatabase() until it returns a *null*.

You can also use DbDirectory to locate a database by name or replica id, locate your default mail database, or open a database only if it has been modified since a specified date.

As with the Session.getAddressBooks() call, Database instances returned by the DbDirectory navigation calls are bound to a real database, but the database is not opened. Some information about the database is available in a cached buffer, even though the database is not open, so you have a high performance way of finding out certain things about a Database instance that doesn't involve all the overhead of opening the file. The cached properties that are available in unopened databases (see the description of the Database class below for details) are:

- Last modification date
- Replica id
- Categories
- Title
- Template name
- Design template

The DbDirectory has only one property, *String getName(),* returning the name of the server. The other methods are listed as follows.

lotus.notes.Database getFirstDatabase(int type)

The constants you can use to select the type of database to search for are all *static final int* members of the class. They are:

- DbDirectory.DATABASE. All NS? files.
- DbDirecotry.TEMPLATE. All NTF files.
- DbDirectory.REPLICA_CANDIDATE. Any database that doesn't have replication turned off.
- DbDirectory.TEMPLATE_CANDIDATE. Any database that might be a design template.

This call does a search of the server's default data directory and all its subdirectories for the type of file you specify. There's no way to have it search a directory that is not the Notes data directory. Each time you call getFirstDatabase(), the search is reset for the new file type you pass in.

Because the internal Notes API mechanism that DbDirectory uses to perform the search is not thread safe, you cannot begin a search with getFirstDatabase() on one thread and then call getNextDatabase() on the same instance on another thread. The method detects this situation, and will throw an exception. I'm told that this will be cleaned up for Domino 5.0. This is the only case in the Domino 4.6 NOI where you are restricted from using certain functions on an object across threads.

lotus.notes.Database getNextDatabase()

Returns the next database in a search. If getFirstDatabase() has not been called, you'll get an exception.

lotus.notes.Database openDatabase(String dbfile)

lotus.notes.Database openDatabase(String dbfile, boolean failover)

Attempt to open the database with the specified name on the server. If the open fails, a Database instance is still returned, but the database will not be open. You can use the isOpen() call to tell for sure whether the call succeeded or not. If you specify *true* for the "failover" argument, then in cases where the server cannot be reached (down, or possibly overloaded), and where the server is a member of a cluster, Notes tries to locate a replica of the same database on another server in the cluster. If it can find one, it will open that one. You can tell if this happened by using the Database.getServer() call, which returns the name of the server the database lives on. If the name returned by getServer() is different from the name of the DbDirectory, then you failed over.

lotus.notes.Database createDatabase(String dbfile)

lotus.notes.Database createDatabase(String dbfile, boolean open)

Create a new database on the server. You must have database creation rights for the machine in question (you always do for your local workstation; for a Domino server, you have to be listed in the **can create databases** field in the server's configuration record). Otherwise, you will get an exception. If you use the flavor of createDatabase() that takes a second argument, you can specify whether the database should be opened as part of this call. The default is to open the database.

lotus.notes.Database openDatabaseIfModified(String dbfile, lotus.notes.DateTime date)

This call is similar to openDatabase(), but will only open the database if it has been modified (design or data). Again, check the isOpen() call.

lotus.notes.Database openDatabaseByReplicaID(String rid)

Given a replica id in string form, this call will attempt to locate a database with that id on the server (the database can have any file name or title, so long as the replica id matches). If no match is found, an exception is thrown. If a match is found, but the database cannot be opened, an exception is thrown. See openDatabase() for details.

static lotus.notes.Database openMailDatabase(lotus.notes.Session session)

This call attempts to locate the current user's default mail database. If you are running your program on a workstation, the location of your mail database comes from your current location setting. If this call is executed from an Agent, the name that's used is the name of the person who last signed (created or modified) the Agent. That name is looked up in the public name and address book, and the person's mail database location is retrieved from there. This call is static because you might not know in advance what server the database is located on.

String toString()

This routine is overridden in DbDirectory to allow you to pass a DbDirectory instance to println(). The server name is returned.

The lotus.notes.Database Class

Database is one of the more functional, and heavily used, classes in NOI. It has a large number of properties and methods, so I've tried to organize them into a few different categories.

