Chapter 3

NOI Part 2: Document, DocumentCollection, View, ViewColumn

In this chapter we'll cover the methods and properties belonging to a few more NOI classes.

The lotus.notes.Document Class

The Document class is in many ways the heart of Domino, because it's where all the data lives. A lot of what developers use the higher level (in the sense of higher in the containment hierarchy) classes for is aimed at sorting, searching, and accessing documents in a database. You get the document or set of documents that you're looking for (or creating), and only then can you go to town and start manipulating actual data values.

I've divided the interface to document into several sections:

- Properties
- Document management methods
- Item and value accessor methods

Don't be cowed by the number of calls in this class, most are very straightforward.

One topic worth mentioning briefly before diving into the interface concerns *items*. Notes represents a document internally in memory as a header data structure that describes the document level attributes. The header points to a chain of value blocks, called items. Each item is self-describing, in that the item data structure tells you the type of value the item contains (number, text, text list, etc.), how big the value is, in bytes, and so on. The item also contains a pointer to the actual value. Items always have a name, though sometimes the name is one with a meaning special to Notes. The special item names are usually (but not always) prefixed with a $.

My main reason for bringing up the topic of items here is that there's often some confusion about the difference between an item and a field. Understanding that before going into the details of the Document interface will avoid some confusion, so let's get that out of the way here.

An item, as I've described above, is a data element that lives in a document. It has some attributes (such as data type, length, summary and name flag settings, and so on), and a value. A field, on the other hand, lives in a form, and is a design element, not a data element. Fields describe the presentation of items in the Notes UI. When you create a field in a form, you specify the display format, font, color, help string, and various other *presentation* attributes. You don't enter any actual data in the form. The way the Notes UI works when rendering a document to the screen or printer is to find all the fields in the form, then look up each corresponding data item by name in the in memory document. Any items that don't have a correspondingly named field in the form are simply not displayed. Any fields that don't have a matching item are left blank.

It is worth remembering that the back-end classes contained in NOI know nothing about fields—they only know about items. This is true of both the Java and LotusScript bindings. A field "Q" in a form might be defined as being of type number, but if you write an Agent or application using NOI which stuffs a string into an item named Q, and your form is used to render that document, a String, not a number, will appear. This might or might not cause you problems, it depends on whether you have data validation formulas or other programs that count on that item being a number. You're on your own with this, since NOI does no checking whatsoever based on form definitions.

Document Properties

java.util.Vector getAuthors()

This property returns the contents of a document's $UpdatedBy item. This item is maintained automatically by Notes and contains the names of all modifiers of the document. The value of $UpdatedBy is a text list, though sometimes it is missing, and sometimes it has only one entry. The text list is translated into a Vector containing zero or more String objects. The names in the item come from the user's id file, so it is often a hierarchical name.

java.util.Vector getColumnValues()

This property is unique, and in many ways is one of the trickiest in all of the NOI. All other NOI calls that return a Vector actually construct homogeneous lists: All the objects in the Vector are of the same type. The getColumnValues() call returns the data that you see when you look at a view in the Notes UI, often called the "summary data." Because each column in the view can be of any type, the list of column values for a document is heterogeneous (well, it doesn't *have* to be heterogeneous, but it usually is). Thus, when you delve into the contents of a Vector returned by this property, you have to be prepared to parse the type of each object (unless you know in advance what all the column types are; but even if you do, good programming practice dictates that you should still check).

Because getColumnValues() returns a list of values as you'd see them in a view, this call returns *null* if the Document instance that you invoke it on doesn't know who its parent View object is. There is only way a Document object can know who its parent View object is: you must instantiate the Document using the View's navigational methods (see below for discussion of the View interface). When a Document instance is created in the context of a View, the View object caches all the summary data in the

Document. If you instantiate a Document object in some other way, directly from the database, for example, then the summary data is not available.

The objects returned in the Vector by getColumnValues() are in the same order as the columns in the view. You can access these values directly, without causing the entire document to be read from disk, so using the column values is a great idea when you're dealing with large documents. Of course, column values don't help you much if the data you want from a Document is not summary data (doesn't appear in the view). Rich text, for one, is never allowed to be marked as summary data, due to its size. You can, however, have computed values in a column (you supply an @function formula to the column description in the view design), and those computed values show up in the stuff returned by getColumnValues() as well.

The descriptions of the View and ViewColumn classes appear later in this chapter. Listing 3.1 is a quick example of a Java application that makes use of column values.

Listing 3.1 Column Values Example (Ex31ColValues.java)

```
import java.lang.*;
import java.util.*;
import lotus.notes.*;

public class Ex31ColValues
{
    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"mail\\bbalaban.nsf");
            View v = db.getView("($Inbox)");
            Document doc = v.getFirstDocument();
            java.util.Vector vec = doc.getColumnValues();
            System.out.println("Found " + vec.size() + "
column values");
            int i;
            for (i = 0; i < vec.size(); i++)
                {
                Object o = vec.elementAt(i);
                String type = null;
                if (o == null)
                    {
                    System.out.println("Object " + i + " was
null");
                    continue;
                    }
                else {
                      switch (i)
                          {
                          case 0:
                          case 3:
                              if (! (o instanceof Number))
                                  {
                                  System.out.println("Values
0 and 3 are " +
                                      "expected to be
numeric icon values");
                                  continue;
                                  }
                              type = new String("Number");
                              break;

                          case 1:
                          case 4:
                              if (! (o instanceof String))
                                  {
                                  System.out.println("Values
1 and 4 are " +
                                      "expected to be
Strings");
                                  continue;
                                  }
                              type = new String("String");
                              break;

case 2:
                              if (! (o instanceof
lotus.notes.DateTime))
                                  {
                                  System.out.println("Value
2 is expected " +
                                    "to be a DateTime
object");
                                  continue;
                                  }
                               type = new
String("lotus.notes.DateTime");
                               break;
                          } // end switch
                     }  // end else

System.out.println("Value " + i + " is type " + type +
                   ". Value = " + o);
                }  // end for
            }  // end try
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
        } // end main
}  // end class
```

This is a bit longer than most simple examples because, as I said before, column values are tricky customers. A few points to note about this program (which you can find also on the CD):

- Java, like C and C++, uses the backslash character ("\") as an escape character. That's why the path specification for the mail database in the third line of the try block uses a double backslash. The Java compiler reduces "\\" to "\" in the literal string.
- The program uses Inbox from my mail database. Note that we're using a View object for Inbox, but the actual ($Inbox) object is a folder, not a view, in the database. That's okay, because programmatically they are treated the same. More on this is covered in the next section on the View class.
- We print out the count of column values, which we get from querying the Vector's size. If a column doesn't have a value for a particular document, NOI inserts a *null* in that slot in the Vector, to keep the values lined up exactly with the columns. You might notice in some databases that there are more columns according to getColumnValues() than you can see in the UI. This can happen when Notes creates "phantom" columns to use for computational purposes. They'll mostly have no content in the Vector.
- If an element in the Vector is *null*, we just skip it. Otherwise, we want to know if it's the type of object we're expecting for that column. We just use the *instanceof* operator to check for that. There are two columns that (for some, but not all Documents) display icons, one for a mood stamp, the other to indicate that the document has attachments. In the view design property box there's a checkbox that says "display value as icon," and you write a formula to return a numeric value corresponding to one of the icons that Notes knows about. Thus, the value in this column should be an int (actually, since Vectors can contain only Objects, not scalars, it would be an Integer object, not an int). However, we wrote the check to test for *instanceof* Number, not Integer. Why is that? Because all numbers are stored as double precision values in Notes. We could have written it to check for Double, but by using Number (which is a base class for both Integer and Double), we're covering ourselves a bit more.
- One of the columns contains a date (the message date). The getColumnValues() call can't return Notes's internal date format here, because it's meaningless in Java. Instead, date values are returned as DateTime object instances. That's what we're testing for in column 2.
- At the end of the for loop we can just pass the object to println(). All of the objects we might expect to find as column values implement the toString() method, so we should get a reasonable value for all of them.

lotus.notes.DateTime getCreated()

lotus.notes.DateTime getLastAccessed()

lotus.notes.DateTime getLastModified()

Returns the date and time the document was first saved to the database (getCreated()), last read by any user (getLastAccessed()), or last changed (getLastModified()).

java.util.Vector getEmbeddedObjects()

The EmbeddedObjects class is normally used for both file attachments and OLE objects. This property, however, will only return the OLE objects (and that means OLE/2, not OLE/1) used in the document. To get all objects, including file attachments, you have to use the getEmbeddedObjects() call on the RichTextItem class.

