Tips on debugging Java agents

by Bob Balaban

[Editor's note: This article resides in "Iris Today", the technical Webzine located on the http://www.notes.net Web site produced by Iris Associates, the developers of Domino/Notes.]

Introduction

Agents have been a fixture of the Notes/Domino development environment since Release 4.0 shipped over a year and a half ago. Agents, written in LotusScript, can be event triggered by such things as mail arriving in a database, or by a document being created or modified. Agents can also be run explicitly from buttons on a form, from smart icons, and from the "agent view" in the Notes client UI. They can also be set up to run on a scheduled basis, and they can be invoked from a Web browser through the HTTP server process.

Before the advent of Domino 4.6, LotusScript was the language of choice for developing agents (although you also had the choice of programming agents with a "simple actions" wizard and with @function formulas). LotusScript is essentially a version of BASIC, with object-oriented constructs (classes, methods and properties) added. The Notes client software includes a complete Integrated Development Environment (IDE) that you can use to enter, format and debug LotusScript programs. Because the IDE includes a fully capable debugger, with the ability to single step source lines, view and modify the value of variables and of Notes object instances, set execution breakpoints and so on, you can easily debug the most complicated LotusScript program.

Now, with the release of Domino 4.6, we have provided an important new option for developing agents: you can now write them in Java (you can write standalone Java applications too, but the focus of this article is on agents). The Notes Object Interface (NOI, the set of Notes objects that you use to manipulate the product) now has a Java binding as well.

Unfortunately, Domino 4.6 does not contain a Java IDE, so debugging Java agents is not obvious. But fear not, gentle reader, peruse the rest of this article and you'll see that there are ways to crack the Java debugging nut.

Creating a simple Java agent

We'll do this by example. I'll take you through the basic steps for creating a Java agent (though briefly, as the procedure is well documented elsewhere), then show you how to make just a few modifications that will help you debug it.

Let's do one that mimics the famous "hello world" program, yet adds some manipulation of NOI. The following code shows a simple Java class that runs as an agent (triggered in any of the standard ways you'd trigger a LotusScript agent), retrieves the name of the current user and prints a message.

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

public class MyHelloWorld extends lotus.notes.AgentBase
{

public void NotesMain()
{
        try
        {
        Session s = this.getSession();
        AgentContext ctx = s.getAgentContext();
        Name n = s.createName(ctx.getEffectiveUserName());
```

