Chapter 10
Sharing Objects Across Threads

We've seen in the earlier chapters on multithreaded Agents and applications that NOI objects can be used (shared, in a sense) by more than one thread in a program. How does that work? It certainly isn't obvious that it should work, and it isn't always obvious when and why you'd want to do it. This chapter delves a bit into these issues.

Why Share Objects Across Threads?

Let's assume you've made the decision to write a multithreaded application or Agent, and that you already know that one or more of the threads you're going to create will use NOI objects. You've gone through an analysis that concluded something like, "We have multiple, independent tasks that need to be completed, and each task is subject to significant i/o or network delay. Therefore our program will benefit from a multithreaded architecture."

Are there cases where you'd want to access one or more NOI instances on more than one thread at a time? Quite possibly. Some examples might include:

1. Multiple threads creating Documents in a single Database (as in samples Ex74Multi and Ex75Crawl from Chapter 7). There's no reason to create a Database instance for each thread.
2. Multiple threads each conducting a different search on a single Database.
3. More than one thread needing access to an AgentContext instance.
4. Lots of threads making use of a single Session instance.

You can probably come up with numerous scenarios in which you'd want to have a common resource operated on by more than a single thread at a time. Databases certainly fall into this category, but so do Sessions, Documents, DateTime objects, and

others. Pretty much any time you want threads to share some kind of application context you'll be confronted with the sharing issue.

NOI and Thread Safety

Is the Java NOI thread safe? What does *thread safe* mean? We started to address this question in Chapter 7, but now it's time to look at the problem more closely. I'll propose the following criteria for whether or not a program is thread safe:

1. **Stability.** Threads can share objects by invoking methods on those objects and the program will not crash.
2. **Data reliability.** A thread storing data to or retrieving data from an object instance must be sure that the data are internally consistent. Another way of putting this is to say that a data put or get operation on an object by a single thread must be *atomic*; that is, all the data you store in the object are stored at the same time, and all the data you get from the object are retrieved at the same time. No other thread can modify the data in the object during your store or retrieve operation.

One could propose more criteria, and books are constantly being written on the topic. I claim that these are the two most important criteria for using the Java programming interface to Notes. If you'd like to pursue the general design issues of multithreaded programming in Java, one book I like is *Concurrent Programming in Java*, by Doug Lea (Addison-Wesley, 1997).

By these criteria, the Java NOI is indeed thread safe. Any of the objects (with one exception, which we'll get to) can be used simultaneously by any number of threads (as long as each thread is a NotesThread instance). All NOI methods are *synchronized*, meaning that no two methods on an NOI object can be invoked by more than one thread at a time. This in and of itself is not enough, as we'll see, but it is required; it's the only way to ensure that one thread isn't reading a piece of data while another is writing the same piece.

As I mentioned, synchronization of Java methods is necessary but not sufficient to provide thread safety in NOI. It would be if all of NOI were implemented purely in Java, but we know all too well (especially if you read Chapter 9) that most of the real functionality is implemented in C and C++ in the Notes/Domino core. Each Java object instance is really a wrapper for a corresponding C++ object instance, and lots of state data are maintained in the C++ layer. That state data need to be thread safe as well, naturally.

You might suppose that synchronizing all the Java methods might still be enough to protect the internal C++ state data. After all, if only one call at a time can get through to the C++ layer, where's the conflict? The answer is that there is still a great potential for conflict because of the side effects of calling a method on an object. Let me illustrate it this way:

1. You instantiate a Database instance, and do a lookup on a note id to get a Document.
2. You pass the Document to a second thread, to do some manipulation on it.
3. Back on the first thread you continue to look up Documents in the Database.
4. The second thread decides to delete the Document, removing it from the Database.

Removing a Document from a Database on one thread while the Database is in use on another thread poses a potential problem for the Database class. The searching operations that your first thread executes are all synchronized calls to a Database instance. But the Document.remove() method is a synchronized *Document* class call, and the *semaphore* used by Java to implement method synchronization won't prevent the remove() call from executing concurrently with a Database search call. That's because the semaphore (or "lock") is on the object instance that owns the method, and in our example here we're calling two methods on two different objects. While two Database

or two Document calls *would* be synchronized, when you make one call on each object the two calls are *not* synchronized with each other.