While this might seem capricious, there really is a good reason for it. Embedded objects conceptually live in rich text items, which is where they are (usually) displayed. The rich text stream in a rich text item contains rendering information for the embedded object, as well as a link to the actual object information item, also stored in the document. At the document level, we only know about the object information items (called $OLEObjInfo); we don't know anything about which rich text fields contain which objects. Notes only stores embedded object information items for OLE/2 objects, not for OLE/1 objects. File attachment information is stored in items named $FILE.

So, if you want to find all the embedded OLE/2 objects in your document, this is an efficient property to use. The getEmbeddedObjects() call works for all operating systems, and you can examine the EmbeddedObject instances that are returned. Activation of OLE objects is restricted to Windows platforms (and Macs where OLE is installed), however.

If what you wanted was to find all the file attachments in a document, this property won't help you. You can use the RichTextItem.getEmbeddedObjects() call, which returns attachments and OLE objects, but there is one situation where even that won't get you everything. In early versions of Notes (before Release 3) file attachments were supported, but the attachment icons were not stored in rich text items. Instead the $FILE items were created in the document, and no rendering was stored at all. The same situation can occur when a file attachment is created from an API program (from C, for example, not through NOI). Unless the programmer explicitly adds an icon to a rich text item, there will be no "link" from any item to the attachment. When the Notes UI comes across a $FILE item in a document that is "unreferenced" by any rich text item, it simply adds an icon for it to the bottom of the document, so that the user knows it's there, and can detach or view it.

How do you get an embedded object instance for a file attachment? Never fear, there is a way. You can get a Vector containing all the Items in a Document using the getItems() property. Then you can iterate through that list and locate the Item or Items whose name is "$FILE." From there you can get the name of the attached file, and then use the Document.getAttachment() call to get an EmbeddedObject instance. See the sample code for this operation under the getAttachment() call below.

See Chapter 4 for more details on the Item, RichTextItem, and EmbeddedObject classes.

java.util.Vector getEncryptionKeys()

void setEncryptionKeys(java.util.Vector keys)

This property is used when you want to encrypt a document in-place in a database. This operation is not the same as encrypting mail that you send to someone else. In the mail case, your message is encrypted as it is sent, and each recipient's public key is used to do the encryption of the copy that is being sent to that person.

When you encrypt a document in a database, you use a special key (or keys) that you create in your id file (File/Tools/Id from the menu, then click on the **Encryption** button). Doing the encryption is a three-step process:

1. Mark the items in the document that you want to be encrypted (use the isEncrypted/setEncrypted calls on the Item class). Only the items that are so marked will actually be encrypted, the rest will remain unencrypted. File attachments can be encrypted if you want, just mark the $FILE Item.
2. Create a Vector and add to it (as String objects) the name or names of the encryption keys that you want to use. Pass that Vector to the setEncryptionKeys() call.
3. Invoke the encrypt() method on the Document instance. The in-memory copy of the document is replaced with an encrypted version. You still have to invoke the save() method for the new version of the document to be written to disk.

Encrypted documents are automatically decrypted when they are accessed. If, at document open time, the current user id does not contain at least one of the keys specified at encryption time, then the encrypted items in the document will not be viewable by that user. The user will be able to see items that are unencrypted only. There is no explicit decrypt() call.

Another option for encryption is to not supply any encryption keys, in which case the encrypt() method will use the current user's public key.

int getFTSearchScore()

This property is only valid for Document instances that have been created as the result of a full text search (see Database.FTSearch() in Chapter 2). If your database has a full text index, then each document retrieved in the search will have a "relevance score" associated with it, a number between 0 and 100. This call returns that score. If the database is not indexed, or if the Document was retrieved in some other way, the getFTSearchScore() call returns 0.

java.util.Vector getItems()

This call returns a Vector containing an Item instance for each item in the Document.

String getKey()

String getNameOfProfile()

boolean isProfile()

These properties all relate to Documents that are "profile documents" (see Chapter 2, Database.getProfileDocument() for a discussion of profile documents). When you retrieve (or create) a profile document in a database you specify one or two "key" strings for it. The keys are hashed and are used internally for fast lookup. The getKey() call returns the first key, and the getNameOfProfile() call returns the second key.

If the current Document instance is not a profile document, or if one of the keys was not supplied at create time, the getKey() and/or getNameOfProfile() calls will return *null.*

Use isProfile() to determine whether a Document instance represents a profile document.

String getNoteID()
String getUniversalID()
void setUniversalID(String id)

Document identifiers can be a confusing topic, but knowing the difference between a document's "note id" and its "universal id" is important. The note id is unique to a document only within the scope of a single database, while a universal id is unique to a document across all replicas of the database. So, for example, if you have two replicas of your mail database, one on ServerA and one on ServerB, a document that exists in both copies of the database might have different note ids, but it will (guaranteed) have the same universal id. *Universal ids* are what the replicator uses to match up documents when it synchronizes two copies of a database.

You'll have noticed that the note id property is read-only, while the universal id property is read-write. That's because the note id is assigned when a document is first saved to the database, and is based on its physical location in the database file. The universal id (often referred to as the UNID) is not related to physical location of the document, and therefore can be settable. A word of warning, however: You really have to know what you're doing when you *set* a document's UNID, or you risk corrupting data in your database. Domino's Calendaring and Scheduling system sometimes sets a document's UNID, but this must be done safely under restricted circumstances.

- Never set a document's UNID to a value that already exists in the database.
- Never change a document's UNID once it has been saved.

The C&S system manipulates UNIDs so that, for example, when you send an invitation to a bunch of users, the invitation document in each recipient's database will have the same UNID as the original meeting document in your database. This makes the handling of responses to invitations much easier. If you want to delve into the mysteries of how it all works, you can examine the (LotusScript) source code in any Domino 4.5 or 4.6 mail template or mail database (in the main navigator expand the *Design* twistie and select **Script Libraries**. Most of the code for processing meeting invitations and responses is in there).

lotus.notes.Database getParentDatabase()

lotus.notes.View getParentView()

Document instances always have a parent Database object, since NOI maintains a strict containment hierarchy. If you navigate to a Document instance from a View, that Document will also have a parent View; if not, it won't. The getParentView() call returns *null* if the Document was instantiated from the Database.

String getParentDocumentUNID()

Sometimes you want to find the top-level parent of a given document, which might be nested seven or eight levels deep in the response hierarchy of the current view. A document is a response to another document if it contains an item named *$REF*. The $REF item contains the UNID of the response's parent document. Listing 3.2 shows how you might code a subroutine that takes a Document instance as input and returns the top-level parent of that Document. Note that this is not a full application or Agent.

Listing 3.2 Finding A Parent Example (Ex32Parent.java)

```
public lotus.notes.Document
FindTopLevel(lotus.notes.Document starting)
{
    String unid;
    Document parent;
    Database db;

    if (starting == null)
        return null;

    try {
        // get the document's parent database. Is starting a
response?
        db = starting.getParentDatabase();
        unid = starting.getParentDocumentUNID();
        if (unid == null)
            return null;

        do {
            parent = db.getDocumentByUNID(unid);
            if (parent != null)
                unid = parent.getParentDocumentUNID();
           } while (unid != null);

// we're done when the parent has no parent
       } // end try
    catch (Exception e) { e.printStackTrace(); }
    finally { return parent; }
}
```

lotus.notes.DocumentCollection getResponses()

When you navigate through a view, you have a lot of flexibility in how you move around (see the description of the View class later in this chapter for details). But if you need a high-efficiency way to collect all the first-level responses to a particular Document instance without going through the view, then you can use the Responses property.

There are two important differences between the way you navigate through responses in a view and with the Document.getResponses() call:

- getResponses() only returns the immediate responses to the current Document. You have to write a loop in order to get the second- and lower-level responses.
- When you navigate through the responses to a Document in a view, you're retrieving them in the order in which they appear in the View (which in turn depends on whether and how you specified column sorting in the view's design). When you get a collection of response Documents from the getResponses() call, the Document instances in the collection are unordered, because the retrieval is not in the context of a View.

Listing 3.3 is an example of how to "drill down" the response hierarchy of a given document. Again, this is just a subroutine, not a full fledged class.