Database Properties

This section lists all the properties on the Database class. As always, if there is only a get call for an attribute, then that property is read-only. If there are both a get and a set call, then the property is read-write.

lotus.notes.ACL getACL()

Returns the ACL object for the current database. See below for details on ACL and ACLEntry.

java.util.Vector getAgents()

Returns a Vector containing all the Agents in the database that are visible to the current user. Unless the current user id has Manager access to the database, private Agents belonging to other users will not be included in the list. Each element of the Vector is an instance of the Agent class.

lotus.notes.DocumentCollection getAllDocuments()

Returns a DocumentCollection instance containing all the data documents in the database. Note that DocumentCollection contents are ordered, but not in any way that would be useful to a developer.

String getCategories()

void setCategories(String categories)

The categories referred to here are not the same as the categories you find in some database views. The Database Categories String can be found in the database properties box (select **File/Database Properties** from the Notes menu, go to the **Design** tab in the properties box). If you set a new category string, it is updated to the database immediately.

lotus.notes.DateTime getCreated()

Returns the date/time the database was created on the machine.

int getCurrentAccessLevel()

Returns a constant indicating the access level with which the database is currently open. The possible values are all declared *static final int* in the Database class. The choices are:

- ACLLEVEL_NOACCESS
- ACLLEVEL_DEPOSITOR
- ACLLEVEL_READER
- ACLLEVEL_AUTHOR
- ACLLEVEL_EDITOR
- ACLLEVEL_DESIGNER
- ACLLEVEL_MANAGER

See the following Database Administration section for a description of some additional methods that manipulate access control at the database level.

String getTemplateName()

String getDesignTemplateName()

If the current database is a template (NTF), then you can get the template name (which is not the same as the database name or title) using getTemplateName(). If the current database was created from a template and if it inherits its design from that template, then you can find out the name of the template from which it inherits with the getDesignTemplateName().

String getFileName()

Returns the database file name (the name of the actual disk file). No path information is included.

String getFilePath()

This property returns the "path," or disk location of the current database. Somewhat counterintuitively, you get different results depending on whether the database is local (on the machine where you're running the program) or remote (on a server somewhere). For local databases, you get the full file system path name.

On my Windows NT system, for example, you might get c:\notes\data\names.nsf, or, if the database is in a subdirectory of the default data directory, c:\notes\data\subdir\setup.nsf. If the database is on a remote server, then all you get is the path relative to the default Notes data directory (names.nsf, or subdir\setup.nsf, for example). This was done for security reasons: Notes should not expose the directory structure of servers.

java.util.Vector getForms()

Returns a Vector containing all the Form instances available to the current user. As with Agents, private forms belonging to other users won't be in the list.

lotus.notes.DateTime getLastFTIndexed()

The date the database's full text index was last updated. If there is no full text index, a *null* is returned.

lotus.notes.DateTime getLastModified()

The date and time of the last modification to the database as indicated. Both data and design modifications are included in the last-modified date.

java.util.Vector getManagers()

Returns a Vector containing a list of Strings. Each String is the name of a user with Manager access to the database.

lotus.notes.Session getParent()

Returns the Database's parent Session instance.

double getPercentUsed()

Returns the percentage of the database that is "occupied." This number is the same as what you see when you bring up the database properties box and click on the **% used** button in the **Information** tab.

String getReplicaID()

The replica id of the database, in string format; this is useful if you want to open another replica of the same database on another machine. See DbDirectory.openByReplicaID().

String getServer()

The name of the server that this database lives on, often a hierarchical name.

double getSize()

The current size, in bytes, of the database on disk.

int getSizeQuota()

void setSizeQuota(int quota)

The "quota" for a database is set by a server administrator and represents the maximum size the administrator wishes to allow for a database. This property is different from the user-settable size limit, or maximum size to which the database can grow. The quota is only settable by a user with Administrator privileges on the server. The size is represented in kilobytes. If you set this property, the value is stored in the database immediately.