It matters because the remove() call on the Document must cause the Document instance that's going away to be removed from the list of Documents maintained by its parent Database. Thus a call to a Document method will cause an indirect invocation of some code in the Database class, and the list of Documents belonging to the Database must therefore also be protected by a semaphore. If it weren't, you could easily have a situation where Database.FTSearch() is locating a Document instance and returning it while Document.remove() is in the middle of deleting that Document. The Database instance's internal data structures would get out of synch and you'd have a problem somewhere down the road—maybe a crash, maybe a search result set full of already deleted Documents.

So, the point is that all the NOI objects that maintain state data that might be modified through calls to other (related) objects must be made explicitly thread safe in the C++ code, through the use of locks or semaphores.

There's an interesting distinction that should be made here, and one which I think the Java mechanism of synchronization blurs a bit: It is always the case that when writing code that is meant to be thread safe, you must worry about what data structures need to be protected by semaphores, not what code needs to be protected. This might sound subtle, and it is, but it's very important, and if you can grasp the distinction and take it to heart you'll end up writing better code.

Let's take the Database object's list of child Documents as an example. Each Database instance will have its own list, of course. Each Database instance can be invoked from multiple threads, both directly (via calls to Database methods) from Java and indirectly (via calls to methods on child Document instances, for example) from C++. Because the contents of the Document list must retain integrity, any operation that adds a Document to the list or removes a Document from the list must be atomic, that

is, it must be guaranteed sole access to the list during the lifetime of the add/remove operation. It would clearly be a big problem if a piece of code was attempting to add a new element to the end of a linked list at the same time that another piece of code was doing the same thing: Someone somewhere is bound to come up with an invalid pointer. Thus we semaphore (protect) the list. The code that wants access to the list must obtain a lock, guaranteeing sole access, before modifying the list, and it must release the lock when done. But it's always the list, not the code that is being protected.

Another wrinkle which Notes deals with very nicely internally, and Java doesn't, is the distinction between read locks and write locks. To continue our Database list example, if you only have one kind of semaphore (as Java does), then readers and writers of the list both have to try to get the exact same lock before they can access the list. But it is also true that we can achieve greater overall program concurrency if we allow multiple simultaneous readers. There's no reason to make two threads who both just want to scan the list for a matching Document wait for one another; they can both read the list at the same time without doing any damage.

One issue (I won't say problem) with Java is that there is only one kind of semaphore built into the language, and it makes no distinction between reading and writing. Of course, you can write your own semaphore class that does all the right things for distinguishing between readers and writers, but that's a lot of work (check out the section on "Readers and Writers" in Chapter 5 of *Concurrent Programming in Java*). Maybe someday they'll add something to the language to support that.

In the meantime, Notes has some great internal semaphoring capabilities, including read-write semaphores. The logic of a read-write semaphore is roughly as follows:

- Permit any number of readers to obtain a lock as long as there is no writer.
- Permit only one writer at a time to obtain a lock.
- If a writer holds the lock, no readers may obtain a lock.
- If any readers hold the lock, no writer may obtain a lock.

The advantage of using a scheme like this is, as I mentioned above, that you get greater concurrency (fewer readers of a data structure have to wait per session overall) and therefore better throughput. The overall point here is that not only are the Java NOI classes thread safe (with a couple of restrictions mentioned below), but that the internal mechanisms by which they are made thread safe are optimized for performance.

What *Isn't* Thread Safe?

There are three topics worth mentioning here: (1) restrictions on multithreaded use of the lotus.notes.DbDirectory class, (2) a warning about algorithms (as opposed to classes) that aren't thread safe, and (3) restrictions on memory management and the use of the Session class.

Multithreading Restrictions on DbDirectory

The DbDirectory class is mostly used to locate and get Database instances on a server. You can use DbDirectory to iterate through all the Databases of a particular type (NSF, NTF, replicas, etc.) on a specified machine (see Chapter 2 for a writeup of all the DbDirectory methods). Typically, you do an iterative search by first calling DbDirectory.getFirstDatabase(), then successively calling getNextDatabase() until a *null* is returned, indicating the end of the list.

