Debugging LotusScript: Domino Applications Part 2

Level: Intermediate
Works with: Notes/Domino
Updated: 01-Aug-2003

by
Andre
Guirard

In Part 1 of this article series, we discussed the LotusScript debugger and common error messages. In Part 2, we talk about debugging best practices and introduce more advanced debugging techniques for situations in which the debugger can't help you. Topics covered in this article include:

- Making errors "in the field" return better diagnostic information
- Macro formulas
- LotusScript in dialog boxes
- Scheduled agents and Web agents

This article uses the same sample Notes database referenced in Part 1, which you can download from the Sandbox. The database contains example agents and script libraries that you can reuse to improve your own code's error reporting. The screen shots and names of menu items shown in this article are taken from Notes/Domino 6.5 (Beta version). However, except as noted, everything discussed here should also work in earlier versions back to R5.0.2.

This article assumes that you're an experienced Domino application developer with some familiarity with LotusScript.

Debugging best practices

In this section, we examine:

- "Defensive" coding with Assert
- Print vs. Messagebox vs. NotesLog
- Trapping errors to get a call stack

Defensive coding with Assert

If you're familiar with the concept of defensive driving, then you can think of Assert statements as defensive programming. The idea is to keep an eye out for any little problems and to slam on the brakes before they turn into big problems.

Java implements the Assert statement; LotusScript does not. However, you can create your own LotusScript Assert statement by writing the subroutine shown in this section. Where would you use this? Suppose you have a subroutine to delete all documents created before a certain date. The date is supplied as an argument to the routine:

```
Sub DeleteAllBefore(cutoffDate As Variant)
```

You expect cutoffDate not to be in the future; if it is, the search you're using would delete *all* the documents—very bad. If the value you're passed isn't actually a date, that may also be bad because you're not sure which documents may end up being deleted.

Based on your knowledge of the code that calls this routine, you think that you'll never receive anything except a past date. But just in case you ever do, the code should respond more reasonably than deleting all the documents—it should treat a future date or non-date as a serious error and halt execution. Here's the LotusScript subroutine that lets us write an Assert statement:

```
Sub Assert(Byval condition As Integer, message As String)
    If Not condition Then
        Dim callerName As String
        callerName = Getthreadinfo(10) ' LSI_THREAD_CALLPROC
         Stop ' in case the user has the debugger on.
        Msgbox "Assert failure in " & callerName & ": " & message, 16, MSG_TITLE
        End ' which means, abort execution of this script.
    End If
End Sub
```

We use the End statement here, rather than throwing an error, because we expect this code never to execute. If it does, the situation is so grave that the script must end right away. If we just throw out an error, another module might intercept it with an On Error statement and ignore it. After you have this subroutine, you can call it wherever you want to perform a one-line test to guarantee some condition before you proceed. In the previous example, this would be as follows:

```
Sub DeleteAllBefore(cutoffDate As Variant)
    Assert Datatype(cutoffDate) = V_DATE, "cutoffDate must be a date/time value"
    Assert cutoffDate < Today, "cutoffDate must be earlier than today"
...
```

Using Assert slows your code a little because it takes time to evaluate the condition and to make a subroutine call. But you may decide the cost is worth it for preventing disasters. (Note that the Debug script library in the sample database contains a more elaborate version of this subroutine, which takes advantage of the "trace" functionality discussed later in this article.)

Print vs. Messagebox vs. NotesLog

For situations in which you can't use the debugger, add output statements to your code so that you can see which parts of the code are executing and what the values of variables are. In both client-side code and server agents, you can do this with Print or Messagebox statements or the NotesLog class. NotesLog is more useful for code that's already released or for situations in which you expect a lot of debug output and want to put it in a file. During development, it's a matter of personal preference whether you use Print or Messagebox. We generally prefer Print in client-side code because it doesn't require a click to continue, and you don't have to rely on your memory of which messages came up in which order—you can look at the status bar history or, if using the debugger, the Output tab. In the debugger's Output tab, you can copy the output to the clipboard if you want.