String getTitle()

void setTitle(String title)

The title of the database, as seen in the database properties box. You must have Designer or above access to the database to set the title. If you set this property, the value is stored in the database immediately.

java.util.Vector getViews()

Returns a Vector containing all the views (and folders) in the database to which the current user has access. As with Agents and forms, private views and folders belonging to other users will not appear in the list.

boolean isDelayUpdates()

void setDelayUpdates(boolean flag)

This property only has an effect when you use it on a remote database. Normally (and when the DelayUpdates property is set to *false*) when documents are updated on a server (a document delete operation also qualifies as an update, by the way), the update is written immediately to disk and the client waits for the update to complete before continuing (a "blocking," or synchronous call from the client to the server).

If you're performing a lot of updates (or deletes) in a tight loop and want to gain some performance throughput, you can set the DelayUpdates property on the remote database to *true.* This has the effect of batching up a series of update operations for later completion by the server, and the client is not blocked for the full amount of time it takes to do the update to disk. There's a risk associated with using this feature, however: If the server crashes between the time you post the update operation and the time the update is written to disk, then the change is lost.

boolean isFTIndexed()

Returns *true* if the database has a full text index.

boolean isMultiDbSearch()

Domino/Notes allows you to create a full text index that includes more than one database. The databases can all reside on a single server, but they don't have to because your index can span multiple servers. A multi-database index lives in a special database that you create. This property returns *true* if the current database has a multi-database index; otherwise, it returns *false.*

The steps for creating a multi-database index are simple:

1. Create a new database from the Search Site template (srchsite.ntf).
2. Hit **Esc** to exit the default search form that comes up first.
3. From the Create menu, select **Search Scope Configuration**. Select the scope of the databases that will be included in the index (Database, Directory, Server or Domain), and the name of the server, directory, and so on as required. This specifies where the indexer should look for

databases. The server name to look on doesn't have to be the current machine, or the machine where your new search site database will live.

4. You need to mark all databases that you want to be included in the index as available for multi-database indexing. For each database that you want included, bring up the database properties box, go to the **Design** tab, and check the **Include in multi-database indexing** box.
5. Go back to your new search site database, and bring up the properties box. Go to the **Full Text** tab, and click on **Create Index**. The multi-database index will be created in the background. This may take a long time, depending on how many databases are included and on how big they are.

Any full text searches that you perform on this database will now implicitly search all databases included in the index. This is a great way to implement a site-searching feature for use by dumb browsers over the Web. See below for further discussion of full text searching.

boolean isOpen()

True if the database is open; otherwise, *false.*

boolean isPrivateAddressBook()

boolean isPublicAddressBook()

When you use the getAddressBooks() call on the Session class, you get back a list of (unopened) Database instances; each represents an address book. If the database was retrieved from the local machine, it's considered a "private" address book. If it was retrieved from a server, then it's a "public" address book. These two calls help you tell the difference. Another way of telling is to look at the string returned by the Database.getServer() call.

Database Design Elements

This section lists some methods that retrieve design elements from a database.

lotus.notes.Agent getAgent(String name)

Use this call to find a specific Agent in the Database by name. If the Agent exists but is a private agent belonging to another user, or if the Agent doesn't exist, then a *null* is returned.

lotus.notes.Form getForm(String name)

Finds a Form by name; it returns *null* if not found.

lotus.notes.Document getProfileDocument(String key1, String key2)

Profile documents were added to Domino/Notes in Release 4.5, mainly to solve certain performance problems. Before Release 4.5, as the use of LotusScript started to mushroom, developers were finding that it was very expensive to store per-user profile information in a database in a clean and robust way. Sure, you could create a special form for the profile information and then create a special view in the database that selected only documents created with that form. But then you also had to go to every other view in the database and modify each of the other view selection formulas to exclude documents created with the profile form. Furthermore, every time you needed to look up someone's profile, you had to go to the profile view and look up the correct document, usually by the user's name. And what if you needed more than one profile per user in a database, maybe for two different workflows or something? In that case, your profile lookup had to be by a multi-field key. All in all, something of a pain.

Profile documents are a huge performance win: They are cached in the server's memory, so multiple lookups do not cause the document to be read from disk every time. They also have a two-level hash key. One is usually a user name, though it can be any string, and the other (if supplied) can also be any string. Thus, looking up a profile document is much faster than finding and accessing a data note in a view. Furthermore, profile documents are design elements, not data notes, and therefore will never appear in any view.

