

Chapter 7

Writing NOI Applications

If you've been checking out the examples in all the previous chapters, you've already become familiar with how to write a Java program that runs from your command line and manipulates Notes objects. We'll review the basic techniques again in this chapter, then go on to show how to write interesting multithreaded Applications. This chapter sticks to topics having to do with writing Applications, while Chapter 8 covers the technology specific to Agents.

Single-Threaded Applications Using NOI

A single-threaded application is easy – all the examples up until now have been such.

The only thing you need to remember is to initialize and terminate the current thread in your class's *main* function using the static NotesThread calls `sinitThread()` and `sternThread()` (the latter should be invoked from a *finally* block for maximum safety).

Listing 7.1 is a skeleton NOI application that does nothing but start up, print a message, and shut down:

Listing 7.1 Skeleton Single-Threaded Application

```
import java.lang.*;
import java.util.*;
import lotus.notes.*;
public class Skeleton1
{
    public static void main(String argv•)
    {
        try {
            NotesThread.sinitThread();
```

```
        System.out.println("Hello World!");
    }
    catch (Exception e) { e.printStackTrace(); }
    finally { NotesThread.stermThread(); }
} // end main
} // end class
```

Naturally in this example the NotesThread init/term calls are completely unnecessary since we haven't created any Notes objects. Still, this very simple example has a few points worth emphasizing, even if you're familiar with Java, so that the topics we cover later are more comprehensible:

- This is a true application, as distinct from Agent, Applet, or Servlet. Its *main* function is invoked directly from the Java interpreter, and you start it by typing *java Skeleton1* on the command line.
- The import statement for *java.lang.** is redundant, because Java always gives you those classes anyway. The import statement for *java.util.** and *lotus.notes.** are most definitely *not* redundant; you need them in order to use any of the *java.util* classes (most notably *Vector*, but there are others that are pretty useful) or any of the Notes classes.
- The *main* function has to be *static*, meaning that no instance of the *Skeleton1* class exists when Java runs the program. We'll see what effect this has on us later.
- The input argument to *main* is reminiscent of C, except that there's just an *argv*, no *argc*. You don't need the argument count as a separate parameter, as you do in C, because all arrays in Java have a built-in *length* field you can use to query the size of the array. *Argv* will contain any input arguments from the command line, always passed in as Strings.
- The Notes initialization and termination that we're doing are only valid for the current thread, the one on which *main* is invoked.
- The *System.out* stream is predefined and set up by Java. We can write to it and not really care where it goes. In the case of a command line program such as this, it goes to the window that launched the program. Likewise, we can read from the "console" window using *System.in*.

- When the *main* function ends, the current (and only) thread terminates, and you're back in the console window staring at a "Hello World!" message.

In a real application you'd replace the `System.out.println` call with all of your program's logic, probably complete with manipulation of other classes, including possibly some NOI classes. The Skeleton framework, however, is still good enough to get you going in all cases, so long as you only ever need one thread.

Skeleton Multithreaded Program

But what if you want to (or like to) use more than one thread to execute your Application's logic? There are lots of reasons why you'd want to do that: You can achieve greater parallelism by executing two or more tasks simultaneously, especially when one or more of them is subject to long delays waiting for network or disk i/o. Any program that accesses the Internet, for example, is subject to delays while it waits for a response from some distant and overloaded server. Even if your Application is just doing a lot of disk i/o, you'd be surprised how much of the total run time your program spends waiting for the disk to spin to the proper location so that the magnetic head can catch a few bytes of data at a time.

Multithreaded programming is one of the really strong points of Java; the language has lots of built in constructs and utilities that make it pretty simple to deal with. Let's do another skeleton, this time with the basics for a multithreaded NOI Application.