When debugging scheduled server agents, Print and Messagebox do the same thing: The output goes to the server console and to the Miscellaneous Events log document in Log.nsf. Optionally, it may also be logged to a text file on the server, depending on server settings. (See "Agent Manager debugging information" in the Domino Designer help.) With a Domino Web agent, there is a difference between Print and Messagebox. Except in Webquerysave events, Print output is sent to the browser, while Messagebox output is sent to the console and error log, same as other server agents.

NotesLog is the only way to write to the agent log of a server agent. While there are other ways besides NotesLog to log to a text file, to a mail message, or to a Notes database, this class simplifies matters by giving you a single standard interface for writing log information, regardless of where the information ends up.

Trapping errors to get a call stack

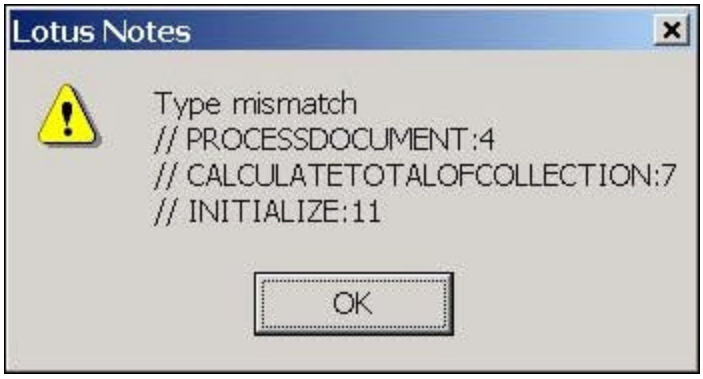
LotusScript error messages can sometimes be less informative than we would like. For example, open the sample database, highlight the oaken bucket document, and click the action total \ 1. original. You should get a "Type mismatch" error. You can read about this message in the list of common errors from Part 1 of this article series, but the point is that there's no indication where in the code the error occurred. This is inconvenient when someone calls it in as a support incident on an existing application. You may not be able to easily reproduce the error based on information from the user, and in a long script, there's no telling where in the code it may have happened.

To avoid that kind of situation, consider routinely coding your applications to provide more complete information about any unexpected errors they encounter. Add just a few lines like the following to every subroutine and function that may possibly fail:

```
Sub YourRoutine
    On Error Goto Repeater
     … ' put the rest of your code here.
     Exit Sub ' or Exit Function if this were a function
Repeater:
    Error Err, Error & {
//} & Getthreadinfo(1) & {:} & Erl
    ' 1 = LSI_THREAD_PROC
End Sub
```

This error trap is what we call a *Repeater* . Whenever there's an error, it repeats the same error to the module that called it, but with a little added text on the end. This additional text contains the name of the function in which the error occurred and the line number. To ensure the exact same code can be used in each module, use the built-in function Getthreadinfo to find out the name of the current module. (Note that the constant LSI_THREAD_PROC is defined in lsprcval.lss and lserr.lss. We hardcoded the constant 1 instead of using the name so that this code would work universally, but for your own code, it's better to use a meaningful name instead of a "magic number.")

The routine that called this one should have the Repeater code also to add its name and line number and to repeat the error back to its caller. The error gets passed back up the chain until it reaches the main routine (Initialize in case of an agent), which adds its own name and line number and lets Notes display the result. The output looks like this:



This would be interpreted as follows: There was a type mismatch on line 4 of ProcessDocument, which was called from line 7 of CalculateTotalOfCollection, which was called from line 11 of Initialize.

If this error text is reported to you, you have enough information to look at the code in Domino Designer, to find the line on which the error occurred, and to see just how it got to that point. (Note that when editing code, the line number within the current module is displayed in the lower right corner of the editing window.)

An On Error statement at the top of the module does not prevent you from doing other error handling within the

module. You can override the general purpose repeater by adding more On Error statements that look for specific errors and do something different in that case, for instance:

```
On Error Goto Repeater
On Error ErrSubscriptOutOfRange Resume Next
```

Note: The constant ErrSubscriptOutOfRange is defined in lserr.lss.

