Error Handling in VFP 8

Doug Hennig
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Voice: 306-586-3341
Fax: 306-586-5080
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com

Overview

VFP 8 now 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 document examines structured error handling and show how it fits in with a complete error handling strategy.

Introduction

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 hole in this strategy is that if an object with code in its Error method calls procedural code (such as a PRG) or a method of an 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.)?

Here's an example that shows this (taken from ErrorOverridesONERROR.prg, included with the sample files accompanying this document):

```
loObject1 = createobject('Object1')
loObject1.Test()

define class Object1 as Custom
  procedure Test
    local loObject2
    loObject2 = createobject('Object2')
    loObject2.Test()
  endproc
  procedure Error(tnError, tcMethod, tnLine)
     wait window 'I have no idea what went wrong'
  endproc
enddefine

define class Object2 as Custom
  procedure Test
    local llError
    on error llError = .T.
    wait window xxx
    on error
    if llError
      wait window 'Object2 handled its own error'
    endif llError
  endproc
enddefine
```

When you run this code, you'll see that Object1's Error method was called rather than the error handler set up by Object2.

Here's a concrete example of the problems this can cause. I recently added a third party component to one of my applications. All of a sudden, errors started showing up whenever I called the component's methods. None of these problems showed up when I had tested the component in a test environment outside of the application (this reinforces the idea that you need to do system testing even after a component has been unit tested). When I traced the code, I found that the component used ON ERROR to trap errors rather than having code in its Error method. As a result, when an anticipated error occurred, my error handler was called rather than the component's. Since my error handler had no idea what went wrong, all it could do was log the problem and throw up a generic error message. When I moved the ON ERROR code to the Error method of the component, the component properly handled the problems and the error messages went away. However, note that I was only able to solve this problem because I had source code for the component and was willing to spend the time to track down the problem.

Another issue is that since we can't use a local ON ERROR handler in an object if we're using the Error method, that method must handle all errors in all methods of the object. That often leads to code in Error that looks like this:

```
lparameters tnError, tcMethod, tnLine
do case
  case tcMethod = 'OpenCursor' and tnError = 1
    * handle "file not found" in OpenCursor method
  case tcMethod = 'OpenCursor' and tnError = 15
    * handle "not a table" in OpenCursor
  case tcMethod = 'OutputResults' and tnError = 1102
    * handle "cannot create file" in OutputResults
  * more cases here
endcase
```

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.

CATCH Me if You Can

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 and evaluates it. 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). Here’s a typical example:

```
try
  * some code
catch to loException when loException.ErrorNo = SomeErrorNum
  * handle this type of error
catch to loException when loException.ErrorNo = AnotherErrorNum
  * handle this type of error
catch to loException
```

```
  * handle all other errors
endtry
```

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.

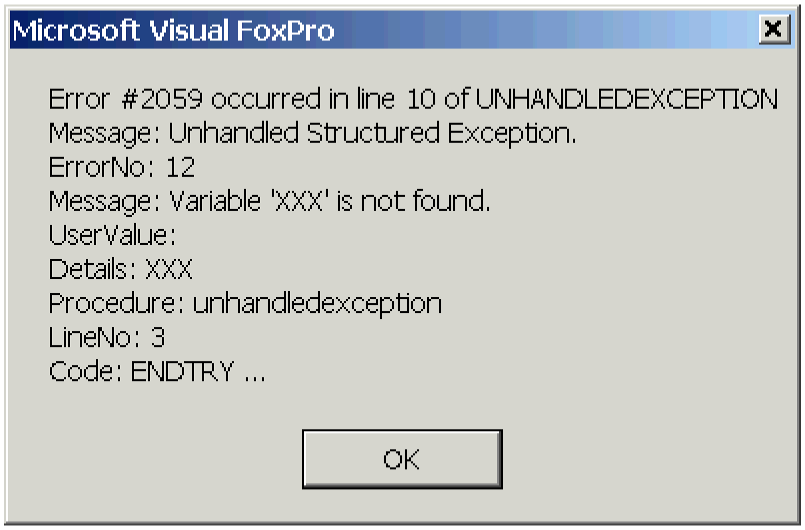
If VFP doesn't find a CATCH statement to use, then an unhandled exception error (error 2059) occurs. You don't really want that to happen for a variety of reasons, the biggest one being that the problem that caused the original error isn't handled because now we have a bigger mess on our hands.

