Chapter 8
Writing NOI Agents

In Chapter 7 we covered a lot of aspects of using Java and NOI to write applications; could writing Agents be very different? The answer is both yes and no. The basics are definitely the same, particularly the use of the NOI classes. There are some differences that are worth exploring in detail, however:

1. The Agent execution environment is worth understanding in some detail because there are benefits and gotchas that you should know about concerning identity, security, timeouts, and so on.
2. Agents get to use the AgentContext class.
3. There are some important special considerations when writing multithreaded Agents.

Each of these points is considered in detail in this chapter.

Agent Infrastructure: The lotus.notes.AgentBase Class

None of the preceding chapters on the NOI interfaces talks about this class, primarily because you never use it directly, unlike the other classes. It is, however, very important to you if you're writing Java Agents for Domino: All Agent programs must include a class that extends AgentBase; this is the *main* class for the Agent.

AgentBase does a lot of work for you:

- It extends NotesThread, so your main class is all set up and initialized for Notes.
- AgentBase creates a Session instance and pre-loads it with an AgentContext instance all set up for the current Agent.
- It creates a java.lang.ThreadGroup instance to which all threads spawned by the Agent automatically belong. This is used for shutting things down when the Agent terminates (see below).

- AgentBase also creates a timer thread to monitor the progress of the Agent. If the Agent exceeds the administrator-set time limit on Agents, the Agent can be killed (the technical term for this is "terminated with extreme prejudice").
- AgentBase handles redirection of the standard Java output streams System.out and System.err to the Notes log and to the Java console. Notes does the same for LotusScript (except that there's no Java console), so that, for example, output from the MsgBox and Print statements are redirected.
- It creates a special output stream using the standard Java class java.io.PrintWriter. This stream is used when an Agent is invoked from the HTTP server. Text output to this stream is cached by the Agent and is returned to the HTTP server as the result of running the Agent. It typically is served back to a browser.
- Finally, AgentBase invokes your class's NotesMain() function, which is where you write your program.

Okay, so why another method (NotesMain) that you have to keep track of for Agents? Mainly because AgentBase itself extends NotesThread, and therefore must implement its own startup and shutdown logic in a runNotes() method. Because we have to be absolutely sure that AgentBase's own runNotes() method gets called from NotesThread, and not your class's, runNotes() is declared *final* in AgentBase. AgentBase's runNotes() method calls NotesMain(), which is where you implement your Agent's logic.

Let's look at a simple example of a single-threaded Agent. This Agent is meant to be run from a View Action on Documents that have been selected in the view. It uses the Newsletter class to format a Document containing doclinks to each of the selected Documents, and mails it to the Agent's author (the person who signed it).

Listing 8.1 Simple Agent Example (Ex81Agent.java)

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

public class Ex81Agent extends AgentBase
```

```
{
    public void NotesMain()
        {
        try {
            Session s = this.getSession();
            AgentContext ctx = s.getAgentContext();
            Database db = ctx.getCurrentDatabase();
            DocumentCollection dc =
ctx.getUnprocessedDocuments();
            System.out.println("Found " + dc.getCount() + "
selected docs");
            Newsletter nl = s.createNewsletter(dc);
            nl.setSubjectItemName("Subject");
            Document doc = nl.formatMsgWithDoclinks(db);
            doc.send(ctx.getEffectiveUserName());
            System.out.println("Newsletter sent to " +