```
            String st = n.getCommon();
            System.out.println("Hello " + st + "!");
            }
            catch (Exception e)
                  { e.printStackTrace(); }
}
}           // end class
```

Now, so long as this code resides in a file named MyHelloWorld.java and the Notes classes that ship with Domino 4.6 reside in a directory that's on your CLASSPATH environment variable (the classes are in a file called notes.jar, and live in the Notes executable directory), you can go ahead and compile this program.

Simply type the following in a command prompt window:

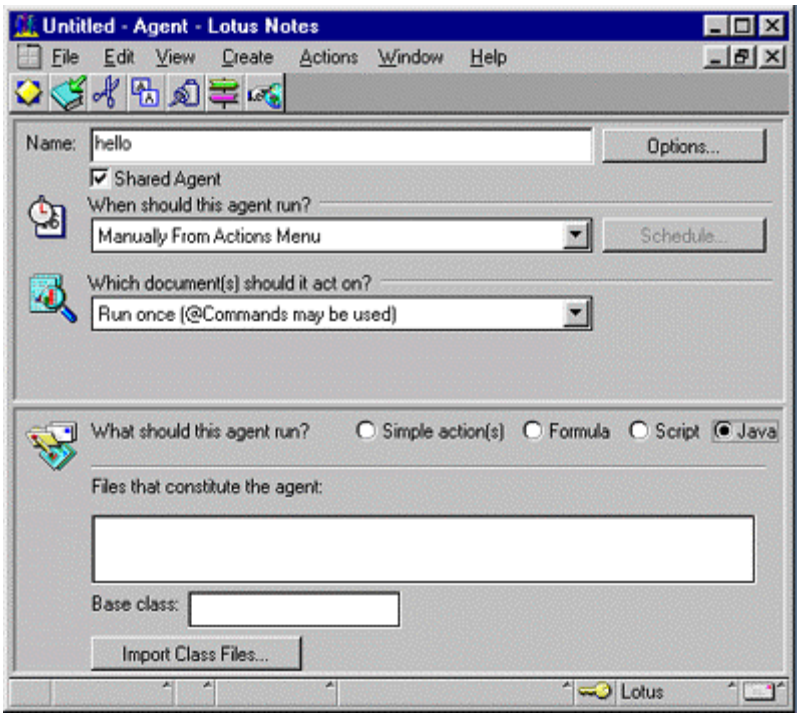
```
java MyHelloWorld.java
```

Note that case sensitivity matters with most Java compilers, and that the file name must match the name of a public class in the file. The Java compiler produces a file called MyHelloWorld.class, which we can now bring into Notes as an agent.

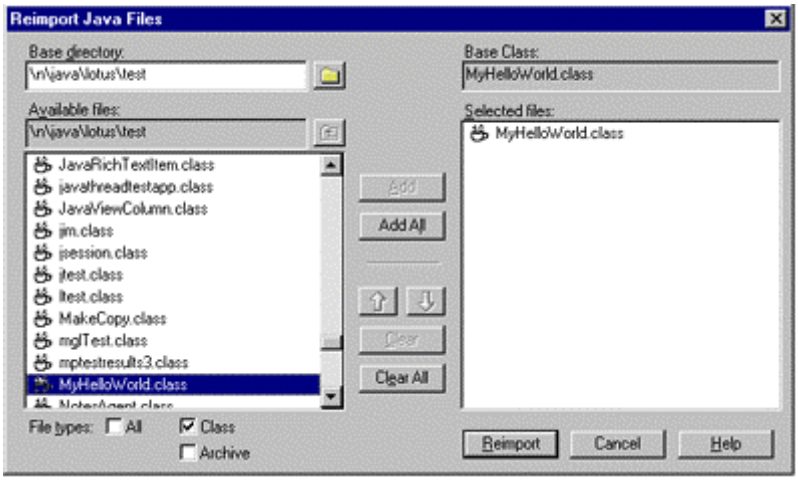
Could we have used a third-party Java development IDE to type in and compile this program? Sure, so long as the tool uses at least Java 1.1.1, you'll have no problem (some tools require that you specially configure a CLASSPATH for it to use, so make sure you add the location of notes.jar to whatever .INI file your tool uses).

Since we're just using any old Java compiler (Sun's JDK, or whatever other tool you like) to create the .class file, why can't we also use some Java IDE to debug it? Well, the answer is that you can try, but it won't work, for two reasons: first, there's no "main" method in our class, or in any of the classes from which it derives. The Java interpreter needs to know what method to call to get a program going, and requires that there be a main() somewhere. Secondly, even if the program were to run (if we wrote a main(), say), it wouldn't work right, because the IDE doesn't know how to provide the agent's "context". We'll come back to this after we finish creating the agent.

Start up the Notes client, and select the database where you want this agent to live. In this simple example, any database, either local or remote, will do just fine, as we aren't doing any data manipulation. Single-click on the database icon to select it, then choose Create - Agent. This brings up the Agent Builder UI. Give the agent a name, decide if it should be shared or not, select the trigger and so on. In this example, we'll call the agent "hello", make it shared, and set it up to be invoked from the menu. Now look down below, and you'll notice that the familiar three radio buttons for the type of agent (simple action, formula, and LotusScript) have been supplemented by a fourth: Java. Select Java. The following screen shows how the Agent Builder looks when we've set up our agent thus far.

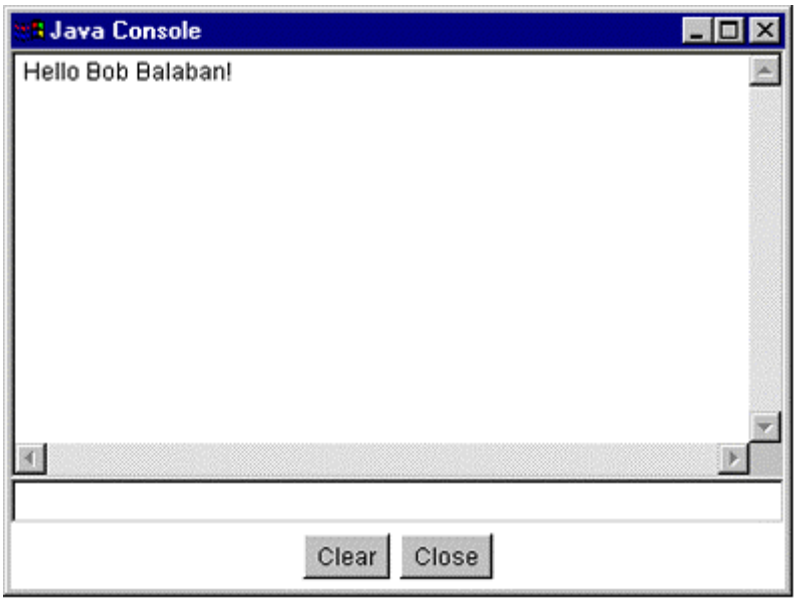


Now, click the "Import Class Files" button way down at the bottom of the screen. This brings up the following dialog box that prompts you to select the file or files that you want to have imported into the database. In this case, we only need the one .class file. We could, if we want, also import the source file for later use. It doesn't hurt anything to do that, but (at least in Domino 4.6) Domino ignores it.



The name of the "main" class to run as the agent must be provided. This is the class that contains the NotesMain() method. Now, we can just save the agent as usual. Let's try it out: choose "hello" from the Actions menu, and see what happens.