The getProfileDocument() call finds the profile document in the current database that has the same one- or two-level key (the second key is optional—you can specify *null* or ""). If the document doesn't exist, one is created. Once you have a valid profile document reference, you can treat it just like any other Notes Document instance (get/set item values, update, and so on). The only way to tell a profile document from a regular data document is by looking at the Document.isProfileDocument() property. See Chapter 3 for details.

lotus.notes.View getView(String name)

Returns the specified View instance (could be a view or a folder), if it exists and is accessible by the current user. Returns *null* if not found. If the name refers to a folder of type "private on first use," an exception is thrown, because these are not accessible via NOI (mainly because the newly created private folder must live in the desktop file, and there is currently no back-end access to it).

Database Searching

These calls relate to finding documents in a database.

lotus.notes.DocumentCollection FTSearch(String query)

lotus.notes.DocumentCollection FTSearch(String query, int maxdocs)

lotus.notes.DocumentCollection FTSearch(String query, int maxdocs, int sortoptions, int otheroptions)

These three flavors of FTSearch allow you to find all documents that match a query that you supply. The query can be a simple search string, or you can embed in the query special keywords. See the Domino online documentation for a full specification of the query language—there's too much of it to go into here in any detail. The result of performing a full text search is a list of Document instances that match the query. Most of the time this result set is sorted, as we'll see. The default is to sort the contents of the list by *relevance score*, a number between 0 and 100 that more or less indicates how relevant the document is to the query.

If you specify the maxdocs parameter, then no more than that many documents will be returned. This is useful in cases where a query might result in many hundreds of matches, and you don't want to spend lots of CPU cycles processing them all. There is, in any event, an upper limit of 5000 documents in a result set.

There are two parameters relating to searching and sorting options. To specify sort options, use one of the following constants (defined, as usual, as *public static final int)*:

- Database.FT_SCORES. This is the default. It tells Notes to sort the result set by relevance score.
- Database.FT_DATE_ASC. This constant sorts by document modification date, ascending (earliest date first).
- Database.FT_DATE_DES. This constant sorts by document modification date, descending (latest date first).

The sort option is ignored if the database does not contain a full text index.

The other options allow you to specify the granularity of the word matching algorithm used. You can specify one or both of the following constants. If you want both options, you must add the values together:

- Database.FT_STEMS. This constant tells Notes to look for "stem" matches instead of exact matches. For example, if your query is simply "Geek," but you want variants of the word to also match (Geeks, Geekitude, Geekness, and so on), then specify FT_STEMS.
- Database.FT_THESAURUS. This constant tells Notes that synonyms of a word, as found in the system thesaurus file, also count as matches. For example, if your query specifies Geek, then you might want Propeller Head to match as well. It will, provided your thesaurus file contains the link.

These options are ignored if the database does not contain a full text index. In cases where you invoke one of the FTSearch methods on a database that has no full text index, you will get a valid result set. However, the search will take much longer than it would if there were an index, and you don't get any relevance scores or sorting options. Multi-database searches must always have a full text index.

If the database that you perform the search on is a multi-database index, then all the same options apply, but the results are a bit more interesting. Normally the result set returned by a full text search is a simple sorted array of document ids, and possibly relevance scores. With a multi-database search, however, you need to get more information back than just a document id. You need a way to find out which specific database each document is in. Thus, the result set of a multi-database search contains not just document ids, but full doclinks, which also include a database replica id. You don't really need to know this to write a valid program, however. Just access the documents in the returned DocumentCollection as you normally would (see Chapter 3). Be aware, however, that there's a performance implication when you have a list returned from a multi-database search. Instantiating a document from one of these might be expensive, because it might live in a remote database that isn't open in your program's process space yet. If that's the case, NOI will simply generate a Database instance for that database behind the scenes, and then create an instance of the correct document from that database (remember, no NOI object can be instantiated without a valid container). If you later reference that same database, you won't have to pay the performance penalty twice, as the object will already be cached for you.

When we get to our discussion of the Newsletter class (Chapter 6), you'll see a truly cool (and high performance) application of multi-database result sets.

void updateFTIndex(boolean create)

Use this call to create a new full text index for a database, or to update an existing index. In the create case (you specify *true* for the input argument), default indexing options are used. If you're updating an existing index, then the options stored in the index when it was first created are used to update it. This call works for both local and remote databases. If you specify *false* for the **create** option and there is no index in the database, then nothing happens.