ctx.getEffectiveUserName());
            }
        catch (Exception e) { e.printStackTrace(); }
        }
}  // end class
```

Pretty simple for an interesting piece of functionality, no? We're getting lot of leverage out of AgentBase and AgentContext here:

- The Session has been set up for us, including the AgentContext instance.
- We can get the "current" Database (the one the Agent lives in) from AgentContext.
- We can also get the list of Documents selected in the view from AgentContext.
- System.out.println() is automatically redirected to the Java Console.
- AgentContext also includes the name of the "effective" user, in this case the signer of the Agent.
- If we needed to know something about the current View (from whence the selected Documents came), we could get it from any of the Documents

contained in the collection, by using the Document.getParentView() call. This would require instantiating at least one Document, though.

How did we set this Agent up in the Notes UI? Domino 4.6 does not, unfortunately, include an integrated Java development environment (it's scheduled to appear in Domino 5.0). I wrote the code for Agent using my standard editor, then compiled it using the Java Development Kit (JDK) I downloaded (free! and legal!) from the JavaSoft Web site (http://java.sun.com). From that point I just used the Notes Client to create an Agent in my Ex81.nsf database (select **Create Agent** from the menu). From there, follow these steps:

1. Type in the Agent's name. You can make it either shared or private.
2. Set the Agent to run "manually, from the menu."
3. Set the Agent to run on "selected documents."
4. Select the Java radio button (see Figure 8.1 for a screen shot of the setup)
5. Click on the **Import class files** button, select the **.class file** (or files) that the Java compiler produced for you.
6. Save the Agent. Notes will sign it with the current id.
7. Go into view design for the View from where you want to run the Agent.
8. Create a View Action. Pick the **Add Action** button.
9. For the type of action, select **Run agent**, and specify your new Agent.
10. Save the View.

Figure 8.1 *Setting up a Java Agent.*

At this point you can go into the View and select any number of Documents, then click on the action button at the top. The Agent runs, and the set of selected Documents is delivered to it in the AgentContext.unprocessedDocuments property. The first time you run an Agent like this in a session it will take a little extra time, as Notes has to start up an instance of the Java Virtual Machine (VM). Agents run in this way from the Notes Client will run synchronously, meaning that you can't do anything else in the UI while the Agent is executing. You can interrupt the Agent by hitting **Ctl-Break** on the keyboard.

The last thing the Agent does when it runs is send mail to the author (or last modifier) of the Agent (the *effective user*). See Figure 8.2 for an example of the View and the Java Console on top. See Figure 8.3 for a screen shot of the resulting Newsletter.

Figure 8.2 *Notes view and Java console.*

Figure 8.3 *Newsletter Created By Ex81Agent*

Agent Identity

We keep referring to the Agent's effective user name. As we described back in Chapter 5 in the section on the Agent and AgentContext classes, background server Agents (those triggered on a scheduled basis, by a new-mail or new/modified Document event, or by a URL via the HTTP server) run with the privileges of the last signer of the Agent. Thus, the Agent, when it runs, assumes the identity of the signer, not of the current Notes id, which in the case of a server based Agent is the server's id. To do otherwise would be a security violation: an Agent running with the server's privileges when the creator of the Agent has (most of the time) fewer access rights would potentially cause big problems.

Thus, the *effective use*r is the one whose identity the Agent is temporarily assuming. The Agent cannot do anything on the server that the user herself could not do, in terms of creating or deleting databases, opening databases, creating/deleting/modifying Documents, and so on. There is one additional wrinkle here: When you create an Agent that is meant mainly for Web-based applications (meaning that the server will be accessed from a Web browser, using the HTTP server module, which itself can trigger Agents), you have the option for each Agent of declaring that it will run *either* with the signer's identity *or* with the Web user's identity. You select which option you want in the Agent Design properties box. Go to the Agent View in your database, and right-click on the Agent, then select **Agent Properties**. See Figure 8.4 for a picture of the Agent Properties box.

Figure 8.4 *The Agent properties box.*

There are several choices worth exploring here, but the **Agent identity** option is the last one on the Design tab. What would happen if we make just this one change to the Agent, and then access the database from a browser? The answer is: Nothing, because View Action buttons aren't supported by the Domino HTTP server. We could, however, create a slightly different Agent that better illustrates some of the ways you'd take advantage of the HTTP server (see Listing 8.2).

Listing 8.2 Web Browser Agent Example (Ex82Browser.java)

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

public class Ex82Browser extends AgentBase
{
    public void NotesMain()
        {
        try {
            Session s = this.getSession();
            AgentContext ctx = s.getAgentContext();
            PrintWriter output = ctx.getAgentOutput();
            String st = new String("Hello " +
ctx.getEffectiveUserName());
            output.println(st);
            System.out.println("<B>" + st + "</B>");
            }
        catch (Exception e) { e.printStackTrace(); }
        }
}
```

Note that we write out two messages here: one to the special PrintWriter stream, which goes back to the invoking browser, and one to System.out, which goes to the system log file, and (if we run the Agent from the Notes UI) to the Java Console window. I created

a new Agent in the Ex81.nsf database, called "WebGuy, " and imported the Ex82Browser.class file. Leaving the design properties untouched for the moment, I started my Domino/HTTP server, brought up the Internet Explorer browser, and typed the following URL for my new agent: http://localhost/Ex81.nsf/webguy.