Listing 7.2 Skeleton Multithreaded Example Using NotesThread

```
import java.util.*;
import lotus.notes.*;
public class Skeleton2 extends NotesThread
{
    public static void main(String argv•)
    {
```

```

        try {
            Skeleton2 s2 = new Skeleton2();
            s2.start();
            s2.join();
        }
        catch (Exception e) { e.printStackTrace(); }
    } // end main

    public void runNotes()
    {
        System.out.println("Postcard from Another Thread");
    }
} // end class

```

The most interesting part of this version of Skeleton is that our *main* program creates an instance of the class that it is declared in, then calls the *start* method on it. The Skeleton2 class extends (inherits from) `lotus.notes.NotesThread`, not `java.lang.Thread`. The reason for this is simple: Notes is not completely (or even mostly) implemented in Java, although it is a product that runs portably on many different operating systems. It requires some setup and shutdown *per thread*, not just per process. The best way to both ensure that this happens and that it isn't too inconvenient for you, the NOI programmer, was to invent the NotesThread class. It handles the per-thread initialization and teardown of Notes for you, and in all other respects is exactly like `java.lang.Thread`, which it extends.

One difference, though, is in the way that you use NotesThread, versus the normal way of using Thread. When you have a class that extends Thread, you implement your logic in a method named *run()*. When you use NotesThread you implement your logic in a method named *runNotes()* instead. In fact, if you tried to write something like Skeleton2 and implement a *run()* method, you'd get a compile time error, because NotesThread implements *run()* as a *final* method; you can't override it.

Why have this difference? The reason is simple: NotesThread needs to be sure that it gets invoked *before* any code in your class can possibly run on the thread, so it can do you the favor of initializing Notes and your NOI objects will work as advertised. NotesThread also needs to be sure that it does the right per-thread shutdown of Notes when your program (or at least this thread in your program) is done. Not to do the setup would mean that the first time you tried to use an NOI object, you'd probably crash. Not to do the teardown would mean that when your last NotesThread instance was done, your program would hang. That wouldn't be too cool, so here's the order in which things happen when you run Skeleton2:

1. You type "java Skeleton2" on the command line. The Java Virtual Machine (VM) is started in your current process space.
2. Java creates a Thread instance, loads and verifies Skeleton2, and calls the static main() function. No instance of Skeleton2 has been created yet. We'll call this "thread1".
3. The main() function does a *new* on Skeleton2, so Java creates an instance of it. Skeleton2 inherits from java.lang.Thread. Note that we didn't provide a constructor for Skeleton2. Doesn't matter, Java will use a default one (no arguments) for us.
4. main() calls start() on s2. It then (while step 5 proceeds on another thread) immediately calls s2.join(), which just waits for s2's new thread to complete.
5. Java creates a new execution thread, let's call it "thread2", transfers control to it and, using that new thread, calls the run() method, which was declared *final* in NotesThread. We've started executing on thread2 now.
6. The NotesThread run() method makes sure that the required Notes libraries are loaded, and calls an exported C routine to initialize Notes for the current thread (thread2).
7. If step 6 goes well, NotesThread.run() next invokes the runNotes() method.
8. Because Skeleton2 has a runNotes() method and is the last extended class in the inheritance chain, its method gets invoked, still on *thread2*. If we had omitted a runNotes() method from Skeleton2, the one in NotesThread (the

next class up the inheritance chain) would get called (all Java methods are *virtual* in the C++ sense). In that case there's no problem, since NotesThread has a default runNotes() implementation, just in case. It doesn't do much, just returns.

9. Skeleton2's runNotes() method is called. It sends a string to the system output stream (Java's version of *stdout*), then returns.
10. We're back in NotesThread.run(). Even if Skeleton2's runNotes() (or something it called) had thrown an Exception, we'd still get back to NotesThread.run() because the call to runNotes() is in a *try* block, and following that is a *catch* that does nothing but print a stack trace (for your debugging convenience). Following the *catch* block is a *finally* block, which Java guarantees to execute after the *try* is done, regardless whether it exits normally or because of an exception. The finally block simply calls Notes again to shut down the current thread, then it exits.
11. NotesThread.run() returns to java.lang.Thread.start(), which kills the system thread.
12. *Thread2* is gone now, so back on *thread1* our main() function returns from its call to s2.join(), then exits.
13. Java shuts down *thread1*, and having nothing else to do, exits. The Java process terminates, and control returns to your console window.

