CATCH Me if You Can

Doug Hennig

VFP 8 has structured error handling, featuring the new TRY ... CATCH ... FINALLY ... ENDTRY structure. This powerful new feature provides a third layer of error handling and allows you to eliminate a lot of code related to passing and handling error information. This month, Doug discusses structured error handling and shows how it fits in with a complete error handling strategy.

VFP 3 greatly improved the error handling ability of FoxPro by adding an Error method to objects. This allowed objects to encapsulate their own error handling and not rely on a global error handler. However, one of the downsides of putting code in the Error method of your objects is that it overrides the use on the ON ERROR command. That makes sense, since to do otherwise would break encapsulation. However, one problem with this is that if an object with code in its Error method calls procedural code (such as a PRG) or a method of another object that doesn't have code in its Error method, and an error occurs in the called code, the Error method of the calling object is fired, even if the called code set up a local ON ERROR handler. There are two problems with this mechanism:

- Many types of errors can be anticipated in advance, such as trying to open a table that someone else might have exclusive use of. However, since the ON ERROR handler set up by the called routine isn't fired, the routine doesn't get a chance to handle its own error.
- Since the calling object has no idea what kind of errors the called routine might encounter, how can it possibly deal with them except in a generic way (logging it, displaying a generic message to the user, quitting, etc.)?

Clearly, we need a better mechanism. Fortunately, VFP 8 gives us a better tool: structured error handling.

Structured Error Handling

C++ has had structured error handling for a long time. .NET adds structured error handling to languages that formerly lacked it, such as VB.NET. So what the heck is structured error handling? Structured error handling means that code in a special block, or structure, is executed, and if any errors occur in that code, another part of the structure deals with it.

VFP 8 implements structured error handling in the following way:

- The TRY ... ENDTRY structure allows you to execute code that may cause an error and handle it within the structure. This overrides all other error handling.
- A new THROW command allows you to pass errors up to a higher-level error handler.
- A new Exception base class provides an object-oriented way of passing around information about errors.

Let's take a look at these improvements.

TRY, TRY Again

The key to structured error handling is the new TRY ... ENDTRY structure. Here's its syntax:

```
try
  [ TryCommands ]
[ catch [ to VarName ] [ when lExpression ]
  [ CatchCommands ] ]
  [ exit ]
  [ throw [ uExpression ] ]
[ catch [ to VarName ] [ when lExpression ]
  [ CatchCommands ] ]
  [ exit ]
```

```
  [ throw [ uExpression ] ]
[ ... (additional catch blocks) ]
[ finally
  [ FinallyCommands ] ]
endtry
```

TryCommands represents the commands that VFP will attempt to execute. If no errors occur, the code in the optional FINALLY block is executed (if present) and execution continues with the code following ENDTRY. If any error occurs in the TRY block, VFP immediately exits that block and begins executing the CATCH statements.

If *VarName* is included in a CATCH statement, VFP creates an Exception object, fills its properties with information about the error, and puts a reference to the object into the *VarName* variable. *VarName* can only be a regular variable, not a property of an object. If you previously declared the variable, it will have whatever scope you declared it as (such as LOCAL); if not, it will be scoped as PRIVATE. We'll look at the Exception base class later.

The CATCH statements can act like CASE statements if the optional WHEN clause is used. The expression in the WHEN clause must return a logical value so VFP can decide what to do. If the expression is .T., the code in this CATCH statement's block is executed. If it's .F., VFP moves to the next CATCH statement. This process continues until a CATCH statement's WHEN expression returns .T., a CATCH statement with no WHEN clause is hit, or there are no more CATCH statements (we'll discuss the last case later). Normally, the WHEN expression will look at properties of the Exception object (such as ErrorNo, which contains the error number).

Once VFP finds a CATCH statement to use, the commands in that block are executed. After the block is done, the code in the optional FINALLY block is executed (if present) and execution continues with the code following ENDTRY.