"Localhost" is just a special TCP/IP name for the current machine, so that you can access your Domino server from the same machine that it is running on (my laptop, for example) without the machine having to be connected to a real network. Figure 8.5 shows the result in the browser.

Figure 8.5 *Result of running Webguy Agent.*

Note that we didn't see the default Agent Done response, we saw the message I sent. And what's more, it came out in a bold font. That's because in the output to the PrintWriter stream I prepended a "<B>" HTML tag, so that the text that followed was made bold by the browser. This is an important point: It shows that you can write a Domino Agent to format rich text output for the browser which invokes it, using simple HTML commands. You use standard Java output primitives on the special stream to buffer the stuff up, and Domino serves it all up to the browser when the Agent terminates. Because the returned Agent output is stored in a single hunk of continuous memory, you want to make sure that you don't let it get bigger than 64KB. That should be plenty, however.

Okay, now let's go and check off that **run as web user** option in the Agent's design properties, and refresh from the browser. Figure 8.6 shows the result.

Figure 8.6 *Running Webguy as the Web user.*

Note that this time the user name that the Agent operated under was "Anonymous." That's because I set up my server to allow anonymous access and gave everyone in the world the right to run Agents on my machine (I edited the server record in the public

address book and entered the hierarchical user name wildcard for everyone, "*" in the **can run restricted agents** field.

When the browser accessed the server and caused the HTTP server to run the Agent, HTTP assigned the name "Anonymous" to my connection, because my user session was not authenticated. Authentication can happen either from a browser using *Secure Sockets Layer* (SSL), or by logging in in response to a Domino username/password prompt, neither of which I turned on for my laptop's server. Thus, I received the default user name. I also had to give an appropriate level of access to the Ex81.nsf database via its ACL.

Agent Security

This joke was popular when we started developing agent technologies for Notes Release 4.0 several years ago: What do you call an Agent that has no security controls on it? Answer: a virus.

Sad but true, there are lots of products out there that make a big deal out of having *mobile Agents*, *intelligent Agents* and *wizards* of all kinds, but when you ask if these beasties are digitally signed, you usually get “Oh, we're addressing that.” Or if you ask how much control the server administrator has over these things, the answer is usually something like, "Yes, we'll have that in our next release."

Domino implements two basic security principles that apply to Agents:

1. Agents can run either in the foreground or in the background, but in no case will an Agent have greater access to a server or a database than does the person who created it.
2. Agents running on servers are subject to control by the system administrator so that the running of Agents is not allowed to consume the server machine entirely.

This seems obvious, but you'd be surprised how often these tenets are overlooked in other products. Let's delve into how Domino accomplishes both of these goals in more detail.

Agent Access Control

The two cases (foreground/background) are quite distinct. Agents running in the foreground are being used from a Notes Client, where there is a user id in force. The Agent might be invoked directly by the user (from the Agents View, from an Action button, from a button on a form, etc.) or indirectly because of, for example, a LotusScript program attached to some event (Database or Document open, maybe) that fires off an Agent. In all cases, however, the Agent is being run because of some explicit action performed by the user, and the Agent is always operating under the auspices of the user's id. Thus, if the Agent tries to go out to some server and modify the public address book, it will only succeed if the user has the authority to do that operation.

Background Agent execution can take place either in the Notes Client or on the server. In the Client case, as with foreground operation, the Agent is running under the authority of a user id, and can do anything the holder of that id can do. When an Agent runs in the background on a server, the only id that is present is the server's id, and we almost never want Agents to operate as the server. Instead, server Agents operate with the privileges of the person who created or last modified the Agent (except when they are triggered by the HTTP server and are set up to run with the identity of the Web user, as shown above, which is a special case).

How do we know who an Agent should run as? There's a digital signature attached to every Agent when it is created or modified. If for some reason the signature is missing or can't be verified (someone tampered with the Agent record in the database, or the signature has no certificates in common with the server where it is being verified), the Agent won't run. All databases that the Agent accesses when run on a

server are opened with the privileges of the user whose name is in the signature (the Replicator does something similar when it synchronizes two databases). Unfortunately, the API technique used to enforce selective privileges on a database only work on databases local to the machine where the program (the Agent) is being run. Therefore, server Agents are prohibited from accessing databases on machines other than the one on which they are running. This restriction may be resolved in Domino Release 5.0.

Agent Administration

Database access is, however, only one aspect of Agent security. How do we ensure that Agents don't bring the server down by making it run out of disk space, or by consuming lots of system resources? To prevent this sort of thing, Domino gives the server administrator all kinds of control over how and when Agents execute.