Listing 3.3 Finding Responses (Ex33Responses.java)

```
public int FindLowest(lotus.notes.Document starting)
{
    DocumentCollection dc;
    int level = 0;
    Document doc = starting;
    boolean done = false;
   try {
    // just gets the first one all the time, counts the
levels
    while (!done)
        {
        dc = doc.getResponses();
        if (dc == null || dc.getCount() == 0)
            done = true;
        else {
             doc = dc.getFirstDocument();
             level++;
             }
        }  // end while
    } // end try
    catch (Exception e) { e.printStackTrace(); }
    finally { return level; }
}
```

String getSigner()

String getVerifier()

Putting a digital signature on a document does two things for you: It allows a user of that document to know reliably who signed it, and it allows readers of the document to detect whether the document has changed since it was signed. When a document is signed, the following things happen:

- A *digest* of all items marked for signing is computed. Think of the digest as a very large and reliable checksum.
- The digest and the signer id's certificate information are attached to the document in an item named *$Signature*.

When you ask for the signer or verifier information, the $Signature item is read, and the signature is *verified*, meaning that the saved digest is compared against the current state of the document, and Notes tries to find at least one certificate in your current id that is also in the signature.

The *signer* is the name of the person whose id was used to create the signature. The *verifier* is the name of the certifying authority that owns the certificate which you have in common with the signer, if there is one. If there is no certificate in common, the signature cannot be verified.

These properties will return *null* if any of the following is true:

- The document is not signed.
- The document has been tampered with (modified) since the signature was created.
- Your id has no certificate in common with the signer, and the signature can't be verified.

Note that section signatures are not handled (at this time) by the Document class. Having signed sections in a document is a great feature, but it causes multiple $Signature items to be added to the document. The information necessary to figure out which signature item goes with which section is maintained in the form, not in the document itself; therefore, the back-end Document class has no way to interpret section signatures.

int getSize()

The Size property tells you approximately how many bytes are consumed by the in-memory version of the current Document. I say "approximately" because the returned value doesn't include the size of some overhead data structures. It essentially computes the sum of the sizes of the individual item's values, including the size of attached files. It doesn't count the size of the document and item data structures themselves.

boolean hasEmbedded()

Returns *true* if the current Document contains any $FILE items; otherwise, it returns else *false.*

boolean isEncryptOnSend()

void setEncryptOnSend(boolean flag)

Set this property to *true* if you want the document encrypted automatically when it is mailed. When you mail a document (see the description of the send() method, below), if you've specified that you want it encrypted, Notes creates an encrypted copy of the original document for each recipient specified in the To, Cc, and Bcc lists. Each copy must be separately encrypted because the recipient's public key (obtained on the fly from the mail server's public address book) is used, so that only the recipient can decrypt the message.

boolean isNewNote()

This call returns *true* if the Document has not yet been saved to disk. NOI can tell if a Document exists on disk by whether or not the Document has a note id.

boolean isResponse()

This case returns *true* if the current Document contains an item named $REF, which indicates that it is a response to some other Document. See the description above of the getParentDocumentUNID() and getResponses() calls for more information on responses.

boolean isSaveMessageOnSend()

void setSaveMessageOnSend()

If this property is *true*, the current document will automatically be saved after you send it. Note that the send() method makes a copy of the Document and makes some modifications to the copy before actually mailing it (see the description of send() below). The Document that gets saved is the *original*, unmodified version.

boolean isSentByAgent()

Wouldn't you like to be able to write a mail Agent that filters out junk mail sent to you by Agents? Well, you can do that very easily with this property. Whenever mail is sent by NOI programs a special item named $AssistMail is automatically attached to the message. The isSentByAgent() call looks for this item, and returns *true* if it finds one. This doesn't guarantee that the message was sent by an Agent, of course, since the Document.send() method can be invoked by any LotusScript or Java program. But it does let you know that the message was generated programmatically, and not by a user directly.

The standard *Out of Office* Agent that comes in the standard mail template uses this property to ignore incoming messages generated by Agents.

boolean isSigned()

This property tells you whether the current Document contains an item named $Signature. It doesn't try to verify the signature. Of course, you could have a signed section in the Document and not have the entire Document itself be signed. This property would return a potentially misleading result of *true* in that case.

boolean isSignOnSend()

void setSignOnSend()

If this property is *true*, then NOI will automatically sign all copies of the outgoing message when you send it.

Document Management Methods

lotus.notes.Document copyToDatabase(lotus.notes.Database db)

This method makes a copy of the current Document instance and inserts it into the specified destination Database. All items and attachments belonging to the Document are copied. The copy is not automatically saved in the destination database, you need to invoke the save() method explicitly.

A Document instance for the new Document is returned. If the copy fails, an exception is thrown.

lotus.notes.Document createReplyMessage(boolean replytoall)

This method is designed for use only where the current Document is a mail message. It approximates the behavior of the **Forward** command in the Notes UI: Create a new Document instance in the current Database; render the contents of the current Document into the "body" rich text item on the new Document; get the contents of the

From item on the original Document, add it to the new Document as the SendTo item; if the replytoall parameter is *true,* the contents of the original Document's CopyTo item are appended to the new Document's SendTo item.

There are two additional points worth making about this method:

1. The original Document is *rendered* into the body of the new message. This means that attachments and embedded objects are not copied or transferred to the new message. If what you want is to have a *deep copy* of the document sent to another person, where all the Items in the original are preserved in the message, then you should simply use the send() method instead (see below).
2. The *replytoall* option ignores the contents of the Bcc field in the original message, since by definition we don't know who else received a blind copy.

The new message instance is returned by the method.

void encrypt()

The encrypt() method causes all appropriately marked items in the current Document to be encrypted. See the preceding description of the EncryptionKeys property for a detailed explanation of how documents get encrypted.

After invoking this method, you must also invoke the save() method to have the encrypted version of the Document written to disk. Because a private key is required to encrypt a Document, you cannot invoke this method from a background Agent; the server does not have the private key belonging to the signer of the Agent, and it would be a real bad idea to encrypt using the server's private key.

If no encryption keys are supplied via the setEncryptionKeys() call, the current id's public key is used.

void makeResponse(lotus.notes.Document newparent)

Use this method to make the current Document a response to the Document you provide as a parameter to the call. A $REF item containing the UNID of the provided

Document is simply added to the current Document. You must invoke the save() method to have the modified Document written to disk. If there was already a $REF item on the current Document, it is replaced.

void putInFolder(String foldername)

void removeFromFolder(String foldername)

A *folder* in Domino is almost the same as a view—the difference being that the contents of a view are computed automatically based on a selection formula that the database designer provides, while a folder can have an arbitrary collection of documents in it. In all other respects, folders and views are the same, so NOI doesn't have a separate Folder class.

Name lookups for folders are done in a case-insensitive way.

Folders know what documents are in them by maintaining a list of Document IDs (the note id is used). To add a Document to a folder, use the putInFolder() call. NOI adds the ID of the current Document instance to the folder's list. This means that the current Document can't be new, as Documents that have never been saved to disk don't have note ids. You provide the name of the folder to which you want the Document added, and NOI creates the folder for you if it doesn't already exist. (There is no way currently to specify what view or folder you want the new one cloned from, you get a new folder cloned from the default view in the database.) Adding a Document to a folder that already contains that Document has no effect.

If you want to remove a Document from a folder, use the removeDocumentFromFolder() call, specifying the name of the folder. Removing a Document from a folder that did not contain that Document has no effect.

There is currently no efficient way to tell for sure whether a given Document is in a given folder. Nor is there a way to generate a list of all folders containing a given Document (the link is maintained in the folder, not in the Document). You can, of course, find out what Documents are in a folder by using the View class navigation methods (see the description of the View class below).

One further restriction: This method will generate an error if you specify a folder that has been designated as "private on first use." The reason this doesn't work is a bit complicated: The private on first use feature works from a shared folder definition. The first time a user accesses the folder (usually in the UI by dragging a document to it), a private clone of the folder is automatically created, and the document is placed in the private copy, not in the shared original. When you use NOI to add a Document to a folder, it will detect that the folder is of this special type. Unfortunately, because NOI is a "back-end" service, it can't just create the private folder for you on the fly. Private folders must live in the desktop file on your machine, and NOI doesn't have access to it. Hopefully this will be fixed in a future release.

boolean remove(boolean force)

Use this method to delete a Document from the current Database. If you specify *false* for the *force* parameter, then the delete operation will fail if the on-disk Document has been modified between the time you accessed it and the time you try to delete it (this kind of thing can happen in a client/server architecture—at least Notes gives you a way to deal with it).

If you specify *true* for force, the Document is removed regardless of anyone else's changes.

The method returns *true* if the delete was successful; otherwise, it returns *false.*

boolean renderToRTItem(lotus.notes.RichTextItem destination)

This method takes the current Document and creates a rendering of it in the specified rich text item. The destination rich text item must not be in the current Document. As with the createReplyMessage() call (which, by the way, uses this method to do its work), only a rendering of the source Document is done; attachments and so on are not carried over.