Gee, apparently nothing happened. We expected to see a Hello message, but none appeared. That's because the Java System.out output stream doesn't get routed to the Notes UI. Instead, it goes to the Notes server log (if you're running a background agent), or (when running in the foreground as in this case) to the Java console. Let's choose File - Tools - Show Java Debug Console. The following screen appears.



And there's the message we expected.

A few words about agent context

The thing that makes agents so powerful in the Notes context, and that also makes Java agents hard to debug properly, is agent context. What does that mean? Essentially it's a bunch of information that you get automatically when an agent runs: object references to the current database, the current agent, the name of the agent's user, and so on. Some of this information is encapsulated in the Session class, most of the rest is in the AgentContext class.

When Notes runs an agent, it creates an instance of the Session class for you, and populates it with an instance of AgentContext. Notes then creates a Java thread (actually an instance of NotesThread, which inherits from Java's Thread class), and runs an initializer method in the AgentBase class, from which your agent class (MyHelloWorld in this example) inherits. AgentBase contains a method called getSession(), which we used in our sample agent to get the reference to the Session object. Session contains a method called getAgentContext(), which we used to get the context object. Then we used the getEffectiveUserName() method on AgentContext to find out the name of the current user.

Then, just to show off a little, I coded the agent to take the effective user name (which is typically a hierarchical name, such as CN=Bob Balaban/O=Iris), create an instance of the Name class from it, and then simplify the output by getting just the "common" version of the name, which we saw in the output message.

The purpose of all this explanation is to make the point that when you run an agent using Domino/Notes, the product does a lot of work on your behalf, and provides you with all kinds of context information automatically, making the job of writing the agent much, much easier. If you were to run the agent program outside of Notes, however, you'd be missing all the context information.

So it seems, at least on the surface, as though we have to choose between getting some nice agent context information and being able to debug our code with a sophisticated IDE, such as Symantec's Cafe, or Borland's JBuilder. What to do? Well, the next release of Domino will, as of this writing, probably include a real Java IDE of some sort. But we can't wait for that, right? Read on.

Rewriting an agent to also run standalone

Our basic strategy, while not a perfect solution, will let us use any Java 1.1.x compatible IDE tool to code and debug our Java agents. There are two steps to follow: re-code the agent slightly to allow it to execute as

a standalone program; then use our own version of the AgentContext and Session classes to make it possible to use the agent both in debug mode (outside of Notes) and in production mode (inside of Notes), all without re-coding the agent at all in between.

First, let's look at the two classes we'll use in debug mode to replace the standard lotus.notes.Session and lotus.notes.AgentContext.