Figure 8.7 shows a screen shot of the Agent Manager section of a typical server record in a public address book.

Figure 8.7 *Agent Manager parameters.*

These are the basic "throttles" that a system administrator can use to control how and when Agents execute on a server.

- **Who can run personal Agents?** If this field is empty, no one can run personal (private) Agents. The ability to *create* personal (or shared) Agents is controlled per database in the access control list, but this field controls whose Agents will actually execute on a server. Shared agents are not affected by this field.
- **Who can run "restricted" Agents?** Restricted Agents have fewer privileges on a system than do unrestricted Agents (see below in Figure 8.8 for a list of all the things restricted Agents can't do). The restrictedness of an Agent does not belong to the Agent itself, however, but to the signature on the Agent. At run time the Agent Manager checks an Agent's signature against the contents of this field and decides whether the Agent is restricted.

- **Who can run "unrestricted" Agents?** If an Agent's user (or group containing the user) name is not in either the Restricted or Unrestricted field, then the Agent will not run. Putting a name in both fields is redundant—you can't be both restricted and unrestricted at the same time. If a name is in both fields, it is considered unrestricted.
- **Refresh Agent cache.** Specifies the time of day (or night) at which the server record is re-read by the Agent Manager, and everything in memory Agent data is flushed and recomputed. This is done so that changes to scheduling parameters and access are revalidated once a day. Changes to individual Agents are picked up right away by the Agent Manager, because it is monitoring changes to Agent records in all databases.
- **Start time.** Agent Manager execution parameters are divided into two groups: daytime and nighttime, allowing for different server loadings depending on whether lots of users are expected to be logged into a server. This field specifies at what time to transition from nighttime to daytime, or vice versa.
- **End time.** Together with the Start Time field, defines the span during which the server uses daytime or nighttime parameters.
- **Maximum concurrent Agents.** The maximum number of Agents that will be allowed to run concurrently. The default is one during the day and two at night, but you can put in whatever you think appropriate. The Agent Manager consists of one overall "supervisor" process, plus one "executive" process for each possible concurrent Agent. So if you want five Agents to be able to execute concurrently on a server, then you'll have a total of six Agent Manager processes in memory.
- **Maximum execution time.** The Agent Manager will "time out" Agents that exceed this execution limit. The limit is in clock time, not CPU time. Agents that time out are made to cease execution, and an entry is made in the server log to that effect. There are some special considerations having to do with time outs and multithreaded Java Agents (see below).
- **Maximum Percentage Busy.** As Agents execute on a server, the Agent Manager keeps track of how much of the CPU is being consumed by Agents. If the percentage exceeds the value entered in the server record, the Agent Manager will delay executing Agents until the number falls to acceptable levels. It was discovered late in the release cycle of Domino 4.6 (too late to do anything about it) that the calculation used to compute the

percentage of CPU taken up by Agents did not properly account for multi-CPU computers. The error was fixed for Release 4.61, but for 4.6 you should adjust this value appropriately for SMP machines. If you want the maximum percentage to be 25% and you have a two-processor machine, set the value to 50%.

Table 8.1 Restricted Agent Operations

	Unrestricted	Restricted	
Disk i/o	Yes	No	
Network i/o	Yes	No	
Embed OLE objects	Yes	No	
Attach files	Yes	No	
Detach files	Yes	No	
Modify environment variables		Yes	No
Read system variables	Yes	No	
Access system clock	Yes	No	
Modify Thread Group	No	No	
Install a Class Loader	Yes	No	
Create Subprocess	Yes	No	
Call System.exit()	No	No	
Load DLL	Yes	No	
Define lotus.notes classes	No	No	

Agent Latency, Or, Why Won't My Agent Run?

A very important design consideration for the Agent Manager was that it should not consume the resources of every server on which it runs. You've seen one outcome of that above in the kinds of controls that a server administrator can exercise over it. Another issue concerns the actual throughput and performance of the Agent Manager implementation itself. In order not to cycle endlessly and suck up all the CPU cycles on the machine, the Agent Manager has several built-in delays, many of which are controllable via "environment variables," values that are maintained in the machine's notes.ini file. For example, when you set an Agent to be triggered by new mail arriving

in a database, Agent Manager doesn't run the Agent every time new mail arrives. It waits a small interval of time, in case a few new messages arrive close together, then starts the Agent and gives it the entire list, which is much more efficient.