Here's an example of what happens when you have an unhandled exception (taken from UnhandledException.prg):

```
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
  wait window '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))
```

When you run this code, you'll see the following error message:



```
Error #2059 occurred in line 10 of UNHANDLEDEXCEPTION
Message: Unhandled Structured Exception.
ErrorNo: 12
Message: Variable 'XXX' is not found.
UserValue:
Details: XXX
Procedure: unhandledexception
LineNo: 3
Code: ENDTRY ...
```

Notice that the error number, line number, error message, and code are all for the unhandled exception error, not the original error. Although the error message includes information about the original error, you'd have to parse it to obtain that error number and message.

Less is More

One of the benefits of structured error handling is that it allows you to reduce the amount of code related to handling and propagating errors. One source estimates that up to half of the code in an application is related to dealing with errors.

Here's some code that might be used to output the contents of a table to a Word document:

```
lcOnError = on('ERROR')
on error llError = .T.
llError = .F.
use MyTable
if not llError
  loWord = createobject('Word.Application')
  if not llError
* process the table, sending the results to Word via Automation
  else
* handle the error instantiating Word
  endif not llError
else
* handle the error opening the table
endif
on error &lcOnError
```

In VFP 8, this code can be reduced to:

```
try
  use MyTable
  loWord = createobject('Word.Application')
* process the table, sending the results to Word via Automation
catch
* handle the errors in CATCH blocks
endtry
```

Because it has fewer lines of code, fewer variables, and a simpler structure, this code is much easier to read and more maintainable.

You Can't Go Back

One important different 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 in the CATCH block (actually, if you use it anywhere in the structure), you’ll cause error 2060 to occur. RETRY is ignored (in my opinion, it should also raise error 2060); a bug in the current beta version also causes weird things to happen, like the routine to be exited completely or an endless loop within the CATCH block. 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 later on, errors can “bubble up” to the next highest error handler, and we may 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
  messagebox('Error #' + transform(loException.ErrorNo) + ' occurred.')
endtry
if used('customer')
  wait window 'Customer table is still open'
else
  wait window 'Customer table was closed'
endif used('customer')

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')
    wait window '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')
  wait window '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 containing information about an error. All of them are read-write at runtime.

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

StackLevel	Numeric	The call stack level of the procedure.	ASTACKINFO()
UserValue	Variant	The expression specified in a THROW statement.	Not applicable

THROW It To Me

VFP 3 introduced a command that was missing in earlier versions: ERROR. This command caused an error to be raised. When I first heard of this command, I thought, “My code has enough errors in it! Why would I want to purposely add some?” However, it turns out that this is a very useful command under the proper conditions.

Typically, you use ERROR to generate an error when a programmer bug that wouldn’t normally generate an error occurs. For example, suppose you have a function that expects three parameters but only two are passed. How do you signal to the calling code that it was called improperly? You could return an unusual value (such as -1, NULL, or an empty string), but that would likely lead to hard-to-find problems in the calling code. After all, if the programmer didn’t know enough to call the function properly, how likely are they to handle unusual return values properly? Instead, you can use ERROR 1229, which causes a “Too few arguments” error to occur, which is the same error that occurs if the programmer calls a native function with too few parameters. You can even specify a string instead of an error number, such as ERROR 'You are one bone-headed dude'. This causes error 1098 to occur, and the string can be retrieved with MESSAGE() or AERROR().