Note also that we didn't need to use the static NotesThread.sinitThread() and stermThread() calls in this version of main(). It's still a static method run on a thread created by the Java VM, but since we didn't create or use any Notes objects in main(), no initialization of Notes was required.

So there you have it, a basic multithreaded Java application all set up for NOI. Before going on to show you how to do really interesting things with multithreaded NOI applications, let's take a quick look at another way to set up a multithreaded skeleton.

Listing 7.3 Skeleton Multithreaded Example Using Runnable

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

```

public class Skeleton3 implements Runnable
{
    public static void main(String argv•)
    {
        try {
            Skeleton3 s3 = new Skeleton3();
            NotesThread thread1 = new NotesThread(s3);
            thread1.start();
            thread1.join();
        }
        catch (Exception e) { e.printStackTrace(); }
    }

    public void run()
    {
        System.out.println("Hello from Skeleton3");
    }
} // end class

```

Skeleton3 illustrates the other common way to launch threads. Sometimes you just can't have your class extend NotesThread. Maybe it already needs to extend some other class, and Java only allows single inheritance (we could have had Skeleton3 extend some other class in this example). Maybe your application is such that you want to manage NotesThread instances instead of instances of other classes. In any case, you can set it up this way just as easily and effectively as the way we did in Skeleton2.

The second standard option for multithreading (the one that we use in Skeleton3) is to create an instance of NotesThread directly, and pass to its constructor an instance of your class. Your class can inherit from anything, but it must implement the Runnable interface. *Implements* simply means that your class has in it all the methods that are defined in the interface. The Runnable interface has only a single method: run(), so (unlike with Skeleton2, where we were prohibited from implementing a run() method)

Skeleton3 must implement a `run()` method. This is the method that `NotesThread` will invoke for you.

Here are the steps that take place when the `Skeleton3` program runs.

1. Java invokes `Skeleton3`'s `main()` function.
2. `main()` instances a `Skeleton3`.
3. `main()` creates an instance of `NotesThread`, passing the `Skeleton3` instance in as an argument. `NotesThread` saves `s3` away for later.
4. `main()` calls `NotesThread.start()`.
5. Java creates a new thread and invokes `NotesThread`'s `run()` method.
6. `NotesThread.run()` does its `Notes` setup *shtick*, then invokes the `run()` method on the reference to `s3` that it saved away in its constructor.
7. `Skeleton3.run()` is invoked, prints out a message, and returns.
8. `NotesThread` has the same logic following its call to `Skeleton3.run()` as it did following its call to `Skeleton2.notesRun()`: a finally block where `Notes` is terminated for the current thread.
9. `NotesThread` returns to `Thread.start()`, which kills the thread and returns to `main()`.
10. `main()` has been waiting on a `join()` call in its original thread. It now continues, and exits.
11. Java shuts down its process, returning to the console window.

Thus the actual differences in the two techniques can be summarized in a small table.

Table 7.1 Two Techniques for Multithreading an Application

Extend <code>NotesThread</code>	Implement <code>Runnable</code>
<code>main()</code> instances your class only	<code>main()</code> instances your class and <code>NotesThread</code>
<code>main()</code> calls <code>YourClass.start()</code>	<code>main()</code> calls <code>NotesThread.start()</code>
<code>NotesThread</code> calls <code>YourClass.runNotes()</code>	<code>NotesThread</code> calls <code>YourClass.run()</code>

Multithreaded Applications Using NOI

The same techniques that we applied above to generate one additional thread can be used to generate any number of them. Any thread that is an instance of `NotesThread`

can be used to manipulate NOI objects. Let's do an example where we share a Database instance across a few threads. This one, and the accompanying database, are on the CD.