The following example, taken from SimpleTry.prg included with this month's Subscriber Downloads, shows how the CATCH statements are evaluated, and that the TRY structure overrides the ON ERROR setting.

```
on error llError = .T.
llError = .F.
try
  wait window xxx
catch to loException when loException.ErrorNo = 1
  wait window 'Error #1'
catch to loException when loException.ErrorNo = 2
  wait window 'Error #2'
catch to loException
  lnError = loException.ErrorNo
  messagebox('Error #' + transform(lnError) + chr(13) + ;
    'Message: ' + loException.Message)
finally
  messagebox('Finally')
endtry
on error
messagebox('ON ERROR ' + iif(llError, 'caught ', ;
  'did not catch ') + 'the error')
```

If VFP doesn't find a CATCH statement to use, then an unhandled exception error (error 2059) occurs. You don't really want that to happen for a variety of reasons, the biggest one being that the problem that caused the original error isn't handled because now we have a bigger mess on our hands.

Here's an example of what happens when you have an unhandled exception (taken from UnhandledException.prg). When you run this code, you'll see an error message indicating that the error is an unhandled exception rather than the problem that started it, variable XXX not existing.

```
on error do ErrHandler with error(), program(), lineno()
try
  wait window xxx
catch to loException when loException.ErrorNo = 1
  wait window 'Error #1'
catch to loException when loException.ErrorNo = 2
```

```
  wait window 'Error #2'
finally
  messagebox('Finally')
endtry
on error

procedure ErrHandler(tnError, tcMethod, tnLine)
local laError[1]
aerror(laError)
messagebox('Error #' + transform(tnError) + ;
  ' occurred in line ' + transform(tnLine) + ' of ' + ;
  tcMethod + chr(13) + 'Message: ' + message() + ;
  chr(13) + 'Code: ' + message(1))
```

You Can't Go Back

One important difference between structured error handling and the other types of VFP error handling is that you can't go back to the code that caused the error. In an Error method or ON ERROR routine, there are only a few ways to continue:

- RETURN (or an implied RETURN by having no RETURN statement) returns to the line of code following the one that caused the error. This typically gives rise to further errors, since the one that caused the error didn't complete its task (such as initializing a variable, opening a table, etc.).
- RETRY returns to the line of code that caused the error. Unless the problem was somehow magically fixed, it's almost certain to cause the same error again.
- QUIT terminates the application.
- RETURN TO returns to a routine on the call stack, such as the one containing the READ EVENTS statement. This is very useful if you want to stay in the application but not go back to the routine that caused the error. Of course, that doesn't mean all is well, but frequently allows the user to do something else if the error wasn't catastrophic (for example, a simple resource contention issue while trying to get into a form).

In the case of a TRY structure, once an error occurs, the TRY block is exited and you can't return to it. If you use RETURN or RETRY in the CATCH block (actually, if you use them anywhere in the structure), you'll cause error 2060 to occur. You can, of course, use QUIT to terminate the application.

FINALLY We Can Clean Up After Ourselves

One things it took me a while to figure out was why the FINALLY clause was necessary. After all, the code following the ENDTRY statement is executed regardless of whether an error occurred or not. It turns out that this isn't actually true; as we'll see when we discuss the THROW command, errors can "bubble up" to the next higher error handler, and we may not return from that error handler. That means we can't guarantee the code following ENDTRY will execute. However, we can guarantee that the code in the FINALLY block will always execute (well, almost always: if a COM object's error handler calls COMRETURNERROR(), execution immediately returns to the COM client).

Here's an example that shows this (UsingFinally.prg). The call to the ProcessData function is wrapped in a TRY structure. ProcessData itself has a TRY structure, but it only handles the error of not being able to open the table exclusively, so the WAIT WINDOW XXX error won't be caught. As a result, the error will bubble up to the outer TRY structure in the main routine, and therefore the code following ENDTRY in ProcessData won't be executed. Comment out the FINALLY block in ProcessData and run this code. You'll see that the Customers table is still open at the end. Uncomment the FINALLY block and run it again; you'll see that this time, the Customers table was closed, so the code properly cleaned up after itself.

```
try
  do ProcessData
catch to loException
  lnError = loException.ErrorNo
  messagebox('Error #' + transform(lnError) + ;
    ' occurred.')
endtry
```

```
if used('customer')
  messagebox('Customer table is still open')
else
  messagebox('Customer table was closed')
endif used('customer')
close databases all

function ProcessData
try
  use (_samples + 'data\customer') exclusive
* do some processing
  wait window xxx

* Handle not being able to open table exclusively.

catch to loException when loException.ErrorNo = 1705
* whatever

* Clean up code. Comment/uncomment this to see the
* difference.

finally
  if used('customer')
    messagebox('Closing customer table in FINALLY...')
    use
  endif used('customer')
endtry

* Now cleanup. This code won't execute because the error
* bubbles up.

if used('customer')
  messagebox('Closing customer table after ENDTRY...')
  use
endif used('customer')
```