Here are some of the environment variables you can control to tune Agent Manager behavior. In order to add an environment variable to your NOTES.INI file, just use a text editor. (Figure 8.8 shows a screen shot of a typical NOTES.INI file.)

- **AgentManagerSchedulingInterval.** Valid values are 1 minute to 60 minutes. Default is 1 minute. Specifies the frequency with which the Agent Manager runs a scheduling cycle, meaning, how often it searches its internal queues looking for something that's ready to run.
- **AMGR_UntriggeredMailInterval**. Valid values are 1 minute to 1440 minutes. Default value is 60 minutes. Sometimes Agents that should be triggered by new mail arriving just aren't. This parameter tells the Agent Manager how often to check for that situation.
- **AMGR_DocUpdateAgentMinInterval**. Valid values are 0 to 1440 minutes. Default value is 30 minutes. Specifies how long to wait between execution of the same Document Update Agent. When a document update event is posted to the Agent Manager it checks any update Agents to see when they last ran. If the interval is less than this parameter, the Agent(s) are not run.
- **AMGR_NewMailAgentMinInterval**. Valid values are 0 to 1440 minutes. Default value is 0. Specifies the minimum interval to wait between executions of a new mail Agent (as distinct from a document update Agent).
- **AMGR_DocUpdateEventDelay**. Valid values are 0 to 1440 minutes. Default value is 5 minutes. Specifies the amount of time the Agent Manager will wait between receiving a document update event and starting execution of any update Agents. This is done so that a rapid series of changes can be batched up and processed by the Agent all at one time.
- **AMGR_NewMailEventDelay**. Valid values are 0 to 1440 minutes. Default value is 1 minute. Specifies the amount of time the Agent Manager waits between receiving a new mail notification and starting execution of any new mail Agents.

- **DEBUG_AMGR.** This parameter, if set, turns on debug reporting to the server console (and therefore to the server log) by the Agent Manager. Several settings are possible here, depending on what kind of information you want. The valid values for this variable are: "*" for all options; "r" for Agent execution events; "s" for Agent scheduling events; "l" for Agent loading events; "m" for Agent memory warnings; "p" for Agent performance statistics; "c" for Agent control parameters; "v" for all other options, such as reporting of database searching and updates to internal queues. You can specify as many individual debug codes as you want. For example, "DEBUG_AMGR=pcr".
- **DominoAsynchronizeAgents.** The Domino HTTP server typically wants to be multithreaded, so that it can handle multiple incoming HTML requests simultaneously. But because these incoming requests can sometimes involve the execution of Agents on the server, a parameter is provided to control whether HTTP Agent execution should also be multithreaded. If this parameter is set to "1", then Agents invoked this way will run concurrently. If it is set to "0", Agent execution will be serialized when invoked from HTTP (only one Agent can execute at a time). You might want to serialize Agents on a server if the Agents are calling out to some back-end API that might not be thread safe. Early versions of LS:DO (the LotusScript classes for ODBC) had problems with this, although these problems have been largely resolved in Domino 4.6. The default value is 0.

Figure 8.8 *NOTES.INI file.*

In addition to the delays enforced above, Agent Manager has a built-in delay in its main processing loop, the infinite (until it's shut down) outer loop that checks for tasks to perform. It runs a few iterations, then sleeps for about a minute. Again, this is to make sure the process is not compute bound.

With all this, there's probably an absolute minimum delay for event-triggered Agents of between 1 and 2 minutes. This is okay for most applications, but it doesn't work well for new mail Agents in applications where it is a hard and fast requirement that the Agent get first crack at incoming mail before a user could possibly see it.

Because new mail Agents never execute synchronously on delivery of messages (it would slow down the Router too much), there is always a race condition, a window of 1 or 2 minutes between when the mail arrives and when the Agent will process it, during which a user could get in there first using the Notes Client. Look for this problem to be addressed in Domino 5.0.

Other Interesting Environment Variables

There are some additional environment variables in NOTES.INI that affect Java execution.

- **JavaUserClasses.** Normally when Domino executes a Java Agent (foreground or background), it uses its own internal CLASSPATH setting to locate predefined Java classes (for example, those belonging to the Java run-time system, and those in notes.jar). Sometimes it is really necessary for you to modify the CLASSPATH Domino uses to add additional directories. If this variable is set, it is prepended to the internal CLASSPATH value.
- **JavaMaxHeapSize.** The Java Virtual Machine maintains its own memory heap, and sometimes it runs out of memory (especially if the garbage collector doesn't get to run very often). The default maximum heap size is 64MB, but you can make it larger (or smaller, I suppose) be setting this variable. The value of the variable is in bytes.
- **JavaStackSize**. The default VM stack size is 400KB—you can make it larger with this variable. The value is in bytes.