Listing 7.4 Example Multithreaded Application (Ex74Multi.java)

```
import lotus.notes.*;
import java.util.*;
public class Ex74Multi extends NotesThread
{
    private int index;
    private Database db;
    public static final int n_threads = 5;

    public static void main(String argv•)
    {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"book\\Ex74.nsf");
            Ex74Multi array• = new Ex74Multi[n_threads];
            for (int i = 0; i < n_threads; i++)
            {
                array[i] = new Ex74Multi(i, db);
                array[i].start();
            }

            System.out.println("All threads started");

            // now we wait
            for (int i = 0; i < n_threads; i++)
            {
                array[i].join();
                System.out.println("Thread " + i + " is
done");
            }

            System.out.println("All threads done, exiting");
        }
    }
}
```

```

        }
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
    } // end main

    public Ex74Multi(int i, Database d)
    {
        this.index = i;
        this.db = d;
    }

    public void runNotes()
    {
        try {
            this.sleep(500);
            System.out.println("Starting thread " + index);
            Document doc = db.createDocument();
            doc.appendItemValue("index", new
Integer(index));
            DateTime dt =
db.getParent().createDateTime("today");
            dt.setNow();
            doc.appendItemValue("creation time", dt);
            doc.save(true, false);
            System.out.println("Thread " + index + "
exiting");
        }
        catch (Exception e) { e.printStackTrace(); }
    }
} // end class

```

Discussion of Ex74Multi

Let's analyze this one a little and dissect a couple of interesting aspects of it. First of all, notice that the Ex74Multi class extends NotesThread, so that we can conveniently use the NOI objects. But then why does main() use the explicit static init/term calls to

NotesThread (sinitThread and stermThread)? Remember that main() is *static*, and is invoked directly from the Java interpreter; no instance of the class has yet been created, and so no NotesThread initialization code has been called for main(). We could have coded this differently by having main() do nothing but create an Ex74Multi instance, then call start() on it, putting the multithreaded logic into runNotes() instead. But then we would have most likely wanted to create a second class to do what we now do in runNotes(), and that seemed like too much work. Later we'll be doing stuff that way.

The main() function does all the setup: gets a Session, opens the Database for this example, and then creates an array of Ex74Multi instances, so we can keep track of them. As we create an instance, we pass in two arguments: the instances index (place in the array) and the Database instance. The class constructor does nothing but cache the arguments in some member variables. Then, after constructing an instance, main() invokes start() on it.

The start() method in java.lang.Thread creates a new thread and (via NotesThread) invokes our runNotes() method. When main() is done with object creation, it prints a message and enters a second loop, wherein it waits for each thread in turn to complete. When each thread is done, it prints another message. Note that calling join() in sequence to make sure all the threads are done is okay, but not the greatest technique in the world. It's safe, because calling join() on a thread that has already terminated is a no-op. The issue is that you don't really know which thread will end first, but in this case we really don't care. Later we'll see other ways of doing this that are more appropriate to the real world.

When all threads have finished, main() cleans up and exits. Meanwhile, what about runNotes()? This is where the meat of the program really is: Each thread adds a single Document to the Database. Note that because of the way we set up our constructor, each thread is operating on the *same* Database instance, although each is creating its own Document and DateTime instances.

The first thing each thread does is sleep for half a second. Why? Well, there are a couple of reasons why you would consider explicitly slowing down each thread like this:

- All user threads (NotesThread is always a user thread, as opposed to a system thread) run at the same priority, and all user threads run at a higher priority than the garbage collection (gc) thread. If you run a big program with a lot of threads where each thread is (albeit temporarily) consuming resources and never "yielding," then the gc thread never gets a chance to run. You could run out of memory, or file handles, or some other system resource, even though you coded your program to reuse object references, and so on. Yielding allows the gc thread a chance to free up memory that is now unused. This technique isn't guaranteed to work, and different VM implementations on different platforms might respond differently. Your other alternative is to explicitly call `System.gc()`, which fires off a garbage collection cycle synchronously (don't do it too often!).
- Another reason for doing it in this example was to randomize the order of completion a bit. I wanted to show that just because the threads started in order 0 through 4, and just because they all executed exactly the same code path didn't mean that they would necessarily complete in order. Adding a `sleep()` call was one way to break up the order of execution a bit. I haven't proved one way or the other whether it really has an impact.