Unfortunately, the Notes APIs that the DbDirectory class uses to implement an efficient search of Databases is not completely thread safe. This means that you can't, for example, call getFirstDatabase() on one thread and then call getNextDatabase() on the same DbDirectory instance from another thread. It won't crash if you do (DbDirectory keeps track of which thread getFirstDatabase() was called on), but it'll throw an exception. Unfortunately there's no way around this, but in practice you shouldn't ever be seriously inconvenienced.

There is absolutely no problem with using DbDirectory on a single thread. Likewise there is no problem in reusing a DbDirectory instance on a second thread: The only

restriction is that if you make a getFirstDatabase() call on a given thread, then all subsequent getNextDatabase() calls must also be made on that thread. You are free to call getFirst/NextDatabase() on one thread, then reuse the same instance on another thread to do a new getFirst/NextDatabase() sequence. There's no particular advantage to doing so, however. You can always just create a DbDirectory instance for each thread that needs one, and never share them.

This is the only restriction like this in all of the Java interface (so far as I know). All other objects may be freely used from as many threads as you like.

Thread Unsafe Algorithms

Regardless of how thread safe the actual NOI code and data structures are, there will always be algorithms that are inherently unsafe when used in a multithreaded environment. You should be aware of this as you code your Agents and applications.

Let's take an example from the Notes UI that many people have come across. We'll postulate two users, UserA and UserB, both remotely accessing the same document in the same database on the same server. Here's the sequence of steps:

1. UserA opens the document, reading it into her workstation's memory.
2. UserB opens the document, reading it into his workstation's memory.
3. UserA and UserB both make changes to the document while its contents are still in their respective workstations' memory.
4. UserA saves her version of the document back into the database on the server.
5. UserB attempts to save her version of the document back into the database, but gets an error saying that the document was modified since the time UserB last read it.

There's no way for Notes to prevent this sort of conflict, given the nature of the beast as a multi-user groupware product. No amount of data structure semaphoring will circumvent this problem. The only thing you (the application developer) can hope to do is detect the conflict and warn the user appropriately, or take corrective action.

The same is true for programs using NOI: You have to be aware of when you might be building multithreading conflicts into your algorithms. For example, the following program (Listing 10.1) is likely to cause problems:

Listing 10.1 Multithreading Conflicts Example (Ex101Conflict.java)

```
import lotus.notes.*;
public class Ex101Conflict
{
    public static void main(String argv[])
        {
        try {
            NotesThread.sinitThread();
            Session s = Session.newInstance();
            Database db = s.getDatabase("",
"book\\Ex101.nsf");
            Document doc = db.getDocumentByID("20FA");

            // start 2 threads, one to modify doc, one to
delete it
            Ex101Modify mod = new Ex101Modify(doc);
            Ex101Delete del = new Ex101Delete(doc);
            mod.start();
            del.start();
            mod.join();
            del.join();
            }
        catch (Exception e) { e.printStackTrace(); }
        finally { NotesThread.stermThread(); }
        } // end main
}  // end class

class Ex101Modify extends NotesThread
{
    private Document my_doc;
```

```
    public Ex101Modify(Document doc)
        { my_doc = doc; }

    public void runNotes()
        {
        try {
            System.out.println("Modifying Document " +
                               this.my_doc.getNoteID());
            sleep(500);
            for (int i = 0; i < 100; i++)
               this.my_doc.replaceItemValue("item2", new
Integer(i));
            sleep(500);
            this.my_doc.save(true, false);
            System.out.println("Modified document saved");
            }
        catch (Exception e) { e.printStackTrace(); }
        } // end runNotes
}  // end class

class Ex101Delete extends NotesThread
{
    private Document my_doc;
    public Ex101Delete(Document doc)
        { this.my_doc = doc; }

    public void runNotes()
        {
        try {
            System.out.println("About to delete document " +
                               this.my_doc.getNoteID());
             this.my_doc.remove(true);
            }
        catch (Exception e) { e.printStackTrace(); }
        }
}  // end class
```

I created the Ex101 database (available on the CD) with a single document in it, then used the Document Properties box to get its note id. The main() program navigates its way to that Document instance in the Database, then starts two threads, passing the Document to each. One thread iterates over a loop updating (in memory) the contents of one of the Items in the Document. Meanwhile the second thread is deleting the Document. I know that (for pedagogical purposes) the delete thread will complete first, because I've added some sleep() calls to the modify thread to slow it down.