With the preceding code, a subscript out of range error is ignored, whereas any other error jumps to Repeater. Or, if you're not sure what the error number may be, you can temporarily switch all errors to a different handler. Just be sure to set it back to Repeater afterwards:

```
On Error Goto Repeater
 …
‘ if ComputeWithForm fails, handle error silently.
On Error Goto ValidationFailure
doc.ComputeWithForm True, True
On Error Goto Repeater ‘ resume normal “repeater” error handler
```

If you know the exact error code to expect, it's better to use that (as we did in the first example) instead of redirecting all errors. You may sometimes get an error from what you expect. For instance, in the above code, suppose you got an error not because the form data were incorrect, but because the form no longer exists in the database. It makes sense to display the latter error and halt, instead of handling it as you would a validation error.

So now we're cookin'! In case of an unexpected error, *if* we can get the user to report the error correctly, we can get a pretty good idea what happened. But there are still some aspects of this error-handling model we could try to improve:

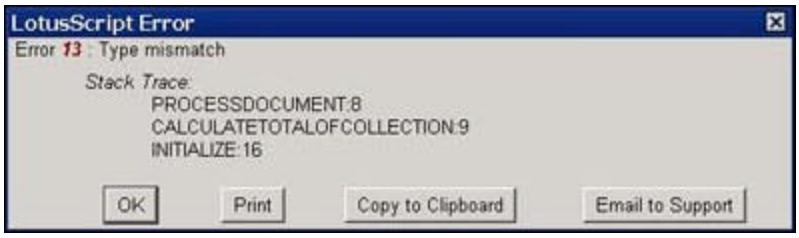
- Getting a correct report of an error message from users can be problematic. Not all users understand the importance of copying down the error text exactly.
- Even if they know what to send, users may not know to whom to send it.
- Users generally send a screen capture bitmap rather than retyping the error. If we could get the text instead, it would make adding the information to a support knowledge base (or searching for it there) faster and less error prone, as well as take up less disk space.
- Sometimes your workstation won't reproduce the problem. Is there any way to find out more about what the code was doing when it went blooey, other than installing Domino Designer on the user's workstation to debug?

Better error reporting

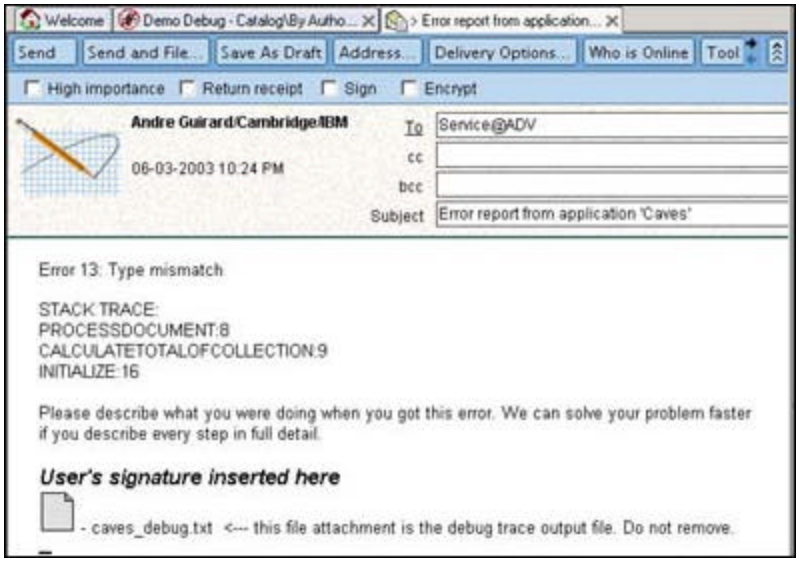
Yes, Virginia, there is a better way. For mission-critical applications, where it's worth some extra effort to get top-notch error reporting, you can get:

- An error dialog that displays the stack trace (as we already saw)
- A button to automatically email diagnostic information to support staff
- Error reports in text form rather than bitmaps
- Detailed logging that you can turn on and off on a user's workstation so that after he gets an error, you can see exactly what the code is doing—for example, which document it was processing at the time