```
/*** The replacement class for Session ***/

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

public class MySession extends lotus.notes.Session
{
        private Session s;

public MySession()
        throws NotesException
{
        super(1);                               // any old value will do
        this.s = Session.newInstance();         // create a "real" session to
}                                               // redirect calls to


public AgentContext getAgentContext()
        throws NotesException
{
        MyAgentContext mac = new MyAgentContext(this.s);
        return mac;
}

public Name createName(String st)
        throws NotesException
{
        return this.s.createName(st);
}

}       // end class


/*** The replacement class for AgentContext ***/

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

public class MyAgentContext extends lotus.notes.AgentContext
{

public MyAgentContext(Session s)
        throws NotesException
{
        super(s, 1);
}
```

```
public String getEffectiveUserName()
{
        return new String("CN=Bob Balaban/O=Iris");
}

}
```

Before continuing on to examine a re-coded MyHelloWorld, we should stop and consider how these replacement classes are set up. Again, let me stress that this technique is not really optimal, but it really does work, and some of you will find that it's worth the small setup effort involved.

First, you'll notice that MySession *extends* (inherits from) Session, but that it also *contains* another instance of Session that gets created in MySession's constructor. What gives? Well, it sure would be better if MySession could be a "real" Session instance that we overload. Unfortunately, real session instances require a bunch of special setup that only gets done by the static method Session.newInstance(). MySession's constructor doesn't know how to use the native methods that the real Session object uses to call into the Notes DLLs and really start a session.

So instead, we fake everyone out by using the correct call (newInstance) to get a real Session object, save that away, and redirect any Session methods that we want to be able to use in our agent to that instance (as with getAgentContext() and createName() in this example). If our agent needed more Session methods, we'd have to override those too and redirect the call to the real Session. Then why inherit too? Because, as we said above, one of our goals is to set things up so that the agent doesn't have to be re-coded when we go to production mode. To accomplish that, each of our replacement classes must be usable as if they were the real thing, and inheritance accomplishes that.

The constructor for MySession, then, just creates a real Session and saves it. The getAgentContext() call creates an instance of our replacement context class, passing the *real* Session instance into the AgentContext constructor. MyAgentContext, in turn, does not have to contain a *real* AgentContext instance, since we don't intend to redirect any calls. We'll just hardwire MyAgentContext to return something valid for all the calls that the agent uses. In this example, only getEffectiveUserName() is relevant, and I've coded it to return what would really be returned for an agent running in the foreground on my machine.

If we wanted to use the very popular getCurrentDatabase() call, we'd add a version of that method to MyAgentContext that used the real Session it caches to open the database we want, and return that Database instance. And so on. I haven't bothered to do that for this article, but you could take the code as is, and add to it yourself.

Note also that we don't have to create a replacement class for Name. Because we're using a real Session to create a Name instance, we can just use it, and it works fine.

Now let's return to the actual agent code and see how we can write it once and use it both as a regular Java agent and standalone for debugging purposes.

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

public class MyHelloWorld extends lotus.notes.AgentBase
{
        // our version of the session
        private Session s;
```

```
public static void main(String argv[])
        throws NotesException
{
        NotesThread.sinitThread();
        Session dummy = Session.newInstance();          // make sure stuff is
                                                        // initialized
        dummy = null;                                   // then get rid of it
        MySession mysession = new MySession();
        MyHelloWorld mhw = new MyHelloWorld(mysession);
        mhw.NotesMain();
        NotesThread.stermThread();
}


public MyHelloWorld()
{
        super();
}

public MyHelloWorld(Session S)
{
        this.s = S;
}

public Session getSession()
{
        if (this.s != null)
                return this.s;
        else return super.getSession();
}



public void NotesMain()
{
        try
        {
        Session s = this.getSession();
        AgentContext ctx = s.getAgentContext();
        Name n = s.createName(ctx.getEffectiveUserName());
        String st = n.getCommon();
        System.out.println("Hello " + st + "!");
        }
        catch (Exception e)
                { e.printStackTrace(); }
}
}       // end class
```

