Chapter 6
NOI Part 5: Registration, Newsletter, Log

This chapter covers three additional utility classes. The Registration class gives you the ability to create and manage Notes id files, while the Newsletter class automates the generation of Documents containing lists of doclinks. The Log class is used to record the activity of LotusScript and Java programs to mail messages, disk files, or Notes databases.

The lotus.notes.Registration Class

The Registration class is new in Domino 4.6. It allows you to create certifier, server, and workstation ids, to store them to and retrieve them from a server public address book, to re-certify and cross-certify ids, and to switch id files in the current session. You create a Registration instance with the Session.createRegistration() call.

The philosophy behind doing this class was to support small-to-medium id administration activities. For example, you might populate a Notes database with the names and other vital stats of a bunch of new users, then need an automated (yet customizable) way to generate new ids. You'd use the Registration class in a LotusScript or Java program to spin through the source database and generate an id for each user. The object model is such that this becomes pretty straightforward: you "preprogram" your Registration instance with the invariant information (certifier id, server name, type of id, and so on), then call a method with arguments that are specific to each new id.

There are so many options and "switches" that are relevant to generating a new id that it might be helpful to look at an example before going into the reference material on properties and methods. This example (see Listing 6.1) reads a few names out of a database and generates user ids. The registration server is the place where the public address book gets updated. We want to create new mail files automatically for each

person, and add their id file to the address book, as well as have it on disk locally. The certifier id has already been created.

Listing 6.1 Registration Example (Ex61Reg.java)

```
import lotus.notes.*;
public class Ex61Reg
{
    public static void main(String argv•)
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("", "book\\Ex61");
            View v = db.getView("New users");
            Document doc = v.getFirstDocument();
            v.setAutoUpdate(false);
            Registration reg = s.createRegistration();

// set up the reg parameters that are always the same
            reg.setCertifierIDFile("javatest.id");
            reg.setNorthAmerican(true);
            reg.setStoreIDInAddressBook(true);
            reg.setUpdateAddressBook(true);
            reg.setIDType(Registration.ID_CERTIFIER);
            reg.setRegistrationLog("book\\certlog.nsf");
            reg.setRegistrationServer("");

            // expiration date is today + 2 years
            DateTime expire = s.createDateTime("today");
            expire.adjustYear(2);
            reg.setExpiration(expire);

            // get each new user document, check "processed"
            String last;
            String idfile;
            String server = new String("");
```

```
            String first;
            String middle = null;
            String certpw = "notvalid";
            String location = "right here, of course";
            String comment = "Created in Ex61Reg.java";
            String maildb;
            String fwd = null;
            String userpw;
            String orgunit;
            DateTime now = s.createDateTime();

while (doc != null)
                {
                // extract the info we need
                String processed =
doc.getItemValueString("processed");
                if (processed.equals("No"))
                    {
                    last =
doc.getItemValueString("lastname");
                    idfile = new String("c:\\tmp\\" + last +
".id");
                    first =
doc.getItemValueString("firstname");
                    System.out.println("User " + first + " "
+ last +
                            " being processed.");
                    maildb = new String("mail\\" + last +
".nsf");
                    userpw =
doc.getItemValueString("password");
                    orgunit =
doc.getItemValueString("orgunit");
                    reg.setOrgUnit(orgunit);
                    boolean success =
reg.registerNewUser(last, idfile,
```

```
server, first,

middle, certpw,

location, comment,

maildb, fwd,

userpw);
                    if (success)
                        {
                        // save the date
                        System.out.println("ID created.");
                        now.setNow();
                        Item date =
doc.getFirstItem("processeddate");
                        date.setDateTimeValue(now);
                        doc.replaceItemvalue("processed",
"Yes");
                        doc.save(true, false);
                        }
                    else System.out.println("Error: id not
created");
                    } // end !processed
                doc = v.getNextDocument(doc);
                }   // end while
            }  // end try
    catch (Exception e) { e.printStackTrace(); }
    finally { NotesThread.stermThread(); }
    }  // end main
}  // end class
```

Discussion of Example

This is a somewhat lengthy example, mainly because there are so many registration options to deal with. Let's go through it in some detail so that you get a picture of how it works, and then go ahead and browse the following reference material.