If the destination RichTextItem already contains something, this method will append the Document's rendering to that Item, not replace it.

There have been reported problems with this method not always working correctly, especially when the source Document makes heavy use of subforms and sometimes shared fields. Your mileage may vary.

boolean save()

boolean save(boolean force)

boolean save(boolean force, boolean makeresponse)

boolean save(boolean force, boolean makeresponse, boolean markread)

The save() method in its various flavors allows you to write the in-memory copy of a Document to disk. There are three options that you can specify:

1. **Force**. If you specify *true* for this option, the Document is written to disk whether or not someone else has modified it since you last accessed it from disk. In a multi-user environment, it is entirely possible that between the time you get an in-memory copy of a Document from disk and the time you save it again someone else might write a new version of that Document to the disk. Specifying *true* for the force parameter tells Notes to ignore and overwrite the other user's changes. This is not the friendliest thing to do, maybe, but sometimes necessary. If you specify *false* for this option, and there is a conflict, then the behavior of save() depends on your choice for the *make response* parameter. If you specify *false* and there is no conflict, the save completes normally.
2. **Make response**. When you have a conflict situation and you specified *false* for the force option, then you can have your version of the Document entered into the database as a response to the version that got there ahead of you by specifying *true* for the *makeresponse* parameter. The idea here is to treat the two conflicting versions as a replication conflict, and make one version a response to the other. Because another user got her version in

there ahead of you, her's gets to be the parent Document. If you specify *false* for "force" and *false* for "makeresponse," and if there's a conflict, then the Document is not written to disk.

3. **Mark read**. If you set this option to *true*, the Document is saved and marked as read in the database's unread list. Note, however, that the Document is marked as read *for the current user id only.* Thus, if you use this option from a background Agent running on a server, then whoever signed the Agent last is the one that will see the Document as read, not you.

The function returns *true* if the Document was successfully saved, and *false* otherwise. If you choose to save a Document as a response in a conflict situation and want to know (a) whether the save() was successful, and (b) whether there was a conflict, you can find out by following this little algorithm:

1. Specify *false* for force and *true* for makeresponse in your save() call. If the current Document is already a response, save the contents of the $REF item as a String (use the Document.getItemValueString() call).
2. Test the return value. If you get a *false* back, there was some error condition and the Document was not saved. Stop. If you get *true,* you know the Document was saved and you can proceed to step 3.
3. To find out whether the Document was saved as a response or not, invoke Document.isResponse(). If the original Document that you were trying to save was already a response, then proceed to step 4.
4. Get the current value of the Document's $REF item, as in step 1. Compare the original and the current values of the UNID contained in that item. If they are the same, then your Document was saved with no conflicts. If not, then the current content of $REF is the UNID of the conflict Document. Alternatively, you could use the getParentDocumentUNID() call to get the current value of $REF.

void send(String recipient)

void send(java.util.Vector recipients)

void send(boolean attachform, String recipient)

void send(boolean attachform, java.util.Vector recipients)

Any Document can be mailed, if you specify at least one valid recipient. There are two ways to specify who should receive the message: You can create the appropriate items in the document yourself, or you can pass one or more names in as parameters.

If you want to set up your recipient list(s) yourself, use these item names (you don't have to be case sensitive):

- **SendTo**. The To: list.
- **CopyTo**. The Cc: list.
- **BlindCopyTo**. The Bcc: list.

Each of these items should contain a string or a text list. The easiest way to create these items effectively is to use the Document.replaceItemValue() call. You don't want to use appendItemValue(), because you might then end up with multiple items of the same name, whereas replaceItemValue() ensures that an existing item of that name will be replaced.

Alternatively, you can just pass a String (for one recipient) or a Vector of Strings (for multiple recipients) in the send() call itself. If you do that, then any existing SendTo item is deleted and replaced with the name(s) you provided in the call. Note that existing CopyTo and BlindCopyTo items are NOT deleted by NOI. Therefore, if you are sending a Document that was originally a mail message, you should be real careful to delete any CopyTo and/or BlindCopyTo items that might be on the original message; otherwise you're leaving yourself open to a potentially embarrassing situation.

There is currently no way (using this method) to send only blind copies or cc's; you must have at least one name in the SendTo item. The Notes UI enforces this restriction as well. To work around it you can always just put yourself in the SendTo item, and everyone else in the BlindCopyTo item.

Here are the steps that the send() method goes through to send mail for you:

- First send() makes a new in-memory copy of your Document. This ensures that any changes it makes to the document don't have to be undone later.
- Check to make sure there is at least one entry in the SendTo item, or that one was provided as an argument. If a recipient list was passed in, replace any existing SendTo item in the copy with the new list.
- For each of the recipient lists on the document (SendTo, CopyTo and BlindCopyTo), make sure that all names in each list is in canonical format. This means that any abbreviated hierarchical names (e.g., Bob Balaban/Looseleaf) are expanded to the proper distinguished name format (CN=Bob Balaban/OU=Looseleaf).
- If you specified that you wanted the form attached to the message, send() then looks for an item named "Form" in the Document. If there is one, and if it contains a string, send() then looks in the Document's database for a form of that name. If it finds one, it copies all the relevant form items into the Document. It then deletes the Form item from the document (it turns out that if you both attach the form and have a form name specified, Notes will use the form name only when opening the document).
- Check the **SaveOnSend**, **SignOnSend**, and **EncryptOnSend** property settings.
- Attach the $AssistMail item, indicating that the message was generated programmatically (see the above description of the isSentByAgent() call), unless one was already present.
- Last but not least, send the message and throw away the copy. If the **SaveOnSend** property was set, save the original document to disk.

Note that you don't get to specify the contents of the From item in the message you're sending. If you're calling send() from anything other than a server Agent, then From will contain the name of the current id. For background Agents, From will contain the name of the signer of the Agent (also known as the *Effective user id*), not the server name. In earlier releases of Domino/Notes, the server name was used in this case, but it caused problems for many users. Most servers do not have their own mail databases, and things like return receipts and nondelivery messages that were generated by recipients of NOI-generated mail were bouncing all over the place because they had nowhere to land. So, we changed NOI to use the effective user name instead.

void sign()

This method attaches a digital signature to the current Document instance. If you want the modified Document saved to disk, you have to invoke the save() method explicitly. See the above description of the getSigner()/getVerifier() calls for more information on how digital signatures work.

NOI uses the current Notes id file to create the signature; therefore, this method will throw an exception if you invoke it from a background Agent, as it does with the encrypt() method.

Item and Value Accessor Methods

lotus.notes.Item appendItemvalue(String itemname)
lotus.notes.Item appendItemvalue(String itemname, double value)
lotus.notes.Item appendItemvalue(String itemname, int value)
lotus.notes.Item appendItemvalue(String itemname, Object value)

These methods give you the ability to create new Items on a Document, and to optionally put a value in the new Item. Note that no checking is done to see if an Item of the same name already exists. You are free to append as many Items of a given name as you wish, but you should be aware that the Notes UI will only display the first one it finds (recall our discussion of Items vs. Fields at the beginning of this chapter). Furthermore, the getFirstItem() call will only return the first instance of the name you provide it. It is possible to get all instances of an Item with a duplicated name via the getItems() property.

The flavor of appendItemValue() with only a single argument creates an empty Item. You can use the Item class's methods later to fill in a value, or you can leave it empty. The versions of the call that take an int and a double both store the numeric value in Notes's internal format, which is double precision.

For all other data types, pass an Object instance in for the "value" argument. Allowable data types are:

- Any numeric type (Integer, Double, Float, etc.). Stored as double precision.
- A String. Converted to the Notes internal character set (LMBCS) and stored as an Item of type TEXT.
- An instance of DateTime or DateRange. Stored as a date or date range.
- A Vector containing one or more Number objects. Stored as a number list.
- A Vector containing one or more String objects. Stored as a text list.
- A Vector containing one or more DateTime or DateRange instances. Stored as a date list or a date range list.
- An Item instance. The value of the provided Item is copied to the new Item.

All Items created this way have their *Summary* flags set automatically (unless the Item is too large, greater than about 15KB), making it possible for their values to appear in a view in the UI.

The new Item instance is returned by the method.

boolean computeWithForm(boolean dodatatypes, boolean raiseerror)

When you design a form for a Notes or Domino application, you get all sorts of opportunities to add automatic processing to the form: input validation formulas, data translation formulas, default value formulas, and formulas to compute a field value on the fly, either when the document is composed or when it is displayed. Frequently, execution of these formulas causes values to come or go in other fields on the document.