There's a fair bit of code involved in doing this, but the code has already been written. Copy it from the sample database, and with very few changes, you can use it in your own applications. You can see this system in use in the sample database. Highlight the oaken bucket document, and use the action totals \ 3. with trace. This displays the following error dialog box:



The four buttons do what they describe: Copy to Clipboard copies the text of the message, not the bitmap image. Email to Support creates a memo, as shown in the following:



And what's that file attachment? Here's the text from it:

```
----- begin trace 3-Jun-2003 22:23:51 -----
@22:23:51   INITIALIZE: begin
@22:23:51   INITIALIZE: 1 documents found.
@22:23:52      CALCULATETOTALOFCOLLECTION: begin
@22:23:52      CALCULATETOTALOFCOLLECTION: fetch first
@22:23:52         PROCESSDOCUMENT: begin
@22:23:52         PROCESSDOCUMENT: Retrieving price
@22:23:52         PROCESSDOCUMENT: noteID = 902
@22:24:00      INITIALIZE: Error 13: Type mismatch

    STACK TRACE:
    PROCESSDOCUMENT:8
    CALCULATETOTALOFCOLLECTION:9
    INITIALIZE:16
```

This contains another copy of the stack trace and in addition, has a list of messages "logged" by the code as it executes. The agent contains special statements that write to the log to show where it is in the code.

The timestamp on each line helps track down performance issues. A "begin" line shows that we are entering a subroutine or function (the subroutine may optionally display the values of its arguments). The level of indentation shows that Initialize called CalculateTotalOfCollection, which called ProcessDocument. And at certain points in the code, there are additional entries telling what's going on (fetch first) or displaying values from variables (noteID = 902).

The creation of a log file would be turned off by default; when a user is having a problem, a support tech sends an email with a button the user can click to enable logging. You can send yourself such a memo with the action "support – send debug memo" in the sample database.

Adding the full "Trace" functionality to your code

To produce error output like this in your own applications, copy these design elements from the sample database to your application:

- The Debug and ServerAgentDebug script libraries (customize two constants in (Declarations))
- The ErrorDialog form
- The DebugMemo form (customize button formulas with an environment variable name)
- The SendDebugMemo agent

For LotusScript—other than server agents—in the (Options) section of your script, add the following statement:

```
Use "Debug"
```

Note: In server agents, use the ServerAgentDebug script library instead.

In the main module (for example, Initialize for an agent), insert the following code at the top:

```
    Dim trak As New StackTrack("")
    On Error Goto ErrorTrap
    Dim ses As New NotesSession
    If ses.GetEnvironmentString(" EnvVarName ") = "1" Then
        Call trak.OpenLogFile("filename ", tempFlag , appendFlag )
    End If
```

where:

- *EnvVarName*
  is the name of an environment variable that controls whether or not a trace file is created. If its value is 1, there is a trace.
- *filename*
  is the name of the trace output file.
- *tempFlag*
  is True if the file is to be created in the user's Windows TEMP directory. If False, *filename* should contain the complete file path for the output file. It's generally a bad idea to create files in Notes with a relative path name because the current directory is not necessarily always going to be the same.
- *appendFlag*
  is True if you want to append this output to any previously existing contents of the file. You should append when your code is in a dialog form or in a separate agent called from the main code, and you want to have a single log file for all the code.

Note that in the TotalPoints Trace agent, we don't check an environment variable before calling OpenLogFile because we want the trace output to always happen for purposes of demonstrating this system. Also, in server agents, use OpenAgentLog( ) instead of OpenLogFile(…). You may also want to use something other than an environment variable to decide whether or not to start logging.

Next, add this code at the end of the main module:

```
    trak.Finish ' so log shows we ended the program normally.
    Exit Sub
ErrorTrap:
    trak.DisplayError
    Exit Sub
```

If you don't call trak.Finish before you exit, the message "Abnormal termination" is written automatically to the log file. This may happen because of an error abort, because of an Assert, or because the user pressed Ctrl+Break.