The database Ex61.nsf (included in the CD) is used. It contains "new user" records for four people. Each record contains the person's first and last names, as well as his or her organizational unit (a subdivision of the organization; three users are in the "Angels" group, one is in the "Devils" group), and an initial password. Each user will, of course, change her or his password later. Each record also contains a *Processed* keyword field (Yes/No, defaulting to No), and a processed date, which the program will fill in.

Setting up the Registration instance is pretty simple, though the interaction of the various settings can be subtle. Outside the main loop we set up the parameters that don't change from user to user:

- The certifier id to use.
- The encryption type.
- We want to update the server address book with person records for the new users.
- We want to store the id files in the address book.
- The id type (flat/hierarchical) is set to whatever the certifier id is.
- We designate the current machine as the registration server.

Be careful if you use the local machine as the registration server, as in this example: My sample Java program ran under the auspices of my admin id, not under the server id, so the person records contained my name instead of the server's.

We also want to set a rolling expiration date, as opposed to a hardwired one, so we just initialize a DateTime object to "today" and add two years. If we were being good about writing the code so that it would work in any country (believe it or not, "today" is not a universal term), we could create an instance of the International class (Session.createInternational()), and use whatever word was returned by the getToday() call.

Other parameters to the registration method that we'll be calling also don't change per user (certifier password, location, and comment), and these are initialized before the loop as well.

The outer loop in the sample is a simple one: traverse the View for all Documents, and ignore any that have already been processed (contents of the processed Item is not "No"). For each unprocessed Document, we need to extract the per-user information: first and last name, password, and organizational unit. We also construct on the fly an id file name and a mail database name (though we didn't specify that we wanted mail databases to be created, so they won't in this example).

Note that OrgUnit is a property, not a parameter to the registration method; it's the only property that we modify for each user. The registerNewUser() call returns a boolean value of *true* for success. If the id was registered successfully, we want to update that person's record. We update the time stamp in the now DateTime instance by calling setNow(), then get the processeddate Item from the Document, and modify the Item's value from the DateTime object directly. We could, in theory, have used Document.replaceItemValue and also passed the DateTime object there. After saving the current Document with our modifications, we get the next one in the View.

The program runs just fine from a command line prompt. We supplied the certifier password in the code, so Notes doesn't have to prompt us for it (it does prompt if the certifier id you use requires a password and you didn't specify one in the registration call).

Registration Properties

String getCertifierIDFile()

void setCertifierIDFile(String filepath)

- Registration.ID_CERTIFIER. Generate flat names if the certifier id has a flat name; otherwise, you generate hierarchical names. This is the default.

int getMinPasswordLength()

void setMinPasswordLength (int length)

Get or set the location of the certifier id file that you will be using to create new server or user ids. Use either a platform-specific or "canonical" (defined as PC syntax) path name for the certifier id file. The file must be on disk; you can't specify a certifier file that lives in an address book.

boolean getCreateMailDb()

void setCreateMailDb(boolean flag)

Set this option to *true* if you want mail databases automatically created for new user ids at the same time the id is created. The default for this setting is *false*.

lotus.notes.DateTime getExpiration()

void setExpiration(lotus.notes.DateTime date)

Get or set the expiration date for new server or user ids.

int getIDType()

void setIDType(int type)

The id type tells NOI how to generate the names of new users and servers when you create new ids. It is a constant and must be one of the following:

- Registration.ID_FLAT. Generate flat (nonhierarchical) names.
- Registration.ID_HIERARCHICAL. Generate hierarchical names.
- Registration.ID_CERTIFIER. Generate flat names if the certifier id has a flat name, else generate hierarchical names. This is the default.

int getMinPasswordLength()

void setMinPasswordLength(int length)

Get or set the minimum password length that a user must provide when he or she changes their password. This limit does not apply to the initial password that you can specify when you create a new id using the Registration class.

If you don't specify a minimum length, NOI uses the certifier's minimum password length. You can specify 0, but it isn't recommended.