Wouldn't it be nice if there were a way to get this to happen when you create a new Document, or process an existing Document using NOI? Yes, and luckily we have the computeWithForm() method. ComputeWithForm() (or as we affectionately call it, CWF) will process the current Document instance according to the form specified in the Form item. If there is no Form item in the Document, or if the specified form can't be found, an exception is thrown. Assuming the form is accessible, CWF executes all the appropriate formulas in the form until it encounters an error, at which point it stops.

You get to specify whether you want an exception thrown when an error is encountered. If you pass *false* for the *raiseerror* argument, then you need to check the return value to see whether the operation succeeded.

You can also decide whether you want CWF to validate the contents of existing Items in the Document according to the data types specified for the corresponding fields on the form. If you select *true* for this option, then CWF checks all the data types in all the Items in the Document and compares them against the field specs in the form. If they differ, it's an error.

CWF will frequently cause new Items to come into existence in a Document, or modify existing ones. You should be sure to invoke save() on the Document to preserve these modifications.

There have been reported problems with CWF when the form uses lots of subforms. We hope these problems will be fixed in a future release.

void copyAllItems(lotus.notes.Document destination, boolean replace)

This method copies all the Items in the current Document to the destination Document you pass in as an argument. It is a very efficient call—much faster than, for example, iterating over all the Items returned from a Document.getItems() call and copying each individually. Each Item retains its original name in the destination Document.

If you want to ensure that the copied Items remain unique in the destination, set the Replace argument to *true*. This makes the method run a bit slower, as it has to do its thing in two passes instead of one (the first pass to delete all occurrences of an Item name from the source in the destination, the second pass to do the copy). If, for example, you know that the destination Document is empty (maybe you just created it), then it is safe to not use this option. Otherwise, specifying *true* for Replace is recommended.

lotus.notes.Item copyItem(lotus.notes.Item sourceitem)

lotus.notes.Item copyItem(lotus.notes.Item, String newname)

This call copies the provided Item into the current Document. The source Item can be from another Document in the same database, or from a Document in any other database. You can optionally supply a String, which is used to rename the Item in the current Document.

An instance of the new Item in the current Document is returned.

lotus.notes.RichTextItem createRichTextItem(String name)

The various flavors of appendItemValue() create Items in a Document for numbers, text, and date values. If you want to create a new rich text item, use createRichTextItem() instead. If the name you provide is already used by an Item in the Document, an exception is thrown.

lotus.notes.EmbeddedObject getAttachment(String name)

Earlier in this chapter I wrote about EmbeddedObjects and file attachments, and how you can use calls like Document.getEmbeddedObjects() to get OLE objects, but not file attachments. I promised an example of how to use getFileAttachment() to extract a specific attachment (which might not have an icon in a rich text item in the Document) from the Document. Well, the time has come to deliver. As always, Listing 3.4 is on the CD included with the book, as is the sample database.

Listing 3.4 File Attachment Example (Ex34Attachment.java)

```
     import java.lang.*;
     import java.util.*;
```

```
import lotus.notes.*;

public class Ex34Attachment
{
    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"book\\Ex34.nsf");
            DocumentCollection dc = db.getAllDocuments();
            Document doc = dc.getFirstDocument();
            java.util.Vector vec = doc.getItems();
            int i, j = vec.size();
            for (i = 0; i < j; i++)
                {
                Item item = (Item)vec.elementAt(i);
                String name = item.getName();
                if (name.equals("$FILE"))
                    {
                    // get the name of the attachment from
the Value property
                    String attach = item.getValueString();
                    if (attach != null)
                        {
                        EmbeddedObject eo =
doc.getAttachment(attach);
                        if (eo != null && eo.getType() ==

EmbeddedObject.EMBED_ATTACHMENT)
                            eo.extractFile("c:\\temp");
                        else System.out.println("Couldn't
get attachment "
                                        + attach);
                        }
                    }  // end FILE
                }  // end for
            } // end try
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
    }  // end main
}  // end class
```

The sample database (Ex34.nsf) has the attachment in the rich text item, so we didn't have to go through all this really, but this code will work for Notes Release 2 style attachments, where the attachment icon is not contained in any rich text item.

The only slightly tricky thing about this technique is that you have to get the name of the actual file that is attached. It isn't the Item's name, because all attachment Items are named $FILE. Luckily (well, it wasn't by accident, let me tell you) for Items of type ATTACHMENT, the Value property will return the name of the attached file. You use that name in the Document.getAttachment() call, and voila, you get your EmbeddedObject instance, which you can use to extract the file (see Chapter 4 for a full description of the EmbeddedObject class).

Note too that, as usual when we have constants to deal with, the Item type is a *static final int* defined in the Item class. See the complete description of the Item type in Chapter 4.

lotus.notes.Item getFirstItem(String name)

Use this method to get an Item instance of a given name. Why isn't this method named findItem()? Well, it's a long story. Originally (Notes Release 4.0), we had decided to expose the "real" way Items worked in Notes. Rich text items, in particular, can be strange. They are a single "logical" item, but because of their size they can actually be made up of more than one individual item in the Document; because a single in-memory Item can't be bigger than 64KB. So, the implementation for rich text always has been to chain multiple items of the same name, and keep track of their sequencing.

As I said, the original 4.0 release of NOI exposed this fact by having two calls: getFirstItem() and getNextItem(), where getNextItem() took as an argument an Item. This would allow you to explicitly access, say, the third piece of a multi-part rich text item.

This turned out to be a really bad idea, for two reasons:

1. **Dangerous**. If you were to (accidentally or not) remove the second item of three in a rich text chain, it would give the Notes UI conniptions. Similarly, if you messed with any of the flags or other settings, the UI might get very confused.
2. **No benefit**. There isn't anything valid you can really do with the individual pieces of a rich text item anyway. You can't use it to access, say, just the third paragraph or something.

So, getNextItem() was removed from the interface. But of course, we couldn't change the name of getFirstItem(), as it was already in a released product, and changing the name would break existing LotusScript applications.

And there you have it. You should always treat RichTextItems as a single logical Item, even if you know in your heart that it isn't really implemented that way. It'll be our little secret.

java.util.Vector getItemValue(String name)
double getItemValueDouble(String name)
int getItemValueInteger(String name)
String getItemValueString(String name)

You've already seen one of these value accessors in action in the preceding file attachment example. Together these calls make it very convenient, at the Document level, to get an Item's value in whatever format suits you best. All versions of this call take only an Item name as input.

The first flavor returns *all* the values in the Item, packed up in a Vector. You use this when you have an Item that contains (or might contain) multiple values, such as a text list or a number list. In LotusScript such values are returned as arrays. In Java we use the Vector class. This makes it much easier to parse multi-valued Items, as each separate value is a separate Object instance in the Vector. It's up to you to figure out what the type of each element is, and how many there are, but that's easy: Vector.size() tells you how many, and you can use *instanceof* to test for Number, String, DateTime or DateRange values. There are other ways to find out as well, but they involve actually instantiating the Item object, and those techniques are described in Chapter 4.

The other versions of the call (getItemValueInteger/Double/String) each return a single value. If the Item contains a multi-valued list, then you'll get the first element of the list. If the type you ask for is not compatible with the internal format of the value (e.g., the Item contains a number and you ask for a String), then you'll get a *null* or a 0. If you want the value coerced, it's up to you to coerce it—Java has some great support for that sort of thing.

boolean hasItem(String itemname)

Returns *true* if the specified Item name exists in the Document. This tells you nothing about the type of the Item, or whether it actually contains a value.

void removeItem(String name)

Removes the named Item from the current Document. You must invoke the save() method to have the modification written to disk. All physical Items of the given name are removed, not just the first one. That fact makes this a handy way to, for example, remove all file attachments from a Document, or so you might think. In actual fact, it would be a bad idea to do that. Here's why:

- Removing all $FILE items in this way won't clean up the rich text fields in which the attachment icons are rendered.

- You might be removing attachments that you really don't want to remove, like embedded OLE objects, whose data are also stored as $FILE items.

The right way to get rid of an attachment is through the remove() call on the EmbeddedObject class (see Chapter 4). It does all the right kinds of cleanup.

lotus.notes.Item replaceItemValue(String itemname, Object value)

This is the call to use when you're not sure whether an Item already exists in the Document. Using the appendItemValue() calls can be dangerous, as described above, in cases where the Item name already exists. Using replaceItemValue() instead is safer. It takes as arguments the name of the Item and the new value. For the new value you can pass in any valid Notes data type, or a Vector containing any number of (homogeneous) data values. If the named Item doesn't exist, replaceItemValue() creates it and appends it to the Document.