One thing ERROR isn’t good for, however, is passing an error from one error handler to another. For example, suppose you have code in the Error method of an object that can handle certain types of foreseen errors that may occur in the object, and the rest you want to pass to a global error handler (such as ON ERROR). You can’t use the ERROR command to do this because if an error occurs while VFP is in an error handler, VFP displays its own error dialog (we’ll discuss this in more detail later), which under no circumstances should a user ever see. So, you typically have to use code like this instead:

```
lcError = upper(on('ERROR'))
lcError = strtran(lcError, 'SYS(16)',   '"' + lcMethod + '"')
lcError = strtran(lcError, 'PROGRAM()', '"' + lcMethod + '"')
lcError = strtran(lcError, 'ERROR()',   'lnError')
lcError = strtran(lcError, 'LINENO()',  'lnLine')
* more STRTRAN statements here
&lcError
```

Notice the use of several STRTRAN() statements to substitute the information about the error into the ON ERROR call so the proper information is passed on. Isn’t this ugly?

Fortunately, VFP 8 provides a better way to deal with this: the THROW command. THROW is like ERROR 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
  messagebox('Error #' + transform(loException.ErrorNo) + ;
    chr(13) + 'Message: ' + loException.Message, 0, ;
    'Thrown Exception')
  messagebox('Error #' + ;
    transform(loException.UserValue.ErrorNo) + chr(13) + ;
    'Message: ' + loException.UserValue.Message, 0, ;
    'Original 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 always creates a new Exception object and throws that, putting *uExpression* 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.

Because I expect getting the “real” error information from an object in an Exception’s UserValue property to be a common thing to do, I created a subclass of Exception called SFException. Note that like some other base classes such as Session, Exception can only be subclassed in a PRG, not visually. SFException, which is defined in SFException.prg, uses Access methods on its properties so the properties of the object in UserValue are returned instead. That way, rather than using code like loException.UserValue.ErrorNo, which could blow up if something other than an Exception object was thrown, you simply use loException.ErrorNo. Here’s the code:

```
define class SFException as Exception
  protected oException
  oException = .NULL.

  procedure Init(toException)
    This.SetExceptionObject(toException)
  endproc
  procedure SetExceptionObject(toException)
    if vartype(toException) = 'O' and ;
      upper(toException.BaseClass) = 'EXCEPTION'
      This.oException = toException
    endif vartype(toException) = 'O' ...
  endproc

  procedure Details_Access
    local loException
    loException = This.GetExceptionObject()
    return loException.Details
  endproc
  procedure ErrorNo_Access
    local loException
    loException = This.GetExceptionObject()
    return loException.ErrorNo
  endproc
  procedure LineContents_Access
    local loException
    loException = This.GetExceptionObject()
    return loException.LineContents
  endproc
  procedure LineNo_Access
    local loException
    loException = This.GetExceptionObject()
    return loException.LineNo
  endproc
  procedure Message_Access
    local loException
    loException = This.GetExceptionObject()
    return loException.Message
  endproc
  procedure Procedure_Access
    local loException
    loException = This.GetExceptionObject()
    return loException.Procedure
  endproc
  procedure StackLevel_Access
    local loException
    loException = This.GetExceptionObject()
    return loException.StackLevel
  endproc

  protected function GetExceptionObject
    local loException
    do case
      case vartype(This.oException) <> 'O'
        loException = This
      case vartype(This.oException.UserValue) = 'O' and ;
        upper(This.oException.BaseClass) = 'EXCEPTION'
        loException = This.oException.UserValue
      otherwise
        loException = This.oException
    endcase
    return loException
  endfunc
enddefine
```