String getOrgUnit()

void setOrgUnit(String name)

Get or set the organizational unit part of the new id's name. This is the "OU=" part of a distinguished name. The organization part of the id ("O=") will, of course, come from the certifying id.

String getRegistrationLog()

void setRegistrationLog(String dbname)

Query or specify the name of the database used to log registration information. If you specify this property, you must specify the name of a database that was created from the certlog.ntf template. NOI adds entries to this database whenever you create new id files.

To be honest, I've never been able to get this property to do anything. It's an open problem report at Iris. Hopefully it'll be fixed in a point release.

String getRegistrationServer()

void setRegistrationServer()

Specify or query the name of the server whose public address book will be updated as a result of creating new user or server ids. If you are running your program on the server that you want to be the registration server, just specify "" for this property. If the registration server is another server, and that server is not available when you run your program or Agent, then NOI will throw an exception.

boolean getStoreIDInAddressBook()

void setStoreIDInAddressBook(boolean flag)

The default behavior is to create an id file on disk, at a location that you specify. Set this property to *true* if you also want newly created ids to be stored in the registration server's public address book. The id will still be created on disk too.

Note that setting this property to *true* has no effect unless you also set the UpdateAddressBook property, because unless a record is created in the database, there's no place to attach the id file.

boolean getUpdateAddressBook()

void setUpdateAddressBook(boolean flag)

Set this property to *true* if you want id creation to automatically update the registration server's public address book when new certifier, server, or user ids are created. A new person/server/certifier record is added to the database.

boolean isNorthAmerican()

void setNorthAmerican()

If this property is *true* (the default), then a North American id will be created. Otherwise, an international id is created. The difference is primarily in the length of the encryption keys used: North American ids have 64 bit keys, while international ids have smaller keys. The international version of Domino/Notes requires the use of international ids, because of United States Department of Defense restrictions on the export of encryption software (write your Senator or Congressperson!).

Registration Methods

boolean addCertifierToAddressBook(String idfile)

boolean addCertifierToAddressBook(String idfile, String password, String location, String comment)

Adds the specified certifier id file to the server's public address book, using the RegistrationServer property to figure out which server to use. If you don't supply a password and the id file requires one, then NOI will prompt you at run time.

If you have set the StoreIDInAddressBook property (and the UpdateAddressBook property), then NOI will attach the id file to the new record in the address book. Returns *true* if successful.

boolean addServerToAddressBook(String idfile, String server, String domain)

boolean addServerToAddressBook(String idfile, String server, String domain, String password, String network, String adminname, String title, String location, String comment)

This method takes an existing server id and creates a new entry for it in the registration server's public address book, using the RegistrationServer property to figure out which server to use. If you don't supply a password and the id file requires one, then NOI will prompt you at run time.

If you have set the StoreIDInAddressBook property (and the UpdateAddressBook property), then NOI will attach the id file to the new record in the address book. Returns *true* if successful.

void addUserProfile(String username, String profilename)

Given the name of a user who already has a Person entry in the server public address book (NOI uses the RegistrationServer property to select which server to check), this call adds the name of a user profile to that person's record. The "profile" name should in no way be confused with "profile documents" referenced elsewhere in this book (see, for example, the Database.createProfileDocument() call in Chapter 2). It refers instead to a "setup profile," stored by name in the server's public address book. The content of a setup profile is a subset of the fields in the Person record. You can use setup profiles to create standard user configurations, and not have to manually add each field for each user.

Adding a profile name to a Person record copies the data from the profile entry to the user's entry in the address book.

boolean addUserToAddressBook(String idfile, String fullname, String lastname)

boolean addUserToAddressBook(String idfile, String fullname, String lastname, String password, String firstname, String middle, String mailserver, String mailfile, String fwdaddr, String location, String comment)

This method takes an existing user id and creates a new entry for it in the registration server's public address book, using the RegistrationServer property to figure out which server to use. If you don't supply a password and the id file requires one, then NOI will prompt you at run time.