Multithreaded Agents

There are a couple of references above to the fact that multithreaded Agents require something special in order to function properly. The reason for this is yet another Agent security consideration, something new to the Java environment that was never a problem with LotusScript, since LotusScript is not a multithreaded language.

Consider a background server Agent written in Java that spawns multiple threads. What happens if the main Agent terminates (or times out), but the child threads do not? What if there are 300 child threads, each of which purposely goes into an infinite loop, and the Agent runs every half hour? Well, the answer is that this would be a bad thing—the server would eventually be brought to its knees. This is often called a "denial of service attack," because the malicious (or just stupid, which sometimes amounts to the same thing) Agent is consuming system resources, and thus preventing others from utilizing them.

For this reason, Domino terminates all child threads belonging to an Agent when the Agent's *main thread* (the one explicitly launched by Domino on which your NotesMain() method is called) terminates. The server keeps track of all threads spawned by a given Agent by keeping them all in the same ThreadGroup (a standard Java class for managing threads). When the Agent's NotesMain() call ends (and we always know when that happens, because it's invoked from NotesThread, remember), the ThreadGroup is terminated as well.

This means that you want to be very careful in how you code a multithreaded Agent: You want the NotesMain() thread to hang around long enough for all child threads to complete, yet you want this to happen in a "system-friendly" manner, where you don't use up too many resources just checking child thread status. The technique used in Chapter 7, Listing 7.4 (Ex74Multi.java) where the main line thread does a sequential series of join() calls, one on each child thread, works okay, but isn't as elegant or efficient as we might like. In Listing 8.3, we recode it to do a better job. At the same time, we'll change it to be an Agent instead of an Application.

Listing 8.3 Waiting For Child Threads (Ex83Multi.java)

```
import lotus.notes.*;
import java.util.*;
public class Ex83Multi extends AgentBase
```

```
{
    public static final int n_threads = 5;

    public void NotesMain()
        {
        int i;
        Object waitSem = new Object();  // used for
wait/notify

        try {
            Session s = getSession();
            AgentContext ctx = s.getAgentContext();
            Database db = ctx.getCurrentDatabase();
            Log alog = s.createLog("Ex83");
            alog.openAgentLog();
            Ex83Child array[] = new Ex83Child[n_threads];
            for (i = 0; i < n_threads; i++)
                {
                array[i] = new Ex83Child(i, db, alog,
waitSem);
                array[i].start();
                }

            alog.logAction("All threads started");

// now we wait
            boolean foundone = true;
            while (foundone)
                {
                // wait to be pinged by a child
                try {
                    synchronized(waitSem) { waitSem.wait();
}
                    }
                catch (InterruptedException iex) {}

                foundone = false;

                // see who it was that notified us
```

```
                for (i = 0; i < n_threads; i++)
                    {
                    if (array[i] != null &&
array[i].isDone())
                        {
                        System.out.println("Child " + i + "
reporting done.");
                        array[i] = null;
                        }
                    else if (array[i] != null)
                       {
                       foundone = true;
                       alog.logAction("One or more threads
still active");
                       }
                }  // end for
            }  // end while
            alog.logAction("All threads done, exiting");
            }
        catch (Exception e) { e.printStackTrace(); }
        }  // end NotesMain
} // end Ex83Multi
/*** Now the child  class that does the work ***/
class Ex83Child extends NotesThread
{
    private int index;
```

```
    private Database db;
    private Log alog;
    private boolean done;
    private Object waitSem;

    public Ex83Child(int i, Database d, Log l, Object sem)
        {
        this.index = i;
        this.db = d;
        this.alog = l;
        this.done = false;
        this.waitSem = sem;
        }

    public synchronized boolean isDone()
        {
        return this.done;
        }

    public void runNotes()
        {
        try {
            this.sleep(500);
            this.alog.logAction("Starting thread " + index);
            Document doc = db.createDocument();
            doc.appendItemValue("index", new
Integer(index));
            DateTime dt =
db.getParent().createDateTime("today");
            dt.setNow();
            doc.appendItemValue("creationtime", dt);
            doc.save(true, false);
            this.alog.logAction("Thread " + index + "
exiting");
            }
        catch (Exception e) { e.printStackTrace(); }
        finally {
```

```
                    synchronized (this) { this.done = true; }
                // notify the parent thread

                synchronized (waitSem) { waitSem.notify(); }

                }


            }
} // end class
```