Take a look first at the new main() routine I've added. This is what the Java interpreter will call when we invoke the class from the command line. The first thing that happens is that we use the static call NotesThread.sinitThread() (with a corresponding term call at the end, which you must never forget!) to make sure that the current thread is properly initialized for making calls into Notes. This is unnecessary for agents invoked via Notes, because AgentBase handles it for you.

Following that we create, and then throw away, a Session instance. This is necessary because (speaking more honestly here than perhaps I should, so go easy on me) I only started writing this article (and working through the debugging issues for Java agents) after it was too late in the 4.6 release cycle to fix up the Notes classes. As a result, the Session constructor calls don't always do the right kind of setup when invoked in the way we're doing here. It's never a problem under normal circumstances, because we'd normally never use the Session constructors this way (and in fact you can't, unless you extend the class). Ninety-nine percent of the time we'd use Session.newInstance() instead.

Next, main() creates an instance of MySession (which is also a Session, don't forget), and uses it to initialize an instance of MyHelloWorld. Note that because main() is a static function (as required by Java), until we explicitly *new* MyHelloWorld, no instance of that class exists. The new MyHelloWorld constructor that I wrote stashes the (My)Session instance away for later use.

Then, main() simply does what AgentBase normally does when you run an agent from Notes: it invokes the agent's NotesMain() method. Note that NotesMain() has not changed at all from the first version of the program. This is where all your normal agent logic goes.

One further trick was necessary to get this to all hang together: we needed a version of AgentBase's getSession() call that returns a MySession instead of a Session. That part is OK, since MyHelloWorld extends AgentBase we can just override the base class's implementation with our own. Our version of the code is pretty simple: if there's a cached Session in our class's member variable, return it. Otherwise, explicitly invoke the real getSession() method in the base class. This works great, because the member variable will only be non-null if our main() method was invoked, which Notes never does. So, when we run this as a real agent, the right stuff happens.

Running the new agent

OK, so after all that, we finally have a version of the agent that should run two different ways: from the command line, and from Notes. Compile the three required files (MyHelloWorld.java, MySession.java and MyAgentContext.java), and type

```
java MyHelloWorld
```

If you set everything up as described, you should get the "Hello!" message in your command window (there's no console window in this case, System.out just goes to the character mode window into which you're typing). Cool! Now if you want to debug the code, you just load it into your favorite Java IDE and party on.

Will it still work as a real Notes agent? Sure it will, however you now need the debugging classes (MySession and MyAgentContext) as well. Right-click on the database where you created the original Hello agent, and select Go To Agents from the pop-up menu. Edit the agent (press Enter after selecting it in the agent view), and click the Reimport Class Files button. A dialog box very similar to the one you used to import the original .class file comes up. Add the two additional class files, and click the Reimport button. Then, press Escape to exit the Agent Builder and save your changes. Choose Actions - Run to run the agent, bring up the Java Console again, and your "Hello!" message should be right there. You don't even have to recompile MyHelloWorld, though for production mode you might want to comment out the references to MySession and MyAgentContext so that you don't have to load the two extra files into the database.

Conclusion

Hack? Sure. Big, bad hack? No, I don't think so. Luckily, if things go as planned with the next release, none of this will be necessary anymore. In the meantime, think of these techniques as a relatively low effort work-around for the lack of a Java debugger integrated with the Designer client.

ABOUT THE AUTHOR

After 10 years, 3 months and 6 days as a Lotus/Iris developer, Bob has now hung up his Lotus badge to become a Business Partner and consultant/contractor. His new company, Looseleaf Software, Inc., will soon have its own Web site at **http://www.looseleaf.net**. Bob can be reached via e-mail at bob@looseleaf.net. He hopes to have a new book on Domino 4.6 and Java available in time for Lotusphere.