If you have set the StoreIDInAddressBook property (and the UpdateAddressBook property), then NOI will attach the id file to the new record in the address book.

boolean crossCertify(String idfile)

boolean crossCertify(String idfile, String certpw, String comment)

The crossCertify() call adds a cross certificate for the specified id file to the registration server's public address book. If the server's certification id requires a password and you do not supply one, NOI will prompt for it at run time.

void deleteIDOnServer(String username, boolean isserverid)

Deletes the id file attachment from the record (either a user or a server) belonging to the name you provide. If the name is a user name, set the "isserverid" parameter to *false*. You must have Editor access to the registration server's public address book. The rest of the record is not modified, only the id file attachment is deleted.

void getIDFromServer(String username, String filepath, boolean isserverid)

If the registration server's public name and address book contains a record for the name you specify (either a user or a server name), and if that record contains an id file attachment, this call will extract the id file to the specified disk location. Use *true* for the

"isserverid" parameter if the name you pass in is a server name. Use *false* if the name you provide is a user name.

void getUserInfo(String username, String mailserver, String mailfile, String maildomain, String mailsystem, String profile)

Embarrassingly, this call does not work at all in the Java NOI (at least it doesn't crash). The point of it (and it does work in LotusScript) was to return information from the registration server's public address book on the specified user entry, using a very efficient lookup mechanism. The fact that this routine didn't work didn't come to light until late in the release cycle for 4.6. It will be fixed in a future point release.

A perfectly good workaround (though a bit less efficient) is to go to the registration server's address book directly (names.nsf), get the People view, and use View.getDocumentByKey() to find the correct record. Then just get the data directly from the appropriate Items.

boolean recertify(String idfile)

boolean recertify(String idfile, String certpw, String comment)

This call re-certifies an expired id file, using the certifier id specified in the CertifierIDFile property. If the certification id requires a password, you can either provide one in the expanded version of this call, or let Notes prompt you for it when you run the program. Set the new expiration date using the Expiration property.

boolean registerNewCertifier(String org, String idfile, String password)

boolean registerNewCertifier(String org, String idfile, String password, String country)

This call creates a new certifier id file. You must specify the RegistrationServer property, this is the server whose public address book will be updated. If you want a new entry made for this new id in the address book, set the UpdateAddressBook property to *true*. If you want the certifier id file attached to the certifier record, set the StoreIDInAddressBook property to *true* also.

The "org" parameter is required, and represents the organization name ("O=" part of a distinguished name) for the id. All user and server ids created with this certifier will have the same organization, if the ids are hierarchical. The organization name must be at least three characters. The "idfile" parameter specifies a disk location for the new id file, it is a required argument. The password is optional, but highly recommended.

This call always creates a hierarchical certifier. If you really want a flat certifier (not recommended), you have to do it through the Notes UI.

The return value will be *true* if the call was successful.

boolean registerNewServer(String server, String idfile, String domain, String password)

boolean registerNewServer(String server, String idfile, String domain, String password, String certpw, String location, String comment, String network, String adminname, String title)

The registerNewServer() method creates a new server id. The "server" (server name), "idfile" (disk location to write the id file) and "domain" (domain name) arguments are required, the others are all optional (though it is highly recommended that you always supply a password too). The location, comment, network, adminname (administrator's name), and title simply go into the appropriate fields of the server record in the registration server's public address book (if you turn on the UpdateAddressBook property). If you set the StoreIDInAddressBook property to *true*, then the new id file is attached to the server record as well as being written to disk.

If you have not set the RegistrationServer or CertifierIDFile properties, an exception is thrown. If the certifier id requires a password, you can either supply it in the registerNewServerCall() or let Notes prompt for it at run time. The type of id generated (flat or hierarchical) is controlled by the setting of the IDType property (default is TYPE_CERTIFIER, which follows the id type of the certifier id).