Exception Object

VFP 8 includes a new Exception base class to provide an object-oriented means of passing information about errors around. As we saw earlier, Exception objects are created when you use TO *VarName* in CATCH commands. They're also created when you use the THROW command, which we'll discuss next.

Besides the usual properties, methods, and events (Init, Destroy, BaseClass, AddProperty, etc.), Exception has a set of properties, shown in Table1, containing information about an error. All of them are read-write at runtime.

Table 1. Properties of the new Exception base class.

Property	Type	Similar Function	Description
Details	Character	SYS(2018)	Additional information about the error (such as the name of a variable or file that doesn't exist); NULL if not applicable.
ErrorNo	Numeric	ERROR()	The error number.
LineContents	Character	MESSAGE(1)	The line of code that caused the error.
LineNo	Numeric	LINENO()	The line number.
Message	Character	MESSAGE()	The error message.
Procedure	Character	PROGRAM()	The procedure or method where the error occurred.
StackLevel	Numeric	ASTACKINFO()	The call stack level of the procedure.
UserValue	Variant	Not applicable	The expression specified in a THROW statement.

THROW It To Me

The last piece of structured error handling is the new THROW command. THROW is like the ERROR command in that it causes an error to be passed to an error handler, but it works quite differently too. Here's the syntax for this command:

```
throw [ uExpression ]
```

uExpression can be anything you wish, such as a message, a numeric value, or an Exception object.

If *uExpression* was specified, THROW creates an Exception object, sets its ErrorNo property to 2071, Message to "User Thrown Error", and UserValue to *uExpression*. If *uExpression* wasn't specified, the original Exception object (the one created when an error occurred) is used if it exists and a new Exception object is created if not. If either case, it then passes the Exception object to the next higher-level error handler (typically a TRY structure that wraps the TRY structure the THROW was called from within). Here's an example, taken from TestThrow.prg:

```
try
  try
    wait window xxx
  catch to loException when loException.ErrorNo = 1
    wait window 'Error #1'
  catch to loException when loException.ErrorNo = 2
    wait window 'Error #2'
  catch to loException
    throw loException
  endtry
catch to loException
  lnError   = loException.ErrorNo
  lcMessage = loException.Message
  messagebox('Error #' + transform(lnError) + chr(13) + ;
    'Message: ' + lcMessage, 0, 'Thrown Exception')
  lnError   = loException.UserValueErrorNo
  lcMessage = loException.UserValueMessage
  messagebox('Error #' + transform(lnError) + chr(13) + ;
    'Message: ' + lcMessage, 0, 'Original Exception')
endtry

* Now do the same thing, but use THROW without any
* expression

try
  try
    wait window xxx
  catch to loException when loException.ErrorNo = 1
    wait window 'Error #1'
  catch to loException when loException.ErrorNo = 2
    wait window 'Error #2'
  catch to loException
    throw
  endtry
catch to loException
  lnError   = loException.ErrorNo
  lcMessage = loException.Message
  messagebox('Error #' + transform(lnError) + chr(13) + ;
    'Message: ' + lcMessage, 0, 'Thrown Exception')
endtry
```

Although you might think the THROW loException command in this code throws loException to the next higher error handler, that isn't the case. THROW creates a new Exception object and throws that, putting loException into the UserValue property of the new object. Thus, the code in the outer TRY structure above shows that the Exception object it receives is about the user-thrown error. To retrieve information about the original error, you need to get the properties of the Exception object referenced by the UserValue property.

The second example in the code above shows that THROW by itself will rethrow the Exception object if there is one. In this case, we have the same object, so the outer error handler doesn't have to reference the UserValue property.