In modules other than the main module, add these lines shown in bold:

```
Function YourFunc(…)
    Dim trak As New StackTrack("")
    On Error Goto ErrorTrap
    ‘ insert your code here
    Exit Function ‘ or Exit Sub
ErrorTrap:
    trak.ErrorTrap
End Function
```

At any point where you want to post a notice in the trace file, use the statement trak.Write *string.* For instance, the “noteID =" line shown previously was added by the following source line:

```
trak.Write “noteID =“ & doc.NoteID
```

Don’t include the routine name or time; the trak object adds those automatically.

Trak.Write doesn’t do anything if the trace file isn’t opened, so you’re not losing much time off your execution. However, if your *string* expression is complex, you may want to avoid evaluating it needlessly by testing trak.isLogging first. For instance, if you have a 100-element array AllNames, you could insert its value into the log at this point, but you wouldn’t want to take the time to loop through 100 values to create a string if it were just going to be thrown away. So use the following line of code:

```
If trak.isLogging Then trak.Write {AllNames = "} & Join(AllNames, {", "}) & {"}
```

Note: The Join built-in function is not available in R5.

The Debug script library also contains the function debugStr which converts an array to a string for debug output purposes. You can also use debugStr with a value of any other type, and it returns a string describing its value. So you could instead write:

```
If trak.isLogging Then trak.Write {AllNames = } & debugStr(AllNames)
```

You may want to copy debugStr for use in other contexts, for instance, to use with Print statements or NotesLog calls in server agents.

Of course, you don’t need a custom script library if you want to produce a log file to trace your execution—it’s just as easy to use the existing NotesLog class for that. The benefit you get from using the Debug script library is that the trak objects keep track of which module you’re in and your depth in the call stack, and it shows that information in the log file.

When the Debugger doesn’t work

As useful and versatile as the LotusScript Debugger is, there are situations in which it is of limited help. These include debugging:

- Dialogbox calls
- Macro language
- Server and Web agents

Not to worry—this section explains what you can do to debug these types of code.

Debugging dialogbox calls

The Notes debugger does not pause while executing LotusScript in a window that was opened with the @DialogBox function (or NotesUIWorkspace.DialogBox). This can make it a challenge to figure out where an error is coming from. Because you use a form (or subform) in the dialogbox, there may be code in that form’s events. Adding the stack trace and debug logging, as described previously, is likely to help you figure out what’s going on. Or you can try the dialog form in a regular document window instead so that you can use the

debugger. Use the Notes Preview function in Domino Designer. Or, if you must have values in the fields, here's LotusScript to fill in the field values and open the form full screen rather than in a dialog box:

```
Dim ses As New NotesSession
Dim ws As New NotesUIWorkspace
Dim db As NotesDatabase
Dim dialogDoc As NotesDocument
Set db = ses.CurrentDatabase
Set dialogDoc = db.CreateDocument
With dialogDoc
    .Form = " yourDialogForm "
    ' set any other fields needed in the dialog, e.g.:
    Set .StartDate = New NotesDateTime(DateNumber(2003, 4, 13))
    ' ...
End With
Call ws.EditDocument(True, dialogDoc)
```

You can also use repeaters or use the trace functionality described in the previous section to create a record of what happens in the LotusScript code of the dialog box. If you do the latter, you may want to have a single log file with the dialog box log messages and the caller's log messages. Use trak.Suspend before opening the dialog box to close the log file; this lets the code in the dialog box use the same file.

Debugging macro language

R5 has a macro language debugging facility (not available in Notes/Domino 6 as of this writing) that you enable by pressing Ctrl + Shift, while you select the menu entry to turn on the LotusScript debugger. Unlike the LotusScript debugger, the amount of code that executes when you step is not a line, but an expression. So for instance, when you step through the following formula:

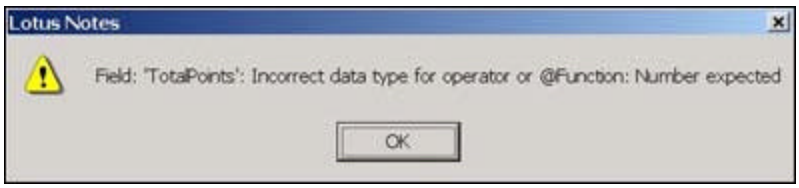
```
@Name([CN]; @UserName) + @Unique
```

it first shows you what value it calculated for @UserName, then what value was returned by @Name, then what was returned by @Unique, then what was returned by concatenating the strings together. This can be interesting to watch as a way to learn how macro language expressions are evaluated, but as a debugger, it's a bit tiresome, especially if the formula you're having trouble with is in the 78th field on the form.