The return value is *true* if the call is successful.

boolean registerNewUser(String lastname, String idfile, String server)

boolean registerNewuser(String lastname, String idfile, String server, String firstname, String middle, String certpw, String location, String comment, String maildbpath, String fwdaddr, String password)

The registerNewUser() method creates a new user id. The "lastname," "idfile" (disk location to write the id file) and "server" (server name) arguments are required, the others are all optional (though it is highly recommended that you always supply a password too). The "firstname," "middle" (middle name/initials), location, comment, and "fwdaddr" (forwarding address) simply go into the appropriate fields of the person record in the registration server's public address book (if you turn on the UpdateAddressBook property).

If you set the StoreIDInAddressBook property to *true*, then the new id file is attached to the server record as well as being written to disk. If you turn on the CreateMailDb property and supply a location for the "maildbpath" parameter, then a new mail database will automatically be created (on the server specified in the "Server" argument) when the id is generated.

You must supply the CertifierIDFile and RegistrationServer properties. If the certifier id requires a password, you can pass it in in the registerNewUser() call, or let Notes prompt for it at run time. Note that the registration server is not necessarily the same as the user's default server. The registration server is the one where the public address book (typically the master address book for the domain) lives, the "Server" argument in this call only specifies where the user's mail database lives (although the argument is required even if you aren't creating a mail database for the user, because the user's "home" server information is needed in the public address book).

The type of id generated (flat or hierarchical) is controlled by the setting of the IDType property (default is TYPE_CERTIFIER, which follows the id type of the certifier id).

The return value is *true* if the call is successful.

String switchToID(String idfile, String password)

The switchToID() call has only one required argument, the disk location of the id file to which you want to switch the current session. If that id file doesn't require a password, or if you want Notes to prompt for the password at run time, use "" for the Password argument.

The following steps are performed:

- Locate the new id file, read it into memory.
- Validate the input password. If it is missing or incorrect, prompt for a password.
- Close all server and database connections for the current id.
- Make the new id the current one

The return value of the call, if successful, is the name associated with the new id.

The lotus.notes.Newsletter Class

The Newsletter class is used to format a collection of Documents in a couple of different ways. You create a Newsletter instance with the Session.createNewsletter() call, passing in a DocumentCollection instance (required). The Newsletter instance then operates on that collection. The two methods on this class both return new Document instances, one with a rendering of the specified Document in the input collection, the other containing a doclink for each of the Documents in the collection. The Newsletter properties control the selection and formatting of the output.

If you refer back to Chapter 2, where we discussed full text searching on a Database, you'll recall that the DocumentCollection returned by a full text search on a single Database and the one returned by a search on a multi-database index are somewhat different. The first simply contains a list of note ids for all the Documents, which are all in the same Database. The second kind contains the equivalent of a

doclink for each Document in the collection, because each could be in a different Database. If you instantiate a Document from a collection resulting from a multi-database search, you might be incurring a lot of extra overhead, because the referenced Database might not be open yet.

Believe it or not, there's actually a reason for bringing this up now: If you create a Newsletter instance with a DocumentCollection that contains the results of a multi-database search, you get very high efficiency when you generate a *newsletter* Document from it (see below). The reason is that the *doclinks* in the collection are simply transferred to the newsletter; the referenced Documents (and Databases) do not have to be opened. All the information needed to generate the Newsletter is contained in the DocumentCollection.

Newsletter Properties

String getSubjectItemName()

void setSubjectItemName

When you create a *newsletter* (Document containing doclinks for each input Document in the collection), you don't want just a row of doclinks, you need some kind of tag line for each one. And, of course, you'd like each link's tag line to come from the individual Document.

This property tells the Newsletter class which Item on the Document to use for the tag line (for single-database result sets). It should be the name of an Item that contains text (or something that can be coerced to text), and the Item should (ideally) be present in all Documents in the collection. That isn't always possible, though, because the Documents resulting from a full text search might very well come from different Views, and be created using different forms. If the specified tag Item is not found for a given Document, the link is still included, but the tag line will be empty.