The next thing `runNotes()` does is print out a message using its current index value. Then it creates a Document in the Database, adds the index value and the current `DateTime` value to the Document, saves it, and prints another message. The following code, `Ex74Multi.java Output`, reproduces one of the runs of this program on my machine:

```
[d:\lotus\work\wordpro\book\  
All threads started  
Starting thread 0  
Starting thread 1  
Starting thread 2
```

```
Starting thread 3
Starting thread 4
Thread 2 exiting
Thread 1 exiting
Thread 3 exiting
Thread 0 exiting
Thread 4 exiting
Thread 0 is done
Thread 1 is done
Thread 2 is done
Thread 3 is done
Thread 4 is done
All threads done, exiting
```

Note that `main()`'s messages are all in order (no surprise, as they are printed out from *for* loops), but the threads themselves exited in an unexpected order: 2-1-3-0-4. If you look in the sample database for this program (`Ex74.nsf`, on the CD) you'll see that I've set up the view to sort by creation date, and that indeed, even though all the Documents were saved within one second, they sort in (almost) the same order as the exit messages. Note that while Thread 0 claimed to have exited ahead of Thread 4, in the database Thread 0 sorts below Thread 4. The explanation is that Notes's `DateTime` granularity is one hundredth of a second, and if two Documents' sorting values are both within the same .01 second, then the sort order is unpredictable.

Is this program thread safe? What does *thread safe* mean? Both good questions. Chapter 10 addresses the issue of thread safety in much more detail, but for now I'll just point out a few relevant facts:

1. Java has a great language construct called *synchronize*. When you synchronize a method or block of code, Java guarantees that no two threads will ever execute that method (or block of code) on the same object instance at the same time. It essentially creates a *semaphore*, or critical section lock, or whatever you're used to calling it, around that code, and invisibly stores that semaphore in the object instance. If you

declare a method as synchronized, the entire method is a critical section. The best thing about synchronize is that when you use it Java cleans it up for you: no matter how you exit a synchronized block of code (return statement in the middle, fall out the bottom, throw an exception, call something else that throws an exception), Java cleans up the semaphore for you. A big win for ease-of-coding.

2. All NOI classes are synchronized on all their methods. This means that even though two threads might be simultaneously attempting to execute the `Database.createDocument()` call on the same Database instance (as in `Ex74Multi`), only one can get there first, and the other has to wait until the first one is done. This doesn't prevent a thread from calling `createDocument()` on a different Database instance, of course.
3. Synchronizing all the methods isn't enough, however. What happens when two different threads try to save two different Document instances to the same Database at the same time? Sure, `Document.save()` is synchronized, but here we have two different Document instances — synchronization doesn't apply. This is a perfect example of a fairly common case where two independent operations (saving two different Documents) have a hidden synchronization requirement: Few file systems allow you to actually write to the disk in the same file at the same time from two different threads. It's for this reason that the code in Notes that handles updates to databases is (and has always been) internally semaphored, or synchronized.
4. All the built-in Java library calls that share resources (`System.out.println`, for example) are synchronized as well, so that we don't get two threads trying to write two messages simultaneously to `System.out`, and having the text of both intermingled on the screen. Does this slow things down? Of course. The whole point is that you trade off local (per thread) throughput for two things: correctness of operation (you don't get garbled messages) and higher global (program wide) throughput. If this trade-off doesn't work for you (either because the result isn't correct anyway or because global performance isn't better), then you should rethink whether your programming task is suitable for a multithreaded solution.

A Multithreaded Web Crawler

Let's use the techniques covered so far, plus some very nice built in Java classes, to write a simple Web crawler. The idea of a Web crawler is that, starting on a given page, you find all the links on that page, and recursively travel each link, to a preset depth. We'll do a very simple one that only goes a couple of levels deep (see Listing 7.5). We'll create a Document in a Database for each link we find, and make each link Document a response to its parent link. When we're done we'll have a Database that represents in document/response format a hyperlinked subset of the Web.