The **create** option is ignored if the index already exists. If the database is on a remote server, you cannot create the index (an exception is thrown), but you can update it.

lotus.notes.DocumentCollection Search(String formula)

lotus.notes.DocumentCollection Search(String formula, lotus.notes.DateTime cutoff)

lotus.notes.DocumentCollection Search(String formula, lotus.notes.DateTime cutoff, int maxdocs)

If you like to specify search criteria using the Notes @function formula language, then these calls are for you. Like the full text variety, they return a DocumentCollection instance containing the results of the search (all the documents that match the query). Also like FTSearch, there is a parameter that specifies the maximum number of documents to retrieve (0 means no limit).

Unlike FTSearch, though, there's an input argument that specifies a date/time before which documents are ignored. This is a highly efficient way of reducing the amount of time it takes to perform the search. If you don't want a time limit, then either use *null* for this argument, or pass in a "wildcard" DateTime instance (one for which you have invoked the setAnyDate() and setAnyTime() methods. See Chapter 4).

The result set of a formula search is never ordered.

lotus.notes.Document getDocumentByID(String noteid)

lotus.notes.Document getDocumentByUNID(String unid)

If you know the note id or the universal id of a document in a database, you can retrieve that document using these calls. Both values are available as properties on the Document class (see Chapter 3). Parenthetically, the note id of a document is valid only within the scope of a single database. The universal id of a document is the same for all document instances across all replicas of the database. Thus, if you get the note id of a document in one database and try to use it to find the same document in another replica of that database, you are not guaranteed success. If you use the universal id, however, you are assured long life and prosperity (unless the document was deleted in the other database and replication has not yet occurred between the two).

lotus.notes.Document getDocumentByURL(String url, boolean reload)

lotus.notes.Document getDocumentByURL(String url, boolean reload, boolean relifmod, boolean urllist, String charset, String webuser, String webpswd, String proxyuser, String proxypswd, boolean nowait)

Since Domino Release 4.5, we've had the ability to set up Web retrieval databases. They can be private (on your workstation), or shared (on a server). You location record tells Notes which one to use. These databases are special in that they bring HTML pages in from the World Wide Web and convert them into Notes documents. The HTML syntax is translated to Notes rich text format, and the new documents are stored in the Notes database. You can then use the Notes Client to view them, just as with any document stored in a database.

Of course you can also use the built-in browser in the Notes Client to view the HTML pages directly. The neat thing about the Web Retriever feature, though, is that it can be utilized in (at least) two situations where a normal browser is useless:

1. You can use a background agent running on a Web Retriever enabled server to bring in pages overnight, while you are safely home in bed, or maybe out drinking somewhere.
2. You can use a server-based Web Retriever to view HTML pages when your machine doesn't have a TCP/IP connection. So long as the server is able to get out to the Web, your client machine doesn't have to speak TCP at all—it communicates with the server via any of the other supported protocols using Notes RPC.

You can use the getDocumentByURL feature to explicitly load a page into the (local or remote) Web Retriever database. The simpler version of the call takes only a URL string and a flag indicating whether you want the page forcibly reloaded. If you specify *false* for the reload flag, and if the page you requested is already available in the Web Retriever database, then you just get that document. If you specify *true*, then the page is always fetched from the Internet, even if it is available locally.

The more complicated version of the call allows you to specify more options:

- *boolean relifmod.* Set this parameter to *true* when you want the page to be reloaded only if it has been modified since last stored in the Web Retriever database. Note: This is one of the few cases where I've ever found an error in the Domino online documentation. The description of this call in the Java Programmer's Guide database omits this parameter.
- *boolean urllist.* If you set this flag to *true*, Notes will find all the links on the page that you are retrieving, and create a special item named URLLinks*n* to hold them. Since there might be more than 64 KB worth of links on a page, Notes numbers the URLLinks items in sequence. So the first links item will be called URLLinks1, the second one URLLinks2, and so on. You can use the contents of this item to "worm" the page and follow the links on it. See Chapter 4 to find out how to use text list items. The default value is *false,* as this option can cause lots of extra processing.
- *String charset.* The MIME name of the character set you want Domino to use when processing the page. Domino converts the text in the page from the specified character set to its internal character set (LMBCS). The character set name for U.S. English is "ISO-8859-1," and for Japanese it's usually "ISO-2022-JP." If you don't know or don't care what character set the page is written in, use a *null* for this parameter.
- *String webuser.* Some Web pages require that you specify a site specific user name and password before you can access them. If that's the case, pass the user name here.
- *String webpswd.* The password that goes with webuser.
- *String proxyuser.* If you're using a firewall to protect an internal LAN from acts of piracy and other evil perpetrated over the Internet, then your

proxy server might require you to specify a user name and password. If so, this is where it goes.