This property is ignored when the input collection is the result of a multi-database search. In that case the tag line is formatted automatically (see the description of formatMsgWithDoclinks(), below).

boolean isDoScore()

void setDoScore(boolean flag)

If this property is *true*, the relevance score associated with each Document in the result set is displayed in the newsletter. Be aware, however, that the relevance score is only included in the results of a full text search when the Database on which the search was performed has a full text index. Otherwise the score will always be 0. The default setting is *true*.

boolean isDoSubject()

void setDoSubject(boolean flag)

The property controls (for single-database searches) whether or not a tag line will appear for each doclink in the newsletter. For multi-database search results, the tag line always appears. The default setting is *true*.

Newsletter Methods

lotus.notes.Document formatDocument(lotus.notes.Database destination, int index)

This method takes a single Document in the collection of search results (specified by the *index* parameter, which is a 1-based index), and essentially sets up a new Document instance with the contents of the source Document rendered in it, similar to what you get when you do a Forward operation on a mail message. The steps that this method goes through are:

- Create a new Document instance. If a "destination" Database was passed in, the new Document is created in that Database. Otherwise the new Document is created in the current user's default mail Database. This can be a problem for background Agents, because (a) you might not want the

new Document created in the *signer's* mail Database, and (b) the default mail Database might be on a different server from the one where the Agent is running, and the Agent will therefore be unable to open it (this is a security restriction on background Agents). It is recommended (for performance if nothing else) that you always supply a Database. If you're going to invoke send() (without saving) on the resulting Document, then it really doesn't matter which Database it's created in anyway.

- Create a new RichTextItem named "Body" on the new Document.
- Use the Document.renderToRTItem() method to render the source Document into the new RichTextItem on the destination Document.

The new ("destination") Document is returned if the call is successful.

lotus.notes.Document formatMsgWithDoclinks(lotus.notes.Database destination)

This is the method that creates a newsletter containing a tag line and a doclink for each Document in the collection of search results. The format of the output Document depends on what type of search was done (single- or multi-database), and on how you set up the Newsletter properties.

For single-database queries, you have the option (see the description of Database.FTSearch() in Chapter 2) of sorting the result set by relevance score or by date. The DoScore property controls whether this column will appear in the output Document. Relevance scores will always be 0 if the source Database does not have a full text index. If the Documents are sorted by date, the creation date is used.

If the DoSubject property is set and an Item name was provided in the SubjectItemName property, then the contents of that Item (if it exists on the Document) are used as the tag line. If the Item does not contain text, the contents of the Item are converted to text for you automatically.

For multi-database search results, the sort key column and the tag line column are always displayed. The sort key column is the same as that which exists for single-database searches (either the relevance score or the Document creation date). The tag

line is the "summary" line stored for the Document in the multi-database index, if one can be found. There are cases where the Document has no summary line. For example, the current user's access to the Database where the Document lives was less than Reader, or the Database had no designated default View. In that case, the name of the database (in parentheses) where the Document lives is used as the tag line.

It all sounds kind of confusing, but let's do two examples (one single-database search and one multi-database), and show you some screen shots of how the output is formatted.

The first example (Listing 6.2) takes a simple discussion database, does a search on it, and formats a newsletter with doclinks to the results.

Listing 6.2 Newsletter Example (Ex62News1.java)

```
import lotus.notes.*;
public class Ex62News1
{
    public static void main(String argv•)
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"book\\Ex62.nsf");
            DocumentCollection dc = db.FTSearch("java", 0,
                                    Database.FT_SCORES,

Database.FT_STEMS+Database.FT_THESAURUS);
            Newsletter nl = s.createNewsletter(dc);

            // store results in another db
            Database output = s.getDatabase("",
"book\\Ex62output");

            // pick an item in both main and response docs
```

```
            nl.setDoSubject(true);
            nl.setDoScore(true);
            nl.setSubjectItemName("newslettersubject");
            Document result =
nl.formatMsgWithDoclinks(output);

// add the query and form name as separate items
            result.appendItemValue("query", "java");
            result.appendItemValue("form", "newsletter");
            result.save(true, false);
            } // end try
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
        }  // end main
}  // end class
```

The output newsletter is saved in a different Database (Ex62output.nsf, also on the CD). The newsletter is in Figure 6.1.

Figure 6.1 *Newsletter result from single database search.*

Figure 6.3 shows what happens when we do a similar thing, but use a multi-database index instead.