Usually, it's quicker to debug macro formulas if you just recognize the different error messages and know what they mean and test your inputs. By far the most common formula problem we've seen is an incorrect assumption about the datatype and contents of the formula input values. For instance, there's nothing wrong with this formula:

```
Worth * HowMany
```

except that it's simply assuming that its inputs (Worth and HowMany) both contain valid number values. If this is true, no problem; if not, there's trouble. If the formula that causes an error is in a field, Notes displays an error and tells you in which field it found the problem. For example, if you use the new \ Catalog 1 action in the sample database, you get the following error:



The TotalPoints field uses the formula Worth * HowMany. At first glance, it seems that there is nothing wrong with this; Worth is a number field and so is HowMany, so where's the problem? We created a second version of the Catalog form, Catalog 1 debug, which shows how to use @Prompt to find out which values are in the inputs

to the formula. The @Prompt statement is rather long, but you just need to fill in your fieldname in the first line. The rest you can copy and paste.

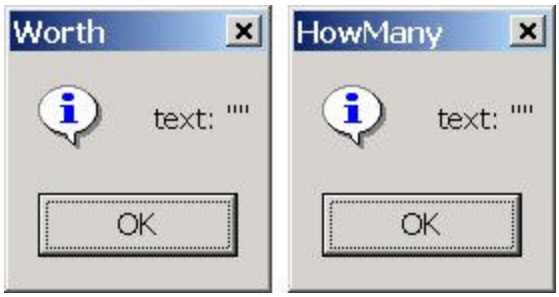
```
_fldnam := "fieldname ";
_fldval := @GetField(_fldnam);
@Prompt([Ok]; _fldnam;
    @If( @IsText(_fldval); "text: \"" + @Implode(_fldval; "\": \"") + "\"";
        @IsTime(_fldval); "time: " + @Implode(@Text(_fldval); ", ");
        @IsNumber(_fldval); "number: " + @Implode(@Text(_fldval); ", ");
        @IsError(_fldval); "error: " + @Text(_fldval);
        "other type: <" + @Left(@Text(_fldval); 100) + ">"
    )
);
```

Note: In Notes 6, we recommend using @StatusBar rather than @Prompt because there are fewer mouse clicks and because you can see all the messages at once in the history.

This is also useful for displaying temporary variables containing intermediate results. To do this, substitute it for the @GetField on line 2 of this code, for example, _fldval := tmp2.

You can put any number of @Prompt statements into your formula without affecting the value returned by the formula, providing you put them all before the last "expression" statement of the formula (in other words, the last one that's not an assignment or SELECT statement). Expression lines return a value, and the last line that returns a value is the value of the formula. You wouldn't want to use the return value of the @Prompt for the value of the formula. Note that if the value returned by the formula is not used (such as buttons or form events) it doesn't matter whether or not you use @Prompt as the last line.

Using two @Prompt statements like the above, we can find out what the values of Worth and HowMany are in the Catalog 1 debug form. Here's what we get:



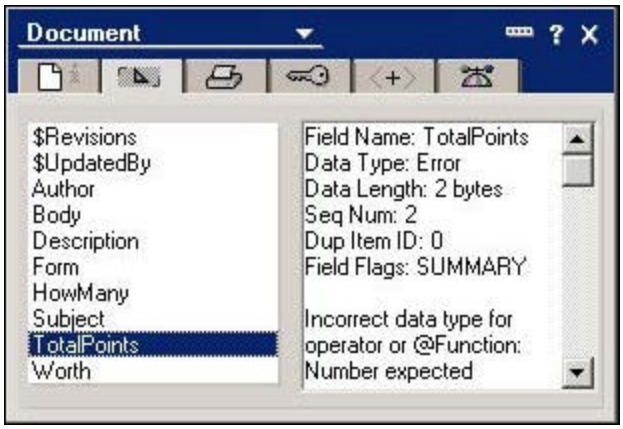
When the formula is evaluated, both Worth and HowMany are not numbers; they have the value "" because they haven't been assigned values yet. That's because we're trying to evaluate this formula when composing the form and the fields have no default value. Our formula should take into account that these number fields may not have anything in them, both on compose and later, when refreshing or saving the document. You need to be careful with your formulas on save because if they calculate an error value, Notes does not display an error message; it just saves the document with an "error" value in the field.