One point to note is that the value parameter is an Object, so if you want to replace an Item value with a scalar value (such as an integer), you need to *promote* the scalar to an object. All scalar values have a corresponding object type. For example:

```
replaceItemValue("SomeItem", new Integer(7));
replaceItemValue("AnotherItem", new Double(1.11111));
replaceItemValue("StellaNuthaItem", "literal string");
```

Literal strings will be constructed as objects by the Java compiler anyway, so there is no need to do "new String."

Another point worth noting, although most applications won't ever depend on this feature, is that replaceItemValue() replaces only the value of the named Item (if it exists); it doesn't change the order of Items in the Document's chain.

String toString()

Again, as with many of the other NOI classes, the Document class overrides the Object class's toString() method, returning the Document Universal ID (UNID).

The lotus.notes.View Class

Views are to Domino/Notes roughly what tables are to a relational database: They are the objects you use to maintain indices, to collect data in a logical way in one place, and to navigate through that data. Views in Notes also have a UI component—they are what you use in order to see your documents, and to make selections and perform various activities on those documents. Unlike a relational database, Notes views can be *flat* (that is, all documents at the same level), or hierarchical (that is, top level documents, indented responses, indented responses-to-responses, and so on, up to around 10 levels deep). In addition, both flat and hierarchical views can be categorized, where groups of documents are sorted together under a single category, or key.

Because Views maintain indices, NOI can implement fast Document lookups. Because Views also maintain a rich nesting relationship among Documents, NOI implements a rich set of navigational methods in the View class. The View class in NOI acts as both a container and as a navigator of Documents.

I've categorized the View calls into four groups:

1. Properties
2. Document searching
3. Document navigation
4. Miscellaneous methods

View Properties

java.util.Vector getAliases()

String getName()

Views can have both names and aliases. When you create or modify a view, bring up the Design Properties box. You'll see that there's an entry for the name, and another for an alias (you can actually enter several aliases, separated by vertical bars). You can specify which character of the name is hot-keyed in the View menu by preceding it with an underscore. For example, if I name a view "Bob's _View", then the V will be underlined in any menu containing a list of view names, and I can just hit **V** on my keyboard to select it. This is a nice feature, but it complicates view name processing. When you look up a View using the Database.getView() call, for example, you might provide the name with or without the underscore in it, and you expect it to match in either case. NOI potentially has to do many lookups:

- First it scans all view names in the database and attempts to match each with the name exactly as you pass it in.
- If that doesn't succeed, then the view names are scanned again and a match is attempted with your String against the view names with underscores removed.
- If that doesn't succeed, NOI takes your String and removes any underscores, then repeats the first two steps again.

The Name property returns the View's name as it is stored in the Database, meaning that if you entered the name with an underscore in it, that's how it will be returned to you when you call getName(). The same is true for aliases, except that there might be more than one, so get a Vector back instead of a single String.

java.util.Vector getColumns()

Returns a Vector containing lotus.notes.ViewColumn instances. See below in this chapter for a description of the ViewColumn class. The ViewColumn instances are in the Vector in the same order as the columns are defined in the View.

lotus.notes.DateTime getCreated()

lotus.notes.DateTime getLastModified()

Returns DateTime instances, as you'd expect.

lotus.notes.Database getParent()

Returns the View's parent Database instance.

java.util.Vector getReaders()

void setReaders(java.util.Vector names)

Notes lets you attach to a View a list of people who are allowed to access that View. The list is implemented as a simple text list Item attached to the View design document. If you are not in the list (or a member of a group that is in the list), then you will not see the View's name in any menu or navigator, and you will not be able to open that View.

If you can see the View, then you can examine the list, which is returned as a Vector of Strings. Each String is a user or group name. If you have Designer access to the database, then you can set this property as well.

boolean isProtectReaders()

void setProtectReaders(boolean flag)

Let's say you have Designer access to a database, and you want only certain people to use a particular View (maybe it's an administrative view that has sensitive data in it). So you write an application that sets a reader list onto a View using the setReaders() call. You think you're done, but you're not. What's going to happen the next time your database's design is refreshed? Well, Design Refresh is sort of like replication, except that it's one-way: stuff from an updated template (NTF file) is pushed to all databases (NSF files), which "inherit" from it. So if the View that you set the reader list on is updated in the template, a new View design document will be pushed into your database, wiping out your little attempt at access control. Bummer.

Never fear, though. There's a way out: When you set the reader list on the View using setReaders(), all you have to do is call View.setProtectReaders(*true)* as well. This tells the replicator to *not* overwrite your reader list if and when it updates the View design. The "protection" only applies to the reader list, so all other attributes of the View will be updated properly.

String getUniversalID()

Returns the UNID of the View design document.

boolean isAutoUpdate()

void setAutoUpdate(boolean flag)

This property affects the way view navigation behaves in certain cases. Originally (Notes Release 4.0 and 4.1) the View class did not contain this property, which caused some problems, as illustrated by Listing 3.5:

Listing 3.5 File Attachment Example (Ex35AutoUpdate.java)

```
import java.lang.*;
import java.util.*;
import lotus.notes.*;

public class Ex35AutoUpdate
{
    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"book\\Ex35.nsf");
            View view = db.getView("Ex35View");
            Document doc = view.getFirstDocument();

// Loop over all documents, and reset their doc numbers
            while (doc != null)
                {
                int i =
doc.getItemValueInteger("DocNumber");
                doc.replaceItemValue("DocNumber", new
Integer(i + 10));
                doc.save(true);
                doc = view.getNextDocument(doc);
                }  // end while
            }  // end try
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
    }  // end main
}      // end class
```

Have you spotted the problem yet? Try running this program yourself, using the sample source file and database on the CD. The database (Ex35.nsf) has one view, and the view has one column, which is the document number, sorted in ascending order. I've preloaded the database with five documents, numbered from 1 to 5. The algorithm seems pretty straightforward: Iterate over all the documents in the view from top to bottom, and increment the document number of each by 10.

The problem is that the *while* loop will only execute one iteration. What happens is that the program is modifying an indexed item, and the View's default behavior is to update its index automatically every time that happens. So, we take the first Document (number 1), change the DocNumber Item to 11, and save the Document. The View immediate re-sorts itself so that the newly modified Document is at the end (the highest numbered Document before was only 5). Then we call getNextDocument(), using Document number 11 as the "current" one. Well, there is no Document following number 11 in the View, so the loop terminates.

This isn't a good situation—you could be damaging your database. For example, saving the modified Document might relocate it *higher* in the View, or even lower down, but in the middle of some other Documents. If that were to happen, some Documents would be processed twice, others not at all. Very bad.

Of course, once you know that this is happening (estimate how long it would have taken you to figure it out; I'll wager that the answer is "too long"), it's not too hard to code around it. You can, for example, do the getNextDocument() call *before* you invoke save() on the current Document. That'll work okay. Or, you can use the new (in Release 4.5) setAutoUpdate() call to turn off automatic re-indexing. It might improve your program's performance as well, since you won't be re-indexing at every save().

Turning off AutoUpdate effectively means that while you navigate and (if you like) modify Documents in the View, the View's index (and therefore the contents of the View that you can see) remain static. Changes you make (and changes that other users make too) are not reflected in your in-memory "snapshot" of the View (you'll see changes made to Documents by other users, of course). You can do all your processing, and then tell the View to update itself when you're done (use the refresh() call, below).

AutoUpdate is *true* by default, to make the View class's behavior backwardly compatible with Notes 4.0.

boolean isCalendar()

Returns *true* if the current View is a calendar View. You can still navigate and search as usual.

boolean isDefaultView()

This property returns *true* if the current View is the default View in the Database. The default View is the one that you see by default when you double-click on a Database icon in the UI.

boolean isFolder()

The difference between a View and a Folder is small, but important: Documents are either *in* or *not in* a View based on the View's selection formula. If the formula evaluates to *true* for a given Document, then that Document is in the View; otherwise, it is not. Folders, on the other hand, have no selection formula. They maintain a list of the ids of Documents that are in the Folder. Documents can be added to or removed from a Folder at will, in the UI and also via NOI (see Document.putInFolder() and Document.removeFromFolder() earlier in this chapter).

This call returns *true* if the current View instance is a Folder.

View Document Searching Methods

Whereas Databases have many ways of searching for Documents, a View has only two: full text searching, and searching by key.

void clear()

int FTSearch(String query)

int FTSearch(String query, int maxdocs)

The FTSearch calls in the View class work just like the ones in the Database class, though you don't get as many sorting options, and the search only covers Documents in the View. The results work very differently, however. Recall that the Database.FTSearch() calls return a DocumentCollection instance containing the result set. The View class, however, tries to emulate in NOI what you get when you do a full text search on a View in the UI instead.