You can use a THROW statement outside a TRY structure, but it doesn't really provide any benefit over the ERROR command, since in either case, an error handler must be in place to catch it or the VFP error handler will be called. In fact, if something other than a TRY structure catches a THROW, it will likely be somewhat confused about what the real problem is, because only a TRY structure can catch the Exception object that's thrown. In the case of an Error method or ON ERROR routine, the parameters received and the results of AERROR() will be related to the THROW statement and the unhandled exception error rather than the reason the THROW was used. Some of the Exception object's properties are

placed into columns 2 and 3 of the array filled by AERROR(), so the error handler could parse those columns. However, that doesn't seem like the proper way to do things. Instead, make sure that THROW is only used when it can be caught by a TRY structure.

Trouble in Paradise

One of the biggest issues in error handling in VFP is preventing errors while in an error condition. By "error condition", I mean that an error has occurred, it has been trapped by the Error method of an object or an ON ERROR handler, and the handler hasn't issued a RETURN or RETRY yet. If anything goes wrong while your application is in an error condition, there is no safety net to catch you; instead, the user gets a VFP error dialog with a VFP error message and Cancel and Ignore buttons. That means your entire error handling mechanism must be the most bug-free part of your application, plus you have to test for things that may not be bugs but environmental issues.

For example, suppose your error handler logs an error to a table called ERRORLOG.DBF. Well, what happens if that file doesn't exist? You have to check for its existence using FILE() and create it if it doesn't. What is something else has it open exclusively? You could minimize that by never opening it exclusively, but to be absolutely safe, you should use FOPEN() first to see if you can open it, since FOPEN() returns an error code rather than raising an error. What if it exists and you can open it using FOPEN() but it's corrupted? You can't easily test for that, unfortunately.

See the problem? Your error handler can start becoming so complex by testing for anything that can possibly go wrong while in the error condition that you actually introduce bugs in this complex code!

In earlier versions of VFP, there was no solution to this problem. You just wrote some reasonable code, tested it as much as possible, and then hoped for the best. Fortunately, we have a solution in VFP 8: wrapping your error handler in a TRY structure. Because any errors that occur in the TRY block are caught by the CATCH blocks, we now have a safety net for our error handling code.

Here's a simple example (WrappedErrorHandler.prg in the download files):

```
on error do ErrHandler with error(), program(), lineno()
use MyBadTableName && this table doesn't exist
on error

procedure ErrHandler(tnError, tcMethod, tnLine)

* Log the error to the ErrorLog table.

try
  use ErrorLog
  insert into ErrorLog values (tnError, tcMethod, tnLine)
  use

* Ignore any problems in our handler.

catch
endtry
return
```

If the ErrorLog table doesn't exist, can't be opened, is corrupted, or for any other reason can't be used, the CATCH block will execute. In this case, there's no code, so the error within the error is ignored.

Error Handling Strategy

Let's tie all of this information together and talk about an overall error handling strategy. Here's the approach I now use:

- Use three layers of error handling: TRY structures for local error handling, Error methods for object-level, encapsulated error handling, and ON ERROR routines for global error handling. At the first two levels, handle all anticipated errors and pass unanticipated ones to the next level.
- Wrap your error handlers in TRY structures to prevent the VFP error dialog from appearing if something goes wrong in your error code.

- Use the Chain of Responsibility design pattern for your error handling mechanism. For details on this, see my January 1998 FoxTalk column, "Error Handling Revisited", or a white paper on error handling on the Technical Papers page at http://www.stonefield.com.
- Don't use a TRY structure to wrap your entire application. If you do, there's no way to stay within the application as soon as any error occurs.
- Don't use THROW unless you know a TRY structure will catch it.

Summary

The structured error handling features added in VFP 8 now provide us with three levels of error handling: local, object, and global. Structured error handling allows you to reduce the amount of code related to propagating error information, making your applications simpler, easier to read, and more maintainable. In addition, some thorny issues, such as one object inadvertently catching errors raised in another, are now neatly dealt with. I suggest you spend some time playing with structured error handling and consider how it will become a key part of your overall error handling strategy.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and co-author of "What's New in Visual FoxPro 8.0", "What's New in Visual FoxPro 7.0", and "The Hacker's Guide to Visual FoxPro 7.0", from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com