- *String proxypswd.* The password that goes with proxyuser.
- *boolean nowait.* Note: This parameter was also omitted from the online documentation for this call. If you specify *true*, then the call returns immediately, and you don't get a Document instance for the page *(null* is returned). The default is *false*. You would use this parameter if you wanted to update lots of pages in your Web Retriever database, but didn't really want to do anything with any of them right away. You might, for example, write an Agent that refreshes a few hundred pages overnight, but that doesn't need to actually look at any of them. If you use this feature, then that Agent will run much faster.

Note that these calls are only valid on a Database instance that is bound to a Web Retriever database. How do you get one of those? Easy! Just use the getURLDatabase() call on the Session class. It locates and returns an instance of the Web Retriever database specified in your location record (if any).

String getURLHeaderInfo(String url, String header, String webuser, String webpswd, String proxyuser, String proxypswd)

The HTTP specification lists a number of Web page attributes that can be accessed without downloading the page itself. To see an up-to-date list of the headers that all Web pages are supposed to support, see the World Wide Web Consortium's site at http://www.w3.org. This call retrieves the value of the specified header from the page whose URL you specify. The other arguments are as described in the getDocumentByURL() call.

Database Administration

This category includes all the rest of the Database methods, most of which have to do with creating, copying or otherwise administering databases, as well as with access control and document creation.

lotus.notes.Document createDocument()

Creates an empty document in the current database and returns an instance of the Document class.

int compact()

Compacts the current database. The database must not be in use by any other user or process (such as the indexer, Agent Manager, or any API program, including any other Java program), or this call will fail. The way compacting works is that NOI closes the current database, and then makes a compacted copy of it to a new disk file, using a temporary name (the replica id and document modification dates are all preserved, as are unread marks). If that part goes well, then the original database is deleted and the new one is renamed to the original file name.

The call returns the number of bytes saved on disk by the operation, if it is successful.

lotus.notes.Database createCopy(String server, String dbfile)

Create a copy of the current database on the specified server, using the specified database file name. The new copy is NOT a replica of the original. A Database instance representing the new database is returned.

lotus.notes.Database createFromTemplate(String server, String dbfile, boolean inherit)

Create a new database using a template (NTF) file. This call will copy all design and data notes from the template file to the new database. This operation is slightly different from a regular database copy, in that the current user id is automatically given Manager access to the new database, regardless of what access it had to the template file.

If you specify *true* for the inherit parameter, then the new database will be marked as inheriting from the template. This means that if the design of the template changes, the design of the inheriting database will be automatically updated by the server. If the new database resides on a workstation, you can force the design to get updated by using the **File/Database/Refresh Design** menu command.

lotus.notes.Database createReplica(String server, String dbfile)

Use this method to create a replica of the current database on the specified server. The new database will have the file name you provide as the dbfile parameter. As usual, you can use "" as the server name if you want the replica created on the local machine.

void grantAccess(String name, int level)

int queryAccess(String name)

void revokeAccess(String name)

Query, set, or revoke the specified user's access to the current database. The level argument is one of the previously listed ACLLEVEL_XXX constants. If you revoke a user's access, be aware that it doesn't necessarily mean they now have no access to the database. Revoke simply removes the specified user name from the access control list of the database, implicitly giving that user default access rights. If you want to exclude any user (or group) from a database, you must grant them an access level of ACLLEVEL_NOACCESS.

boolean open()

This method is useful in cases where you have a Database instance that isn't open: Either you got it from the Session.getAddressBooks() call or you navigated your way to it using DbDirectory.getFirst/NextDatabase(). In the latter case, as described above, some attributes of the database are available even when it isn't open. Others, however, are not, and you will need to explicitly open the database. This method returns *true* if the operation was a success.

void remove()

Deletes the current database from disk. The current user must have Manager access, or the call will throw an exception.

boolean replicate(String server)

Replicates the current database with all databases that have the same replica id on the specified server. Usually you'd have only one replica of any database on a given server, but there's nothing that prevents you from having more than one. The replication is two-way, so changes are both sent to the server and received from the server. There is currently no way to discover programmatically how many documents were transferred as a result of this call. The results are, however, logged in the Notes log. The call returns *true* if the operation was a success.