The following code shows the contents of Ex83Multi Agent Log after executing:

```
Started running agent 'Ex83 Multi-threaded' on 10/20/97
11:08:44 AM
Running on new or modified documents: 5 total
Found 5 document(s) that match search criteria
10/20/97 11:08:44 AM: All threads started
10/20/97 11:08:45 AM: One or more threads still active
10/20/97 11:08:45 AM: One or more threads still active
10/20/97 11:08:45 AM: One or more threads still active
10/20/97 11:08:45 AM: One or more threads still active
10/20/97 11:08:45 AM: One or more threads still active
10/20/97 11:08:45 AM: Starting thread 0
10/20/97 11:08:45 AM: Starting thread 1
10/20/97 11:08:45 AM: Starting thread 2
10/20/97 11:08:45 AM: Starting thread 3
10/20/97 11:08:45 AM: Starting thread 4
10/20/97 11:08:45 AM: Thread 1 exiting
10/20/97 11:08:45 AM: Thread 3 exiting
10/20/97 11:08:45 AM: Thread 2 exiting
10/20/97 11:08:45 AM: Thread 0 exiting
10/20/97 11:08:45 AM: Thread 4 exiting
10/20/97 11:08:45 AM: All threads done, exiting
Ran Java Agent Class
Done running agent 'Ex83 Multi-threaded' on 10/20/97
11:08:46 AM
```

The Ex83Multi Agent differs from the original Ex74Multi Application in several interesting ways:

- Since it's an Agent, we can use AgentContext and Agent log features to simplify the coding a bit, and to keep the Agent's debug output attached to the Agent itself.
- Ex83Multi uses a separate class to do the per-thread work Ex83Child inherits from NotesThread, whereas Ex83Multi inherits from AgentBase. We probably could have left it all in one class, but spinning off multiple AgentBase instances is heavier weight than we need.
- The loop that checks all the child threads for termination actually queries each child class instance to see if the "done" flag has been set internally. Note that the isDone() call is synchronized, because there's a place in Ex83Child's logic where we set the value of the flag. That logic too is semaphored, using a synchronized block. This avoids a race condition where a caller is querying the isDone() call on one thread at the same time that the instance is setting the value of its local variable on another thread.
- The loop that searches for a "done" child thread is not a continuous loop, because to just continuously execute the *for* loop over and over would waste compute cycles. Instead we use the wait/notify feature of Java to good advantage. As with *synchronized*, the wait() and notify() calls need an object instance to operate on, so we create a variable of class Object to serve that purpose. Its only function is to be the hook for the wait/notify sequencing. The parent thread calls wait() on it at the beginning of each *while* loop iteration. That effectively puts the parent thread to sleep until a notify() happens on that same object. The waitSem object is passed to each child thread in its constructor, and when a child is done, it calls notify() on the semaphore object. That causes an InterruptedException to be thrown, which the parent catches. The child threads are in effect "pinging" the parent, waking it up each time. Because the semaphore object is shared, access to it must also be synchronized. The wait() and notify() calls are not declared *synchronized*, so we have to do that explicitly.
- Note that all the threads in this Agent share not only the Database instance (as in the previous example), but the Log instance as well. Several

of the threads can be calling logAction() at the same time, and (because the call is synchronized) they don't step on each other.

Summary of Agent Output Options

As you've seen in the preceding examples, Agents have a few different ways of sending output to the outside world, for logging or debugging purposes. Table 8.2 summarizes them.

Table 8.2 Agent Output Options

	Foreground Agent	Background Agent
System.out	Java console	Server console, server log
AgentBase.getAgentOutput()	(nowhere)	Browser (HTTP agent only)
Log.openAgentLog()	Agent log	Agent log

Special Functionality for Web Agents

We've already mentioned that you can invoke a LotusScript or Java Agent from a Web browser via the HTTP server. We've also mentioned that when you submit a form from a Web browser to an Agent, the AgentContext.getDocumentContext() call will return a Document instance that contains all the fields from the form.