Form Catalog 2 demonstrates a fix to the original problem that didn't go quite far enough—it fixes the error on Compose, but not on Save. The new formula is:

```
@If(@IsNewDoc & @IsDocBeingLoaded; 0; Worth * HowMany)
```

This is fine when the document is composed, but if either Worth or HowMany is still blank when the document is saved or refreshed, the field still calculates an error value. An error value in a field doesn't prevent the document from being saved, and you can see such a document (crystal wand) in the sample database. In this case, the error is fairly obvious because the field is displayed both on the form and in a view, but you can get such errors in hidden computed fields, which are more difficult to track down. Your most valuable tool for finding out whether

or not the fields in stored documents have correct types and values is the Document Properties Box:



Looking at the properties of the crystal wand document, you can see that the TotalPoints field has an error value. Also use this dialog box to make sure that your number fields are really numbers, not text values that happen to contain digits, and that your date fields are really dates. To be really bulletproof, you should use @IsError (or in Notes/Domino 6, the more succinct @IfError) to test the value you're planning for your formula to return to see whether or not it's valid. The final version of this form, Catalog 3, uses the following formula in the TotalPoints field:

```
tmp := Worth * HowMany;
@If(@IsError(tmp); 0; tmp)
```

TotalPoints now always contains a number, not an error value.

By the way, error values can have an interesting consequence when you work with the documents in LotusScript. Refer to the discussion of the "Variant does not contain a container" error in Part 1 of this article series for details.

Tip: When a formula is not working and you're trying to figure out what's wrong, try taking out the @IsError test temporarily. By intercepting the error and substituting a default value, the formula prevents you from seeing the text of the actual error, which is usually a good clue.

Debugging server agents

The *LDD Today* article "Troubleshooting agents in Notes/Domino 5 and 6" contains a great deal of useful information about debugging server agents. We only have a couple of additions:

- Use the error Repeater technique described previously to get stack trace information for any unexpected errors.
- If you want a more comprehensive log showing which statements executed in which order and which documents they processed, you can use a NotesLog object to write to the agent log or to a file.

Debugging Web agents

Again, refer to the "Troubleshooting agents" article, noting that the repeater works in this context also, except that the Initialize function shouldn't just use another Error statement in its error trap. The browser user is waiting for some results. If the error prevents you giving them the information they're waiting for, printing an error message (even with a call stack) is a poor substitute. Use Messagebox to have the call stack appear in the log (and maybe use NotesLog to generate a mail message to the Web support team), but use Print to give the user an explanation why you can't complete the request, tell the user whom to notify of the problem, and redirect him to a more useful page in your site.

Conclusion

This concludes our series on using the LotusScript debugger. With the techniques we've described here and in Part 1, you should now have a good grounding (and good tools) to debug any problem that may develop.

Perhaps your code never contains any errors, but ours does on occasion, so we've found these techniques very useful. There's always another tweak that can make your error handling more perfect, so we encourage you to play around with the code in the sample database and improve it—then let us know about your better way. We're all learning this stuff…

That's all for now. Go forth and correct errors!

ABOUT THE AUTHOR

Andre Guirard is a member of the Enterprise Integration team of IBM Lotus Software, the developers of Lotus Enterprise Integrator (LEI) and other products that let you connect disparate data sources with each other and with Lotus Notes. Andre has made occasional appearances as a speaker at IBM conferences, and his articles have previously appeared in *The View* magazine and elsewhere.