To use this class, pass the received Exception object when you instantiate SFException (or pass the object to its SetExceptionObject method instead). Here's an example (TestSFException.prg) that shows how it can be used. The second MESSAGEBOX() shows the information we want.

```
local loException as Exception, ;
  loExObject as Exception
try
  try
    wait window xxx
  catch to loException
    throw loException
  endtry
catch to loException
  messagebox('Error #' + transform(loException.ErrorNo) + ;
    ' occurred in line ' + transform(loException.LineNo) + ;
    ' of ' + loException.Procedure + chr(13) + ;
    'Contents: ' + loException.LineContents + chr(13) + ;
    'Message: ' + loException.Message, 0, 'Thrown Exception')
  loExObject = newobject('SFException', 'SFException.prg', '', ;
    loException)
  messagebox('Error #' + transform(loExObject.ErrorNo) + ;
    ' occurred in line ' + transform(loExObject.LineNo) + ;
    ' of ' + loExObject.Procedure + chr(13) + ;
    'Contents: ' + loExObject.LineContents + chr(13) + ;
    'Message: ' + loExObject.Message, 0, 'Original Exception')
endtry
```

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. The SYS(2410) function, which we'll see later, could help with this.

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. Bad news! 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 ... CATCH ... ENDTRY statement. 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 (albeit contrived) example (WrappedErrorHandler.prg in the sample files):

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

procedure ErrHandler(tnError, tcMethod, tnLine)
local loException

* 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.

At this point, a twisted mind would probably ask "What happens if there's an error in the CATCH block of an error handler?" There are four places where an error could occur: in the CATCH statement, in any function or method called by the CATCH statement, in the CATCH block, or in any function or method called by the CATCH block. Here's an example showing the first two cases (taken from BadCatch.prg):

```
loObject = createobject('MyClass')
try
```

```
  wait window xxx
* Error in CATCH statement (ErrorNo is numeric): skipped
catch to loException when loException.ErrorNo = '12'
* Error in CATCH statement (DoWeHandleError returns string): skipped
catch to loException when DoWeHandleError()
* Error in procedural code called by CATCH statement: skipped
catch to loException when AnotherHandler()
* Error in method called by CATCH statement: object.Error handles it
catch to loException when loObject.Handler()
catch
  wait window 'Final catch'
endtry

function DoWeHandleError
return 'Yes'

function AnotherHandler
wait window xxx

define class MyClass as Custom
  function Handler
    wait window xxx
  endfunc
  procedure Error(tnError, tcMethod, tnLine)
    wait window 'MyClass.Error'
  endproc
enddefine
```

This code shows a couple of things:

- An error in a CATCH statement, in procedural code called from a CATCH statement, or in the method of an object lacking code in its Error method that's called from a CATCH statement is "eaten" and the CATCH statement is ignored. Execution continues with the next CATCH statement.

- An error in the method of an object with code in its Error method that's called from a CATCH statement is handled by the object's Error method. Notice that the original error isn't actually dealt with in this case (at least in the current version of the VFP 8 beta; this behavior may change in the final release).

The next example showing the second two cases, errors in the CATCH block (also taken from BadCatch.prg):

```
try
  wait window xxx
* Error in CATCH block (ErrorNo is numeric)
catch to loException when loException.ErrorNo = 12
  wait window loException.ErrorNo + ' ' + loException.Message
* You might think this would catch the error in the above CATCH, but it
doesn't
catch to loException
  wait window transform(loException.ErrorNo) + ' ' + loException.Message
finally
  wait window 'Finally'
endtry
```

```
* Show error in procedural code called from CATCH block

try
  wait window xxx
* Error in CATCH block (ErrorNo is numeric)
catch to loException
  AnotherHandler()
endtry

* Show error in an object method called from CATCH block

try
  wait window xxx
* Error in CATCH block (ErrorNo is numeric)
catch to loException
  loObject.Handler()
endtry

* Now show an outer TRY.

try
  try
    wait window xxx
  catch to loException when loException.ErrorNo = 12
    wait window loException.ErrorNo + ' ' + loException.Message
  finally
    wait window 'Finally'
  endtry
* Gotcha!
catch to loException
  wait window transform(loException.ErrorNo) + ' ' + loException.Message
endtry
```

We can see a couple of things from this example:

- An error in a CATCH block, in procedural code called from a CATCH block, or in the method of an object lacking code in its Error method that's called from a CATCH block bubbles up to the next level of error handler, such as the VFP error handler (shown in this case), the Error method of an object (if the TRY was performed within a method of the object), an ON ERROR handler, or an outer TRY (also shown in this example).