Listing 7.5 Web Crawler Example (Ex75Crawl.java)

```

import lotus.notes.*;
import java.util.*;
import java.net.*;
import java.io.*;
public class Ex75Crawl extends NotesThread           5
{
    // members
    private Database theDb;
    private Document myParent;
    private URL theUrl;                               10
    private InputStream theIstream;
    private int myDepth;
    protected static final int MaxDepth = 3;
    protected static final int MaxLinks = 5;
                                           15
    public static void main(String argv•)
    {
        // Print "usage" message if no argument passed in
        if (argv == null || argv.length == 0 || argv[0] ==
null)
        {                                           20
            System.out.println("Usage: java Ex75Crawl
<URL>");
            return;

```

```

        }

        // Construct a URL for the initial page
25
        try {
            System.out.println("D0: Opening page " +
argv[0]);
            URL url = new URL(argv[0]);
            InputStream istr = url.openStream();
                                30
            // if we're still here, set up the Database
            try {
                NotesThread.sinitThread();
                Session s = Session.newInstance();
                Database db = s.getDatabase("",
"book\\Ex75.nsf");    35
                Ex75Crawl highest = new Ex75Crawl(0, null,
db, url, istr);
                highest.start();
                highest.join();
            }
            catch (Exception e) { e.printStackTrace(); }
40
            finally { NotesThread.stermThread(); }
        }
        catch (Exception e) { e.printStackTrace(); }
    } // end main
                                45

    // constructor for this class
    public Ex75Crawl(int depth, Document parent, Database
db, URL url,
                                InputStream istr)
    {
        this.theDb = db;                                50
        this.theUrl = url;
        this.theIstream = istr;

```

```

        this.myDepth = depth + 1;
        this.myParent = parent;
    }
    // main logic for Ex75Crawl
    public void runNotes()
    {
        int linkcount = 0;
        DateTime dt;
        Document doc = null;

        // are we too deep?
        if (this.myDepth > MaxDepth)
            return;

        try {
            this.sleep(100);
            doc = this.theDb.createDocument();

            // top level doc has different form
            if (this.myDepth == 1)
                doc.appendItemValue("Form", "WebURL");
            else doc.appendItemValue("Form", "WebURLR");

            doc.appendItemValue("URL",
this.theUrl.toString());
            doc.appendItemValue("Depth", this.myDepth);
            dt = this.theDb.getParent().createDateTime("");
            dt.setNow();
            doc.appendItemValue("StartCrawl", dt);

            // make this document a response to its parent
            if (this.myParent != null)
                doc.makeResponse(this.myParent);

            // make sure the page is html

```

```

        String type =
this.theUrl.openConnection().getContentType();
        doc.appendItemValue("ContentType", type);
        if (!type.equalsIgnoreCase("text/html"))
90
            {
                doc.replaceItemValue("Comment", "Link is not
html");
                return; // goes to "finally"
            }
95
        // save the doc, else there's no UNID that
children
        // can use to make themselves responses
        doc.save(true, false);
        // parse the input stream for links, create new
object for each
        BufferedReader reader =
            new BufferedReader(new
InputStreamReader(this.theIstream));
        String line;
        while ((line = reader.readLine()) != null)
            {
                105
                doc.replaceItemValue("Comment", "Read line:
" + line);

                int index = line.indexOf("HREF=");
                if (index < 0)
                    continue; // no more links
110
                index += 6; // move past the starting dbl-
quote
                int index2 = line.indexOf("\"", index); //
find 2nd quote
                if (index2 < 0)
                    break;

```

```

        String newUrl = line.substring(index,
index2);          115

        // validate this url, have we seen it
before?

        Item links = doc.getFirstItem("Links");
        if (links != null &&
links.containsValue(newUrl))
            continue;          120
        System.out.println("D" + this.myDepth + ":
Opening page"
            + newUrl);

        // Append this link info to our doc, then
crawl the link

        URL u = new URL(this.theUrl, newUrl);
125

        if (linkcount == 0)
        {
            linkcount = 1;
            doc.appendItemValue("LinkCount", 1);
            doc.appendItemValue("Links",
u.toString());          130
        }
        else {
            if (++linkcount > MaxLinks)
            {
                doc.replaceItemValue("Comment",
135
                    "Max link
count exceeded");

                return;
            }
            doc.replaceItemValue("LinkCount",
new
Integer(linkcount));          140
            Item item = doc.getFirstItem("Links");