If you want to try this out for yourself, you have to add a document to the database, then get the note id and use it in the main() function instead of "20FA", and finally recompile the sources. To get the note id for any document, select that document in a view, or open it. Then bring up the Document Properties box; right-click on a selected document, or use the **File/Document Properties** menu command. The information tab on the properties box contains a long ID string, with colons separating it into sections, as in Figure 10.1. The last part of the ID, beginning with the characters "NT" is the eight-character note id (not including the "NT"). You can use this eight-character string in the Database.getDocumentByID() call (you can also skip any leading zeros).

Figure 10.1 *Document Properties box.*

When I ran it, I got a NullPointerException on the replaceItemValue() call where it calls an internal routine named NotesBase.CheckObject(). The purpose of the CheckObject() routine is to detect conflicts such as the one I've manufactured here: Somehow between the time a Document instance is created and a call on one of its methods is made, the Document is being deleted. CheckObject() notices during one of the replaceItemValue() calls that the object is no longer valid, and throws the exception.

Now, you can argue, as I do, that both the program and the Notes classes are thread safe, because no data is corrupted, and the program doesn't crash. It's the algorithm that's problematic, not the code. It's NOI's job to make sure that the objects behave as

advertised, and to let you, the programmer, know when you have a conflict. It's your job to anticipate the conflicts that might legitimately arise and to account for them in the code. For example, now that you know about the CheckObject() call, you could add it explicitly into the program, so that you check an object before invoking a method on it. Or, you could catch exceptions as we already do in the example, but make the logic a bit more interesting than just printing a stack trace and bailing out. You could recover in some way to preserve your thread's data: take the contents of the field that you're trying to replace and write it to a new Document, or some such technique.

Memory Management and lotus.notes.Session

One final issue that all Java NOI developers should be aware of has to do with the relationship between instances of the Session class and threads.

First a little background. We've said a few times in this book that the NOI classes comprise a *strict containment hierarchy*, which simply means that every object (with the exception of Session, which is the root object in the containment hierarchy) is contained by a parent object (Documents by Database, Items by Documents, and so on). No object is ever instantiated via the Java *new* operator. The root object (Session) is instantiated via a static method, and all other objects are instantiated by a method on their container [Session.getDatabase(), Document.createRichTextItem(), and so on].

The reasoning behind imposing this design constraint on you was straightforward: In the real Notes implementation, no object can be free-floating; all objects have a container of some sort to provide context. To have a Document exist without a Database to give it context is meaningless in the Domino/Notes world.

The corollary to saying that all objects have a container is that if a container object goes away, then all its subsidiary objects must also go away. In LotusScript, objects that go out of scope in a program are automatically destroyed, because LotusScript maintains a reference count for all object instances and keeps track of everything. Java

works a little differently: When an object reference goes *out of scope* (meaning that the object is no longer pointed to by any other objects and is no longer in the current execution scope) the object is "available" for deletion. It isn't actually destroyed until the VM's garbage collection thread gets around to "collecting" the storage belonging to that object.

The effect is the same, however, in that when a container object is destroyed (meaning that the object in memory is destroyed, *not* that the actual Notes object that the Java object represents is deleted) all objects belonging to that container must also be made to go away. In a Java program, we're never sure if or when the garbage collection thread (affectionately known as the gc thread) will get around to cleaning up memory, and we do know that any given Java program that makes heavy use of NOI can in practice create numerous object instances, which can really run up the memory usage bill. If the gc thread takes its sweet time (remember, the gc thread runs at a lower priority than any thread you create, so it can often get "starved") coming around to clean up, we can end up with hundreds or thousands of objects lying around, unreferenced and useless in memory. For example, consider the following code fragment (Listing 10.2):

Listing 10.2 How to Use Up Memory

```
Document doc = SomeView.getFirstDocument();
while (doc != null) doc = SomeView.getNextDocument(doc);
```

Seems innocent enough, right? Wrong. In LotusScript you'd be fine, because each time you assign a new Document instance to the "doc" variable, LotusScript would destroy the old (now unreferenced) instance automatically. Java, however, simply marks the old instance as unreferenced, and it doesn't get cleaned up until the gc thread comes around, which might be seconds, minutes, or hours later. Or never, because if there were a million Documents in the View (a not unheard of situation), you'd run out of

memory before the gc thread ever got to do its thing. The only way around this is to explicitly call System.gc(), which causes the garbage collector to run synchronously on the current thread. Not the greatest solution, but better than nothing. My own opinion is that it would be nice if JavaSoft were to provide some way for a user program to set the priority of the gc thread.