String toString()

As with many of the other NOI classes, Database overrides the toString() method, returning the same string as the getFilePath() call returns.

The lotus.notes.ACL Class

The ACL class is used to navigate through and manage a database's access control list (thus the incredibly inventive name of the class). The list acts as a container for the individual entries, represented by the ACLEntry class (see below).

You get an instance of ACL from the Database.getACL() property. You can do this multiple times on a single database, and each ACL instance will be independent of all the others. You could, in fact (though this is definitely not recommended), make different modifications to three different instances of an access control list for a given Database instance, and then update each of the three ACLs. Of course, only the changes made in the last ACL instance to save itself back into the database will win.

Note that changes you make to the ACL properties or contents are not stored in the database until you invoke the save() method.

ACL Properties

lotus.notes.Database getParent()

Returns the ACL's parent database.

java.util.Vector getRoles()

Returns a Vector of Strings, where each String is the name of a *role* in the ACL. For those of you not familiar with roles, a role is somewhat different from a user entry. You can create a role named, for example, Reviewer, in a database, and then assign Reviewer status to any number of user or group entries in the access control list. You manage roles in the Notes Client by bringing up the Database Access Control dialog box (**File/Database/Access Control** from the menu), and selecting the **Roles** panel from the list on the left side of the dialog box.

If you have a couple of roles in the database, you can "enable" a role for a given user/group entry by selecting the entry in the list box and then clicking on the role name. A check mark will appear. You can then use the role name in place of an explicit entry name when assigning access control privileges (such as reader lists) in the database.

boolean isUniformAccess()

void setUniformAccess(boolean flag)

The "uniform access" flag on a database, if set, means that all replicas of that database will have identical access control lists. If the flag is not set, it is possible to have each replica of a database have a different ACL setup.

One side effect of using the uniform access feature affects your access to local databases. Normally your client (or any API program, including a Java program) running on your local machine has full access to any local database, because the database's access control list is not checked. This does not apply, however, to any database with the uniform access bit set. Such databases allow only the access specified to the current id in the ACL.

ACL Methods

void addRole(String rolename)

void deleteRole(String rolename)

void renameRole(String oldname, String newname)

Use these calls to create, remove, or rename roles in your database ACL. Changes you make are not stored in the database until you invoke the save() method.

lotus.notes.ACLEntry createACLEntry(String name, int accesslevel)

Creates a new entry in the ACL, with the specified access level. An ACLEntry instance is returned, and you can use the methods and properties on that class to further refine the person or group's access rights. The input access level is one of the ACLLEVEL_XXX constants previously described.

lotus.notes.ACLEntry getFirstEntry()

lotus.notes.ACLEntry getNextEntry(lotus.notes.ACLEntry entry)

lotus.notes.ACLEntry getEntry(String name)

These methods allow you to navigate through the entries in the ACL, or retrieve a specific entry by name. Entries can be people, servers, or group names. If you request an entry name that doesn't exist, the getEntry() call will return *null.*

If you want to iterate through all the entries in the list, use the getFirst/NextEntry() methods. When using getNextEntry(), you must supply the previous entry's instance as an argument. The getNextEntry() call returns *null* when there are no more entries.

void save()

When you instantiate an ACL object, NOI caches in the object instance a copy of the data in the database's access control list. As you make changes to the ACL using the methods and properties on the ACL and ACLEntry classes, your modifications are stored in the in-memory copy. In order to save these changes back to the database, you must invoke the save() method.

Note that there is no replication conflict mechanism for access control lists: Whoever saves last wins. Thus, you need to be careful, especially when writing multi-threaded Java programs, not to overwrite someone else's (or your own) changes.