- An error in the method of an object with code in its Error method that's called from a CATCH statement is handled by the object's Error method. As with error in CATCH statements, the original error isn't dealt with in the current beta version.

So, what does all this mean? First, wrap your error handling code in a TRY ... CATCH ... ENDTRY statement to ensure that any errors that occur won't cause a VFP error dialog to appear. Second, if you're doing anything at all in the CATCH statements or blocks (in other words, not just an empty CATCH statement with no code in the block), wrap the TRY ... CATCH ... ENDTRY statement in an outer TRY ... CATCH ... ENDTRY. Here's an example:

```
procedure Error(tnError, tcMethod, tnLine)
try
  try
    * error handling code goes here
  catch to loException
    * handle an error in the error handling code
  endtry
* This is the error handler of the error handler of the
* error handler (does your brain hurt yet? <g>): we'll just
* ignore any errors
catch
endtry
```

Note that the outer CATCH statement and block should be empty, or you'll have to use yet another layer of TRY ... CATCH ... ENDTRY around it; that would get ridiculous very quickly!

Error Handling Strategy

*** NEWOPENTABLE.PRG vs. OLDOPENTABLE.PRG, TESTOPENTABLE.PRG, ERRORDEMOS.SCX, STARTUP1/2.PRG

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 white paper on error handling, available 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.

What Else Ya Got?

There are a few other changes in VFP 8 related to error handling.

First, the new SYS(2410) function returns a numeric value that indicates how an error will be handled at this point in the code. The following are the possible values:

Value	Description
0	VFP's error handler

1	TRY structure
2	Error event
3	ON ERROR

Note that this function isn't perfect. For example, if it's called from within a TRY structure that contains no CATCH blocks that will be executed in the case of an error, SYS(2410) will still return 1.

One use of SYS(2410) is to determine if a THROW statement can be used or not. Since THROW works best if caught within a TRY structure, and you may not know if the next highest error handler is a TRY structure or not (in the case of a function that could be called from anywhere), you could use code like the following:

```
if sys(2410) = 1
  throw 'Error occurred'
else
  error 'Error occurred'
endif sys(2410) = 1
```

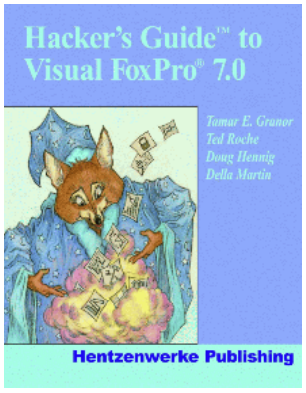
The second improvement is that VFP now handles errors in two places that were previously untrappable: in reports and when tables have invalid DBC backlinks. This is a welcome change because untrappable errors always display the VFP error dialog to the user.

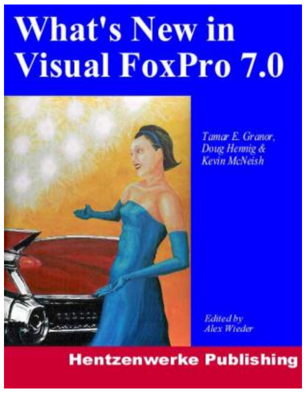
Summary

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.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and co-author of the award-winning Stonefield Query. He is co-author (along with Tamar Granor, Ted Roche, and Della Martin) of "The Hacker's Guide to Visual FoxPro 7.0" and co-author (along with Tamar Granor and Kevin McNeish) of "What's New in Visual FoxPro 7.0", both from Hentzenwerke Publishing, and author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's Pros Talk Visual FoxPro series. He writes the monthly "Reusable Tools" column in FoxTalk. He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals", both 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).




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Edited by Alex Wieder
Hentzenwerke Publishing

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341 Fax: (306) 586-5080
Email: dhennig@stonefield.com
Web: www.stonefield.com