```

```

        item.appendToTextList(u.toString());
    }

    InputStream is = u.openStream();
145
    Ex75Crawl inst = new Ex75Crawl(++myDepth,
doc, this.theDb,
                                u, is);
    inst.start(); // but don't wait for it
    } // end while
    } // end try                                150
catch (Exception e) { e.printStackTrace(); }
finally {
    // add stop time, save again
    try {
        if (doc != null)                                155
            {
                dt =
this.theDb.getParent().createDateTime("");
                dt.setNow();
                doc.appendItemValue("StopCrawl",
dt);
                doc.save(true, false);
160
            }
        }
    catch (Exception e) { e.printStackTrace(); }
    } // end finally
} // end runNotes                                165
} // end class

```

Discussion of the WebCrawler

This program actually does quite a lot in its 170 or so lines. I've numbered the listing for convenient reference, and we'll go through it in some (but I hope not excruciating) detail.

First, we have to import more than the usual number of classes, because we're using some i/o stream classes and some network classes that we haven't so far seen here. Next note that the Ex75Crawl class extends NotesThread, but that main() still does a static NotesThread init/term sequence, because we want to instantiate (and open, which is an expensive step) our Database only one time, and then share it across all threads.

There are two constants that we define at compile time (lines 13 and 14): one for the maximum recursion depth (this isn't really a recursive program, though the term is still a good one for this application), and one for the maximum number of links per page that we'll retrieve. Students of the unintended side effects of combinatorial explosion will understand why I set these limits so low.

The program expects a URL on the command line to be the starting page. The full URL syntax must be provided (although we could have coded around that easily enough), such as `http://www.lotus.com`. Leaving off the `http://` part will cause a malformed URL exception. At line 26 main() prints a progress message (D0 refers to the depth level of the current thread), and instantiates a URL instance. We open an input stream on it, just to make sure that it really exists and can be connected to. Some of the Web sites I tried this with (notably both `www.lotus.com` and `www.microsoft.com`) refused to let my program connect, probably because they want to shut out all nonbrowser traffic. I found that my Domino site provider let me in, however.

<note>Please don't try this using my site provider's URL because too many hits from programs like this can effectively be a denial-of-service attack, which is certainly unethical and probably illegal.

If we don't experience an exception so far, then we go ahead and create a Database instance for Ex75.nsf (it's on the CD). Then (line 36) we instantiate the highest level instance of Ex75Crawl, passing into its constructor our current level (0), the Document instance at the current level (there isn't one yet, so *null*), the Database instance, the URL to crawl, and the input stream instance from that URL. Then main() starts that instance and waits for it to finish (line 38). It probably would make no difference if main() didn't wait, even though it goes on to do a NotesThread.startThread(). Any other Ex75Crawl instances that are busily out there crawling away are, by definition, on different threads, and so have their own init/term logic. Even if main() exits before the other threads are done, Java lets them keep going. The interpreter exits when the last thread is finished.

Following the code for main() is the class constructor, at line 47. All it does, as in the previous example, is stash away the arguments in instance variables. Further along, at line 59, is the real meat of this class, the runNotes() method. All the rest of the logic is in this one method.

The first thing runNotes() does is check for its depth exceeding the maximum "recursion" level. If we didn't have this check in here, the program would run forever, almost guaranteed. Why do I say "guaranteed"? Because, lots of sites have links back to their home pages from down within a "deeper" page. If the crawler were to hit one of those sites, it would go into an infinite loop, always cycling back to the home page. Even if that weren't the case, a given thread would only stop when it hit a page with no links on it. How many of those are there? Not many.