To help work around this problem, the Java implementation of NOI associates each Session instance with the thread instance on which it was created. Each Session object is simply tagged with the id of the thread that was running when the object came into being. When that thread terminates (and you'll remember from our discussion of NotesThread way back in Chapter 2 that we always know when a NotesThread instance terminates), any Sessions created on that thread are explicitly destroyed. What I really mean by "destroyed" in this case is not the Java notion of just setting everything to *null* so that it can be garbage collected later. I mean that the C++ object for which the Session instance is really just a proxy, or wrapper, is deallocated in Notes memory. This is done so that memory gets cleaned up for you more often than Java might be able to do it. Of course the memory that gets released is C++ memory, not Java memory, as there is no way for any Java program to explicitly force the destruction of any Java object. Still, C++ is where most of the memory in an NOI program is consumed.

The implications of this are important to anyone doing multithreaded programming with NOI: When a NotesThread instance terminates, any Session object created on that thread is killed, and (because of our strict containment rule) therefore *all objects created in that Session's context are also destroyed.*

Take, for example, our little conflict program in Listing 10.1 (Ex101Conflict.java). You'll see that the main() function explicitly waits for each of the two child threads to complete before exiting. I have to do that, because it passed an object (Document) to each of those threads, and that object was created (indirectly) from a Session instantiated on the main() thread. If main() were to exit before both of the child threads

terminated, the Session instance, and the Database and Document instances that came from it, would all be destroyed while still in use on the other threads. By waiting, we ensure the integrity of the Session and all its children.

We could, of course, have coded it differently so that main() would not have to wait. We could have had each child thread create its own Session instance, and navigate to the Document individually. That's the trade-off you have to consider when designing a multithreaded program using NOI: Would you rather have the thread that created the container object hang around and wait until everyone who shares that container (and all objects instantiated from the container) is done? Or would you rather not share objects across threads as much, and have each thread potentially duplicate some code (and take some time) to instantiate its own objects.

My inclination is to lean toward sharing objects where it makes the most sense to do so. If you find that you're stretching the code in some way just so that you can share an object across threads, it probably isn't a good idea. If your algorithm requires (for functional reasons, or for performance reasons, or even just for convenience of coding reasons) that an object be shared by more than one thread, go ahead and do it, no need to be afraid. Just be aware of which threads are using which objects, and which threads are using objects created by which other objects, and make sure that the right threads hang around the right amount of time. To the extent that you can, without violating good design practice, keep the use of an object single threaded (create it, use it, and forget it all on one thread)—your life will be a bit simpler.

Summary

I've tried in this chapter to explore some special issues with respect to multithreaded NOI programming. While most people using the Java NOI for run of the mill applications or Agents won't run into serious trouble, it helps to understand how the

underlying system is really put together when you need to develop a large, production quality piece of code. To sum up, here's my recommended do's and don'ts:

1. Do use multithreading to improve overall application throughput when you can. If your code is doing a lot of i/o (disk or network), consider how you might structure your algorithm to allow several threads to perform independent tasks simultaneously.
2. Do be aware of memory usage issues at all times (the Java hype says you never have to worry about memory management, but you do). Use the System.gc() call judiciously, if necessary.
3. Do share Notes object instances across threads if it makes the overall operation of your program simpler.
4. Do think in advance about how your program will operate in a multi-user world.
5. Do test your multithreaded programs at least twice as much as you do your single-threaded programs.
6 Don't use threads in your program just for fun. If you can't explain to someone else why you need multithreading, you probably don't need multithreading.
7. Don't build conflicts into your multithreaded algorithms and then spend eternity semaphoring them. If the algorithm is broken, no amount of code will fix it.
8. Don't let your Session instance die before you're done with all child objects created (directly or indirectly) from that Session instance.

The next chapter talks about programming Servlets with Domino, and tells you how you can convert an existing Servlet into a Domino Agent (and it also tells you why you would want to do that).