When you execute FTSearch on a View, the result is that the View now contains the result set, sorted by relevance score. The "new" View is always flat, even if the original was hierarchical. You navigate the result set just the same as the original View: use the getXXXDocument() calls. Some of the navigational calls that apply to hierarchical Views are mapped appropriately when the View is flat. For example, you might use getChild() to get the first child of the current Document in a hierarchical View. In a flat View, getChild() is simply mapped to getNextDocument().

The View.FTSearch() calls return the number of Documents in the result set. You can reset the View to its original contents by invoking the clear() method. As with the Database FT calls, you pass in a String containing the query, and optionally the maximum number of Documents to put in the result set.

lotus.notes.DocumentCollection getAllDocumentsByKey(Object key)

lotus.notes.DocumentCollection getAllDocumentsByKey(Object key, boolean exactmatch)

lotus.notes.DocumentCollection getAllDocumentsByKey(Vector key)

lotus.notes.DocumentCollection getAllDocumentsByKey(Vector key, boolean exactmatch)

lotus.notes.Document getDocumentByKey(Object key)

lotus.notes.Document getDocumentByKey(Object key, boolean exactmatch)

lotus.notes.Document getDocumentByKey(Vector key)

lotus.notes.Document getDocumentByKey(Vector key, boolean exactmatch)

Keyed lookups in a View are pretty fast, and you can specify as many levels of a key as there are sorted columns in the View (only sorted columns are indexed). There are two basic flavors of keyed lookup: one returns all matches (getAllDocumentsByKey), and one returns only the first match (getDocumentByKey). Within each of these two kinds of lookup, the options are essentially the same: You provide a key (single- or multi-valued), and specify whether you want an exact match on that key or not.

In all cases a valid key component is one of the following data types:

- String
- Number
- lotus.notes.DateTime
- lotus.notes.DateRange

The data type of the key component that you supply must match the data type of the column values. If you want to use a single value key (String value is the most common), you can just pass an Object instance of the correct type. For multi-valued keys, you construct a Vector instance, and add the individual values to it. The values in a multi-valued key must be in the same order as the sorted columns in the View.

Set the "exactmatch" parameter to *true* if you want only exact matches on your key. If you specify *false*, then a partial match on string values will succeed (e.g., "A" will match "Alfred" and "Amy" both). All matches are case and accent insensitive.

Of course a View can have sorted columns interspersed with nonsorted columns. For the purposes of key construction, you ignore the nonsorted columns. Let's look at an example.

The database Ex36.nsf (Listing 3.6, as always, on your CD) contains one view named Lookup. The view has four columns: Name, Creation Date, Date, and Sequence. All except Creation Date are sorted. The Date column contains a date range (start/end DateTime pair). Creation Date is just the date that each document was created in the database. We want to first construct a lookup by name only, then do one by a two value key.

Listing 3.6 Multi-Value Key Example (Ex36Lookup.java)

```
import java.lang.*;
import java.util.*;
import lotus.notes.*;

public class Ex36Lookup
{
     public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"book\\Ex36.nsf");
            View view = db.getView("Lookup");

// first try all "A" names
            DocumentCollection dc =
view.getAllDocumentsByKey("A", false);
            System.out.println("Found " + dc.getCount() + "
matches:");
            int i, j = dc.getCount();
            Document doc;
            for (i = 0; i < j; i++)
                {
                doc = dc.getNthDocument(i);
                System.out.println("\t" +
doc.getItemValueString("name"));
                }

            // now let's try for AE Neuman with a date range
            java.util.Vector v = new java.util.Vector(3);
            v.addElement("Alfred E. Neuman");
            DateTime start = s.createDateTime("4/20/52");
            DateTime endt = s.createDateTime("10/12/97");
            DateRange range = s.createDateRange(start,
endt);
            v.addElement(range);
            v.addElement(new Integer(2));
            dc = view.getAllDocumentsByKey(v, true);

j = dc.getCount();
            System.out.println("Found " + i + " multi-key
matches:");
            for (i = 0; i < j; i++)
                {
                doc = dc.getNthDocument(i);
                System.out.println("\t" +
doc.getItemValueString("name"));
                }
            } // end try
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
    }  // end main
}      // end class
```

Let's dissect what's going on here. In the first go around we simply ask for all Documents where the first sorted column (Name) begins with A. By specifying *false* for the *exactmatch* parameter, we're asking for partial matches to be included (this is what you get when you type a string in the view UI, and an edit control pops up. When you hit **Enter**, you get positioned on the first document that matches. The UI only lets you search on the first sorted column, however). The result (using the sample database) is a DocumentCollection containing six matches: three Alfred E. Neumans, two Alfred F. Neumans, and one more Alfred E. Neuman.

Why aren't they in the same order in which they appear in the view? Because DocumentCollections aren't ordered, except in cases where they contain the result of a full text search. If you must get Documents in the order that they appear in the View, you have to use the getDocumentByKey() call to get the first *hit*, then navigate using the getNextDocument() call. It's then up to you to figure out when to stop.

The next section in the example program builds a Vector containing three key values: "Alfred E. Neuman", a DateRange, and "2" for the sequence number. Note that we're totally ignoring the Creation Date column here, since it isn't sorted, and thus does not participate in the key. This time we call getAllDocumentsByKey() with a *true* for the exactmatch parameter. The result is a new DocumentCollection containing a single Document (the second one in the Lookup view), as this is the only Document that matches all three key values.

Now try an experiment: go into the Ex36 database, edit the Lookup view design. Double-click on the **Name** column to bring up the design properties box, and click on the **Sorting** tab. Make the Name column categorized as well as sorted, and save your changes. Then run the Ex36Lookup program again. Lo and behold, instead of retrieving six Documents for the first search, you only get four, all Alfred E. Neumans. What's up with that? Well, the answer is that keyed lookups will not span category boundaries. The first time, when the view was not categorized, all Documents were basically at the same "level" in the View's index. Once we add categorization to a column, the index is split, and a search can't/won't cross category boundaries. This is a good thing to be aware of when designing fancy lookups.

View Document Navigation Methods

lotus.notes.Document getFirstDocument()

lotus.notes.Document getLastDocument()

lotus.notes.Document getNextDocument(lotus.notes.Document doc)

lotus.notes.Document getPrevDocument(lotus.notes.Document doc)

lotus.notes.Document getChild(lotus.notes.Document doc)

lotus.notes.Document getParentDocument(lotus.notes.Document doc)

lotus.notes.Document getNextSibling (lotus.notes.Document doc)

lotus.notes.Document getPrevSibling (lotus.notes.Document doc)

lotus.notes.Document getNthDocument (int position)

These calls let you spin through a View at any level you wish. For flat Views (where View.isHierarchical() returns *false*) the getChild(), getParentDocument() and getNext/PrevSibling() calls don't have much meaning. They are mapped to the appropriate nonhierarchical call (getNextDocument, getPrevDocument(), respectively).

The getFirstDocument() and getLastDocument() calls are very fast—they just zap to the head and tail of the index. The getNthDocument() call has reasonable performance for small values of "position," but because it always has to start at the top of the index and count every entry, it's slow for large values. The getNext/Prev calls are pretty quick as well, since you provide a current Document each time. Each Document navigated to through a View will know its position in the View. Make sure you don't try to pass in a Document instance that didn't get returned by the current View, because NOI will throw an exception.

Note that for the getNthDocument call, the index is 1-based (the first Document is number 1). All the calls return *null* if the requested Document does not exist.

Navigational behavior is also affected by the AutoUpdate property's setting. If AutoUpdate is on, then the View's index is checked for changes each time you make a navigation call. If the index has been changed, it is automatically refreshed before the operation is completed. For Views that live in Databases on a server, the index could be modified by another user, as the index is always a shared thing. It's not as much of an issue for local databases, where you are the only one who could be making changes (unless you have Agents running in the background).

Note that there's no property or method that tells you how many Documents are in the View. The reason for that is essentially performance and scalability: there might be millions of them, and it would be horrendous to just spin through and count. If you have a case where you *really* need to know, then you have to build a document counter into the View itself.

Note also that the navigation methods say nothing about the non-Document entries in a View. If you have a categorized View, you'll see in the UI that each category has its own line, with the Documents belonging to that category grouped underneath. You might also have subtotal and totals in the View. None of these is retrievable through NOI at the present time (Domino 5.0 may have a new tale to tell—stay tuned). When you navigate through a View (for example, start with getFirstDocument(), and then call getNextDocument() until you get a *null* result), category and other non-Document entries are simply skipped.