The current user id must have Designer (or Manager) access to the database in order to update the ACL.

The lotus.notes.ACLEntry Class

This class allows you to modify the attributes of individual entries in the access control list.

ACLEntry Properties

int getLevel()

void setLevel(int level)

The level codes are the ACLLEVEL_XXX constants described earlier in this chapter.

String getName()

void setName(String name)

The Name property of ACLEntry allows you to query the name of the current entry, or (if you call setName() with a new name) rename the entry. If you rename an entry, the new name you provide must be unique in the current access control list, or an exception is thrown.

lotus.notes.Name getNameObject()

void setName(lotus.notes.Name)

The NameObject property is essentially the same functionality as the Name property, except that instead of getting/setting a name as a String, you can get/set an instance of the NOI Name class. The naming of this pair of functions will probably cause conniptions in Beans-oriented builder tools, as the two function names are not symmetrical. Unfortunately, Java (like C++) does not allow you to overload a method name when two definitions of the method differ only by return type. Thus, we couldn't have two getName() functions, one of which returns a String and the other of which returns a Name instance, so we had to come up with another name for the one returning the NOI object. The corresponding setName can be overloaded, as the argument types are different, but the naming should be parallel. Ooops.

lotus.notes.ACL getParent()

Returns the entry's parent ACL instance.

java.util.Vector getRoles()

Returns a Vector containing the names of all roles that have been enabled for the current entry. Using this method is somewhat more convenient that getting a list of all roles from the parent ACL object and then iterating over each one for an entry with the isRoleEnabled() method.

boolean isCanCreateDocuments()

void setCanCreateDocuments(boolean flag)

Specify whether the current entry is allowed to create new documents in the database.

boolean isCanCreatePersonalAgent()

void setCanCreatePersonalAgent(boolean flag)

Specify whether the current entry is allowed to create private Agents in the database. Only users with Designer access can create shared Agents.

boolean isCanCreatePersonalFolder()

void setCanCreatePersonalFolder(boolean flag)

Specify whether the current entry is allowed to create private folders in the database or not. Only users with Designer access can create shared folders.

boolean isCanDeleteDocuments()

void setCanDeleteDocuments(boolean flag)

Specify whether the current entry is allowed to delete documents in the database. You might want to allow someone to create new documents, but not to delete them afterwards. The default behavior is that anyone with author (or above) access can create documents, and anyone can delete a document that she or he created. This flag modifies that behavior.

boolean isPublicReader()

void setPublicReader(boolean flag)

If this bit is enabled for an entry, it means that the user can access (for reading) public documents in the database. The feature was created in Release 4.5 primarily to allow calendar entries in a mail database to be read by other users who normally have no access to the database. This allows me to (for example) execute a free time search (see the description above in the section on the Session class) and find out if people are available to meet with me, even when I have no access to the databases in which their calendar information is stored.

boolean isPublicWriter()

void setPublicWriter(boolean flag)

This feature was also created to support the new calendaring and scheduling functionality in Release 4.5. Just as the public reader access bit allows people to see public (usually calendar related) records in a database that they normally can't access at all, so the public writer bit allows specially designated users to modify public documents in an otherwise impenetrable database. You would allow this kind of access to someone to whom you have explicitly delegated control over your calendar (an option you can set up in the Calendar Profile in the standard mail template for 4.5 and 4.6).

ACLEntry Methods

As with the ACL class, no changes to any ACLEntry instances are stored in the database until the ACL.save() method is invoked.

void disableRole(String rolename)

void enableRole(String rolename)

boolean isRoleEnabled(String rolename)

These methods are used to manage role settings for a given entry. You can find out if a given role is enabled for the current entry, enable, or disable any role. The role name you provide as an input argument must be a valid role in the database's ACL (see the preceding discussion of the ACL class for details). If it isn't, an exception is thrown. Enabling a role name for a given entry is equivalent to checking that role name for the selected entry in the Access Control List dialog box.

void remove()

Removes the current entry from the access control list.

String toString()

Returns the entry's name.

Summary

Congratulations on making it through the first detailed chapter on the NOI classes and methods. Only four more chapters and 18 more classes to go! Take a deep breath, and turn to Chapter 3, where we’ll cover the Document and View classes, among others.