Listing 6.3 Multi-Database Newsletter Example (Ex63News2.java)

```
import lotus.notes.*;
public class Ex63News2
{
    public static void main(String argv•)
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"book\\Ex63.nsf");
            DocumentCollection dc = db.FTSearch("java", 0,
                                    Database.FT_DATE_ASC,
```

```
Database.FT_STEMS+Database.FT_THESAURUS);
            Newsletter nl = s.createNewsletter(dc);

            // store results in another db
            Database output = s.getDatabase("",
"book\\Ex62output");

            nl.setDoScore(true);
            Document result =
nl.formatMsgWithDoclinks(output);

// add the query and form name as separate items
            result.appendItemValue("query", "java");
            result.appendItemValue("form", "newsletter");
            result.save(true, false);
            } // end try
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
        }  // end main
}  // end tclass
```

Figure 6.2 *Newsletter result from multi database search.*

You'll notice two differences in the output of this example (Figure 6.2): The doclinks are sorted by date instead of by relevance score (that's not an accident—we requested it that way in the second Java program), and the tag lines are different. Instead of some text from the subject item in each Document, the multi-database search results Newsletter shows only the name of the Database in which each Document lives, in parentheses. That's because in each case the matching text (the text that's indexed) is contained in a RichTextItem, not in a plain text Item. Thus, it can't be stored in the search site index. Instead we just get the Database name.

The multi-database (or "search site") index is contained in the Database Ex63.nsf. I went through my *site* (the example databases I put together for this book) and marked a

bunch of the databases as "include in multi-database indexing" by bringing up the Database Properties box, selecting the Design tab and checking off the option. Then I created a "Search Scope Configuration" document in the multi-db index database (Ex63.nsf) and set it to index all relevant database on the server. Then I just created the full text index on Ex63, and ran my Java program.

There is another significant difference between the single- and multi-database search Newsletters: In order to create the single search doclinks, each Document is opened and queried for the relevant information, especially for the tag line. In the multi-database search case, all the information needed to create the Newsletter entry is stored in the search results; the Documents are not opened.

The lotus.notes.Log Class

The Log class is another bundle of utilities, this time aimed at allowing you to conveniently log "actions" and "errors" during the execution of a program. You specify whether you want your log output to go to any of:

- A Notes database
- A mail message
- A disk file
- A network message queue
- The current Agent's log

You create Log instances using the Session.createLog() method, optionally passing a *program name,* basically any string by which you want the Log identified. The program name is prepended to all output messages.

Any Log instance can support multiple simultaneous output streams. You can, for example, invoke openFileLog(), openMailLog(), and openNotesLog() all on the same instance. Each logAction() or logEvent() call will then write output to three streams. No outputs are open by default.

Log Properties

int getNumActions()

int getNumErrors()

Returns the number or errors or actions that have been logged so far.

String getProgramName()

void setProgramName(String name)

Retrieve or reset the program name associated with this Log instance. The name can be set at Log create time in the Session.createLog() call, or can be set/overridden using this property. The program name is prepended to all output messages.

boolean isLogActions()

void setLogActions(boolean flag)

boolean isLogErrors()

void setLogErrors(boolean flag)

These properties control whether errors and actions are actually logged. If they are on, then all calls to logAction() and logError() will result in messages being output. If they are off, then calls to logAction() and logError() are ignored. Useful for turning logging on and off dynamically during a program's execution. Both are on by default.

boolean isOverwriteFile()

void setOverwriteFile(boolean flag)

When you call the openFileLog() method you specify the file that will receive the output messages. If a file of the same name already exists, this property will control whether that file is overwritten, or appended to. The only time this property's setting has any meaning is when the openFileLog() method is called, so make sure you set it first. The default value is *false*.