At line 69 the program sleeps for 0.1 second, in case the Garbage Collector thread is desperate to run. At line 70 runNotes() creates a new Document instance in the current Database. Notes likes it better if top-level and response Documents don't share the same form, so we assign the "Form" Item a value based on our depth (lines 73-75). After that, we add the current URL, depth, and the time we began crawling to the Document. If a

parent Document was passed in, we make the current Document a response to it (line 84).

The next thing to do is to make sure that the current URL is actually something we can parse, an HTML page that is, instead of something else, like a GIF file. We get the "content type" from the URL, and store that in the Document as well, then check for the type being "text/html" (the specified MIME encoding for an HTML page). If the current page is not HTML, we add a comment to that effect to the Document, and bail out. Note that we don't call `save()` before the return statement at line 93. We don't have to, because the return will actually branch down to the finally block following the current *try* block, at line 152, where the `save()` will happen.

The next thing we have to worry about is that we might be starting a new "recursive" thread real soon, and that thread is going to want to make the current thread's Document a parent to the child thread's new Document. It won't be able to do that unless our current Document has a Universal ID (UNID), so we need to `save()` the current Document at this point. That automatically gives it an UNID, and the child Document (if any) will put that UNID in an Item named `$REF`, thereby making itself a response Document (that's what the `Document.makeResponse()` call does).

Now (line 100) we can go ahead and start scanning the current page for links. Links in HTML are encoded as an HTML tag that looks something like:

```
<A HREF="[some URL]"><IMG SRC="/eapps/nocgism.gif" Border=0>
</A>
```

where "[some URL]" is where the actual URL spec would be. So, we scan the page line by line, using a nice Java class called `BufferedReader`, to find a substring "HREF=". If we find one (the value returned by `String.indexOf()` is greater than or equal to 0, at line 109) we skip ahead to the first double quote, then scan the string for the closing double quote, and make a new `String` instance out of what's in between (lines 111-115).

At line 118, we try to figure out whether or not we've ever seen this same URL on this page already, because if so, then we want to skip it. That turns out to be pretty easy, because we already save every link we find on the page in a text list on our current Document instance. All we have to do is find the Item, and invoke the `containsValue()` method. If it returns *true*, we just skip to the next line in the page. If this really is a new URL (at least on the current page), then we print out a little message, including the current recursion depth (line 121), create a new URL object, bump the page's link counter, and add the new URL to our text list (lines 125–130). If the link counter for the page exceeds our limit, we bail. (I admit that I did not code this optimally: the limit test should be *after* we process a link, which would save us reading lines that we know we aren't going to care about. Why don't y'all go in and fix it up?)

If we're not above our limit, then (line 145) the program opens an input stream on the new URL, creates a new `Ex75Crawl` instance, and spawns a new thread. Note that we don't wait for the new thread at all. The next thing that happens is that we add the end-of-crawling date/time to the Document (lines 157–160) and `save()` the Document again; then `runNotes()` exits.

In order to keep the length of the example within reason, I didn't make the URL parsing as robust as it would be for a production program. For example, the code assumes only one HREF per line of HTML, with no spaces around the `"=`". Even more importantly, we've ignored the possibility of a BASE tag on the HTML page, which (if present) changes the meaning of each URL on that page.

You can see the results of my test run in the database `Ex75Crawl`, on the CD. Again, please *do not* try to repeat my experiment using the same URL that I did – pick another one – there are lots to go around.

Summary: Multithreaded NOI Applications

I hope this chapter has given you a good grasp of how you can write standalone NOI applications that do interesting and powerful things. I especially hope that you have gained some appreciation for the subtleties and power of multithreaded programming. This combination of the power of Domino/Notes and the power of Java are why I come to the conclusion over and over again that the combination of the two is enormously important, and interesting, to us developer types.

Coming up next, Chapter 8 goes into great and gory detail on how to write Domino Agents in Java, including multithreaded Agents. Chapter 9 tells you how to debug Agents, given that Notes does not (in Release 4.6 anyway) include an integrated Java development/debugging environment. Chapter 10 will consider some additional issues with sharing NOI objects across threads.