The trade-off that was made in the object model design was to favor simplicity over functionality: There would have had to be another class returned by the View navigational methods that encompassed any kind of thing that you might find in a View, only one of which would be a Document. At the time that NOI was first designed, for Release 4.0, this was clearly the correct decision (in my opinion). The object model will clearly evolve over time as more functionality is added.

View Miscellaneous Methods

void refresh()

This method causes the View to refresh its in-memory cache of its index. Any changes (made by you or by any other user) will be reflected. You might have AutoUpdate disabled while you iterate through Documents in a View for a while, then you could call refresh() to update the View’s index all at once.

void remove()

Removes the current View from the Database. This change is committed to disk immediately. If you have less than Designer access to the Database, this call will cause an exception.

String toString()

Returns the View's name.

The lotus.notes.ViewColumn Class

The ViewColumn class has no methods, only properties, and all of the properties are read-only. You can use the View.getViewColumns() call to get a Vector containing ViewColumn instances in the order in which they were defined in the View.

ViewColumn Properties

String getFormula()

If the column was defined using an @function formula, this property will return the formula that was used.

String getItemName()

If the column was defined as simply the contents of a particular field (item really, but the term *field* is used in this context in the UI) in a Document, then this property returns the name of the item. All columns have an item name property, but the name is automatically generated for formula columns.

int getPosition()

This represents the column's position within the View (beginning with 1).

String getTitle()

This represents the column's title, as defined in the View.

boolean isCategory()

Returns *true* if the column is categorized.

boolean isHidden()

Returns *true* if the column is hidden.

boolean isResponse()

Returns *true* if this column is set to show only responses.

boolean isSorted()

Returns *true* if the column is sorted.

String toString()

Returns the column title.

The lotus.notes.DocumentCollection Class

The DocumentCollection class is a strange but useful beast. It isn't a generic collection class, because you can't add and remove things at will. You can only instantiate a DocumentCollection instance by executing some kind of Database level search (formula or full text), by retrieving a Document selection set from the Database class (getUnprocessedDocuments(), for example), or by doing a keyed lookup in a View (getAllDocumentsByKey()).

I've broken up the DocumentCollection calls into three groups: *Properties*, *Navigation*, *Other*.

DocumentCollection Properties

int getCount()

Returns the number of Document instances contained in the collection.

lotus.notes.Database getParent()

Returns the collection's parent Database instance.

String getQuery()

If the DocumentCollection was created as the result of a search, this call returns the actual text of the query, whether it's a formula or a full text query. If the collection was not created from a search operation, this property returns *null.*

boolean isSorted()

Returns *true* if the collection is the result of a full text query, where the Documents are sorted by one of the available criteria (relevance score, date, and so on). Formula search results are not sorted. Full text search results are sorted if the Database contains a full text index (or a multi-database index). Collections containing selected Document lists are not sorted.

DocumentCollection Navigation Methods

lotus.notes.Document getFirstDocument()

lotus.notes.Document getLastDocument()

lotus.notes.Document getNextDocument(lotus.notes.Document doc)

lotus.notes.Document getPrevDocument(lotus.notes.Document doc)

lotus.notes.Document getNthDocument(int position)

These calls do pretty much what you'd expect. They return *null* if the requested Document doesn’t exist. The getNthDocument call (as with the method of the same name in the View class) is 1-based. However, the performance characteristics of the navigational calls differ depending on what kind of DocumentCollection you have. This point is worth diving into a bit.

DocumentCollections store essentially two kinds of data structures, depending on how they were created. The *sorted* variety is implemented by storing arrays of data structures. For example, when you get a result set from a full text search that has an index behind it, the Notes API call that DocumentCollection uses gives you back a buffer containing a Note ID and relevance score for each Document. The buffer is sorted by (let's say in this example) relevance score. Because it's an in-memory buffer, there's a practical limit on its size of around 64KB. Thus, full text searches are limited to a result set of 5000 Documents. Multi-database searches contain more information in the buffer for each Document, but the idea is the same.

For nonsorted result sets (e.g., a formula search), the API (different call) gives you back a Notes data structure called an *IDTable*. This is a compacted data structure with its own set of navigational API calls. IDTables can hold hundreds of thousands of entries, and can still be navigated efficiently, especially if you are looking up a particular Note ID.

Knowing this, we can predict which DocumentCollection methods are more efficient for a particular collection:

- Nonsorted collections are best accessed with a call where the Note ID of a reference point is known: getFirstDocument() and getNextDocument(). That's because looking up a given Note ID is very fast, and therefore finding the next one after a known one is also fast. Getting the last Document from an IDTable is slow, because you have to spin through the entire table. This is true also for the getPreviousDocument() call: You have to start at the front of the table and keep going, always remembering the last Note ID until you hit the one you know about. Then you have the previous one. The getNthDocument() call has a similar problem: You have to basically do a getNext() N times.
- Sorted collections are arrays internally, so indexing into the array at any point is okay. All the calls have about the same overhead, though the getNextDocument() getPrevDocument() methods have a tiny bit extra work to do: They have to extract the index of the Document you provide as an argument. No big deal, really.

Note that Document instances are created on the fly as you make the navigational calls. The DocumentCollection class doesn't pre-instantiate each Document and keep a pointer around.

DocumentCollection Other Methods

void FTSearch(String query, int maxdocs)

When you use the Database or View FTSearch() calls, you get back a DocumentCollection instance. In cases where you want to further refine your original search, you can use the DocumentCollection.FTSearch() call. This method takes the original result set as a starting point, and executes a new full text query on it, replacing the original result set with the new one (the Query property is also replaced, and a new Count is calculated).

You can refine the search as many times as you like. One common occasion for doing this is when an initial query results in a huge result set. Rather than traverse thousands of Documents (because it can get expensive to instantiate a few thousand objects when you don't ever really know when Java's garbage collector will get around to freeing stuff up) and filter one by one, it can be much more efficient to narrow the search criteria, since FTSearch makes use of an index, if there is one.

One problem is that if you end up narrowing the search too far, you can't back up easily, as the last result set is destroyed when a new one is created. There's no undoFTSearch() call.

void putAllInFolder(String foldername)

void removeAllFromFolder(String foldername)

If you know that you want all the Documents in a collection either added to or removed from a particular folder, then these methods are the most efficient way to do that.

void removeAll(boolean force)

If you know that you want to delete all the Documents in a collection from the Database, then this is a very efficient way to handle that task. The force parameter has the same meaning as in the Document.remove() method, but there's no makeresponse equivalent in the DocumentCollection version of this operation.

void stampAll(String itemname, Object value)

Given a DocumentCollection instance, this method provides a way to very efficiently "stamp" a single value into each Document, using the Item name that you supply. You don't have to invoke the save() method on each Document either, since this call commits the changes to disk immediately. This last feature can be a two-edged sword: stampAll() bypasses any in-memory Document information and goes right to the disk. Thus, the following program fragment will lose some of your changes:

```
DocumentCollection dc = db.FTSearch("some query");
Document doc = dc.getFirstDocument();
doc.replaceItemValue("subject", "A new Subject");
dc.stampAll("subject", "A different subject");
doc.save();
```

The problem here is that if doc's subject item is modified in memory, then the DocumentCollection sets the subject item of all contained Document instances (on disk, not in memory) to something. Then doc overwrites the subject set by stampAll() with a save() call. Of course, the save() could have occurred immediately after the replaceItemValue(), but then the stampAll() would have wiped out that value anyway.

The moral of the story is: Know your DocumentCollection.

void updateAll()

When you write an Agent that is set up to run on "all documents that are new or modified since the last time the Agent ran," you must deal with the AgentContext.getUnprocessedDocuments() call (see Chapter 5 for details on the AgentContext class, and see Chapter 8 for more detail on writing Agents), which returns a DocumentCollection. When you have a list of unprocessed Documents, you generally iterate through them, and do something to each (or not, as the case may be). In any case, you must then explicitly tell the AgentContext to remove the Documents you're done with from the list; otherwise, they'll reappear next time you run the Agent, and you almost never want that. Thus, AgentContext has a method called updateProcessedDoc(), which does just that.

Well, 99.9% of the time, you want to remove all the unprocessed Documents from the list, whether you actually do something to them or not. I have yet to come across a case where someone legitimately wanted to have a Document appear in multiple invocations of an Agent. That's why we added this method to the DocumentCollection class in Notes Release 4.5: With a single call you can "mark" all the Documents in the unprocessed collection as "processed," instead of having to remember to code individual calls to AgentContext.updateProcessedDoc().

Summary

You've successfully navigated (and I use the term advisedly) the longest chapter in the book. The next one, Chapter 4, discusses still more NOI classes, with emphasis on data types.