Log Methods

void close()

Closes the Log and flushes all output. The instance is not destroyed when you do this, so you can reuse the Log by invoking any of the open methods again.

void logAction(String message)

void logError(int code, String message)

Logs actions and errors to the Log instance's output stream or streams. The message can be any String. The error code can be any integer. You can, for example, use the text and error code contained in a NotesException instance that you *catch*. No translation of the error code is done, it is logged as a number.

void logEvent(String text, String queue, int event, int severity)

This event is not often used, but can be useful in certain cases. For one thing, it only works on servers where event reporting is enabled. Secondly, you have to know the name of a relevant event queue when you write the program, or else acquire it somewhere at run time.

When you log an *event* you specify the text of a message, as with actions and errors. You also supply the name of an event queue on which to place the event, and event and severity codes. The valid event codes are:

- Log.EV_ALARM
- Log.EV_ COMM
- Log.EV_ MAIL
- Log.EV_ MISC
- Log.EV_ REPLICA
- Log.EV_ RESOURCE
- Log.EV_ SECURITY
- Log.EV_ SERVER
- Log.EV_ UNKNOWN
- Log.EV_ UPDATE

There is no checking done to to verify the event code you use (so long as it is one from this list), it's pretty much up to you. Make sure that you use something that's reasonable and understood by your intended recipient.

The valid severity codes are:

- Log.SEV_FAILURE
- Log.SEV_ FATAL
- Log.SEV_ NORMAL
- Log.SEV_ UNKNOWN
- Log.SEV_ WARNING1
- Log.SEV_ WARNING2

void openAgentLog()

This call is only valid when the current program is an Agent with a valid AgentContext. If you invoke it from any other kind of program, NOI will throw an exception. If the openAgentLog() call is made on a Log instance, all logged errors and messages will be appended to the Agent's log, which you can view by selecting the **Agent** in the Agent View UI, then selecting **Agent/Log** from the menu.

This call can be made in either foreground or background Agents. The output stream is flushed when the Agent terminates, or when you invoke the close() method.

void openFileLog(String filespec)

This method causes action and error output to go to the specified disk file. If, when you invoke openFileLog(), a file of the same name already exists on disk, then the behavior of this method is controlled by the OverwriteFile property:

- If OverwriteFile is *true*, the existing file is overwritten
- If OverwriteFile is *false*, new output is appended to it.

This method cannot be used from a background Agent running on a server unless the signer of the Agent is designated to have "unrestricted" privileges in the server's entry in the public address book. Restricted Agents are not allowed to access disk files.

The output stream is flushed either when the close() method is invoked, or (if the Log instance is used in an Agent) when the Agent terminates. If you are using the Log instance from an Application, the output stream is flushed when the Session's NotesThread instance terminates.

void openMailLog(java.util.Vector recipients, String subject)

This method tells the Log instance to create and keep track of a Document instance that will be used to collect all output messages. The message text is appended to a RichTextItem named "Body."

You must supply a recipients list when you invoke this method, because there is no way for you to add one later (you never get to see the mail Document). The recipients list is stored in the SendTo Item for later mailing. You can optionally supply a String that is stored in the Subject Item. This is the line that is typically displayed in most mail database Inbox views.

Output to the Document is flushed either when the close() method is called or when the Session's NotesThread instance terminates. The Document is mailed at that time also. There is no way to cancel mailing of the Document created by openMailLog() once you've called this method.

void openNotesLog(String server, String database)

This method causes output to go to the specified Notes database. Each message (action or error) causes a new Document to be created in the database, and the message is split up into multiple Items in each Document.

There is a standard template (alog4.ntf) that you can use to create databases that are suitable for this kind of output. The template contains one view and one form, named "Log Entry". The Items that are written to this form by the Log class are:

- A$LOGTIME. The date/time the message was logged.
- A$PROGNAME. The program name.
- A$USER. The user id.

- A$ACTION. The action message, if any.
- A$ERRCODE. The error code, if any.
- A$ERRMSG. The error message, if any.

You can, of course, have Notes database output go to any database, it's just more convenient to use one generated from the template, or to copy the form from that database to another one that you want to use.

String toString()

Returns the Log instance's program name.

Summary

Congratulations! You've made it all the way through the play by play (blow by blow?) on all 23 Domino classes. Smile! The next chapter talks about how to write Java applications using NOI, and Chapter 8 tells you how to write Agents using NOI.