What wasn't mentioned before is that the AgentContext.DocumentContext property also provides you with a number of metadata about the way the Agent was invoked. If you're a CGI programmer, then you're already familiar with the concept of *CGI Variables*. If not, read on.

There are a number of parameters that the Domino HTTP server automatically sets into the context Document before invoking the Agent. They are all listed and described in the Domino online help database, so I'll just summarize them here (all values are of type String):

- **Auth_Type.** Authentication method used to validate the user. Often missing.

- **Content_Length**. Length of the content as provided by the client. Often missing.
- **Content_Type**. When there's attached information (e.g., for POST and PUT), this is the data content type (e.g., "text/html").
- **Gateway_Interface**. CGI version number for the server. Often missing.
- **HTTP_Accept**. The MIME types that the client accepts.
- **HTTP_Accept_Language**. The two letter language code (e.g., "en" for English).
- **HTTP_Referer**. The URL of the page from which the user arrived. Sometimes missing.
- **HTTPS**. "ON" if SSL mode is enabled, otherwise "OFF".
- **HTTP_User_Agent**. The name of the user's browser. The values here are not as straightforward as you might expect. When I dump the value for Internet Explorer version 3.02, I get this string: "Mozilla/2.0 (compatible; MSIE 3.02; Windows NT)". Go figure.
- **Path_Info**. The part of the URL following the database name with which the Agent was invoked.
- **Path_Translated**. The contents of Path_Info with any virtual-to-physical mapping translated. All the ones I've ever seen when invoking Domino Agents were *null*.
- **Query_String**. The part of the URL following the "?" (if any).
- **Remote_Addr**. The IP address of the client machine making the request.
- **Remote_Host**. The name of the client machine making the request.
- **Remote_Ident**. The remote user's name as known to the server. Often *null*.
- **Remote_User**. Authentication method used to get a validated user name. Often *null*.
- **Request_Method**. The HTTP method used to invoke the request ("GET", "HEAD", "POST", etc.).
- **Script_Name**. Virtual path to the current script, used in self-referencing URLs. Often missing.
- **Server_Name**. The server's host or DNS name, or IP address.
- **Server_Protocol**. The name and revision of the protocol used to make the request. For example, "HTTP/1.0".
- **Server_Port**. The port to which the request was sent. Usually 80 for HTTP requests.

- **Server_Software**. Name and version of the server software ("Lotus-Domino/4.6").
- **Server_URL_Gateway_Interface**. The CGI version spec with which the server complies. Often missing.

There may also be other Items in the ContextDocument whose names begin with HTTP_ or HTTPS_. Some that I've seen are HTTP_CONNECTION, HTTPS_KEYSIZE, HTTP_HOST, HTTP_UA_CPU, HTTP_UA_OS, HTTP_UA_COLOR, and HTTP_UA_PIXELS.

Let's do a quickie Agent (Listing 8.4) that dumps the contents of the context Document. This Agent is on the CD in source form and is also contained in Ex84.nsf.

Listing 8.4 CGI Variable Dumper (Ex84CGI.java)

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

public class Ex84CGI extends AgentBase
{

    public void NotesMain()
       {
     try {
     Session s = this.getSession();
     AgentContext ctx = s.getAgentContext();
     Document doc = ctx.getDocumentContext();
     PrintWriter pw = this.getAgentOutput();

     if (doc == null)
     pw.println("No context document");
     else
          {
          java.util.Vector v = doc.getItems();
     pw.println("Found vector with " + v.size() +
                                       " elements<BR>");
```

```
        int i;
        for (i = 0; i < v.size(); i++)
        {
        Item item = (Item)v.elementAt(i);
        pw.print("Item " + i + ": ");
        if (item == null)
             pw.print("NULL");
        else pw.print(item.getName() + " / " +

    item.getText());
        pw.println("<BR>");
        }
             }
        }

        catch (Exception e) { e.printStackTrace(); }
        }
}
```

We'll use this later in Chapter 11 as the foundation for an Agent/Servlet adapter. Figure 8.9 shows the output of the Agent in a browser window.

Figure 8.9 *Output of Ex84CGI Agent.*

Figure 8.10 Shows the output of the same URL with some query parameters added.

Figure 8.10 *Ex84CGI Agent output with query.*

Summary

We've seen in this chapter how to write single- and multithreaded Agents for NOI. We've also seen how Agents can be triggered from the Domino HTTP server, and how to pass CGI variables through to an Agent. In the next chapter, we consider some of the problems with debugging Domino Java Agents.