Try/Catch/Finally in Visual FoxPro 8

Hector J. Correa
hector@hectorcorrea.com
www.hectorcorrea.com

Overview

Visual FoxPro 8 provides a new control structure, Try/Catch/Finally, for exception handling. Exception handling is a different paradigm from traditional error handling mechanisms like On Error and Error Method. This document provides a guide on how to take advantage of Try/Catch/Finally to write cleaner code, and to separate code that performs functionality features from exception handling code.

What is Try/Catch/Finally?

Try/Catch/Finally is a new *control structure* in Visual FoxPro 8 that allows you to handle errors and exceptions in your applications.

Control structures are commands that affect the execution path of the code at runtime. For example, Do While/Enddo is a control structure to create loops. Likewise, Do Case/Endcase is a command structure that allows branching at runtime. Try/Catch is a control structure geared towards error/exception handling.

Let's see how Try/Catch works. Suppose you have the following code to open a table. Note that we are deliberately trying to open a table that does not exist.

```
? "hello world"
nMyVar = 0
TRY
      ? "executing code"
      nMyVar = 1
      USE anonexistingtable
      nMyVar = 2
      nMyVar = 3
CATCH
      ? "uh-oh: something went wrong"
ENDTRY
? "nMyVar = ", nMyVar
? "goodbye"
```

If you run this code, VFP will print the following messages on the screen:

```
hello world
executing code
uh-oh: something went wrong
nMyVar = 1
goodbye
```

Notice that the printed value of variable `nMyVar` is one, not three. Why is that? The reason for this behavior is because, when an error (or more exactly an exception) is thrown, VFP jumps to the nearest Catch that can handle the exception. In our example, attempting to use a table that does not exist causes an exception. At this point, VFP jumps to the catch statement and skips the lines of code between the code that caused the exception and the catch statement.

What is an Exception?

An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions[1].

[1] http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html

Traditionally, when an error occurs in your code (e.g. attempting to open a table that does not exist) VFP generates an *error*. However, when the code that causes the error is inside a Try/Catch block, VFP generates an *exception* instead.

There is a new class in VFP that is called Exception class. All exceptions automatically generated by VFP are instances of this Exception class.

You can catch these exceptions via the Catch statement of the Try/Catch control structure. Let's look at the following code:

```
? "hello world"
TRY
      ? "executing code"
      USE anonexistingtable
      nMyVar = 2
CATCH TO oEx
      ? "uh-oh: something went wrong"
      ? oEx.ErrorNo, oEx.Message
ENDTRY
? "goodbye"
```

In this example, the exception that VFP generates when it cannot find the table is caught in the Catch statement and assigned to the `oEx` variable. Notice that the code inside the Catch uses properties `ErrorNo` and `Message` of the Exception class to display more information about the exception generated.

Error Handling versus Exception Handling

As mentioned above, Try/Catch is a control structure for exception handling. But, what is the difference between an *error handling* and an *exception handling*? The short answer is that exception handling unwinds the stack, while error handling does not[2].

In traditional error handling, like the one achieved with On Error command and the Error method, when an error is detected VFP executes code to handle the error but, immediately after that, VFP continues execution in the line after the one that caused the error. The following sample code shows this:

```
lError = .F.
ON ERROR lError = .T.
USE anonexistingtable
IF lError
      ? "something went wrong"
ELSE
      ? "so far so good"
ENDIF
```

In this example, when an error occurs, VFP sets variable `lError` to true and continues execution on the *next* line of code.

[2] Abrahams, David. 2001-2003. http://www.boost.org/more/error_handling.html

Exception handling on the other hand unwinds the stack, which means that the execution of the code does not continue on the next line after the one that caused the exception. Once an exception is thrown, the execution will *jump* to the catch statement that can handle the exception. The code immediately after the after line that caused the exception will not be executed. The following example shows this. Notice that the line **`? "so far so good"`** will not be executed.

```
TRY
      USE anonexistingtable
      ? "so far so good"
CATCH
      ? "something went wrong"
ENDTRY
```

Be aware that the catch statement that can handle the exception might not be on the same procedure that throws the exception. It might be on any of the programs in the stack that were called before reaching the line of code that threw the exception. We'll talk more about this in the following sections.

Try/Catch – The Fundamental Idea

Although the difference between error handling and exception handling might seem subtle at first sight, there are some two significant advantages that Try/Catch brings to the VFP developer.

The first advantage is the *separation of responsibilities* that could be achieved with Try/Catch. Code structured with Try/Catch is usually cleaner than code written with On Error. This is due to the fact that the developer does not need to spread the exception handling code all over the place. With Try/Catch exception handling code can be confined to the Catch block.

The following code shows how much cleaner code looks with Try/Catch when compared to a traditional On Error implementation. Which one would you rather maintain?

ON ERROR Sample

```
LOCAL lError
ON ERROR lError = .T.

CD c:\temp
IF not lError
  USE mytable EXCLUSIVE
  IF not lError
    PACK
    COPY TO myothertable
  ENDIF
ENDIF

IF lError
   ? "something went wrong"
ENDIF
```

Try/Catch Sample

```
TRY
  CD c:\temp
  USE mytable EXCLUSIVE
  PACK
  COPY TO myothertable
CATCH
  ? "something went wrong"
ENDTRY
```

This separation of responsibilities is even more visible when writing routines that will be reused across several systems or across several pieces of the same application.

For example, it is common for the developer of a routine to be aware that the routine could cause some exceptions (e.g. the routine might not be able to open a table exclusively.) However, this developer might not know what to do with these errors. Should his routine display the error on the screen? Should the routine log the error to a text file? Should an administrator be notified via e-mail about the error? On the other hand, the developer that will use the routine might know what to do about the exceptions, but she does not have a way to detect them, she can only react to them.

The following code shows an example of this situation. In this particular example the called routine might fail because `mytable` cannot be found or cannot be opened exclusively. Yet, the routine does not handle the error, it merely throws an exception. The caller program will catch this exception and advise the user to notify technical support department. Another caller program might implement a different exception handling mechanism and log the error in a text file instead.

Caller Program

```
TRY
   DO MyBackup
CATCH
   ? "backup failed"
   ? "contact tech support"
ENDTRY
```

Called Routine

```
PROCEDURE MyBackup
   USE mytable EXCLUSIVE
   PACK
   COPY TO myothertable
RETURN
```

Another example where this separation of responsibilities is very beneficial is on n-tier applications. On this type of applications, it is often the case that the business tier does part of the exception handling and then notifies the caller presentation tier (either a thin or a fat client) than something went wrong. The presentation tier in a fat client application might use a `MessageBox` to let the user know about the problem, while a thin client application might render a new web page.

The second advantage that we get with exception handling is that now we have the *assurance* that when an exception is detected VFP will automatically jump to an exception handling code. In other words, the compiler is forcing exception handling rather than leaving it up the developer's discretion. How many times have you run into errors that could have been avoided had the developer evaluated the error code returned by a function in the previous lines? With Try/Catch this overlooking of errors is reduced dramatically because VFP will automatically jump to an exception handling block as soon as an exception is detected.

Catching Exceptions

In our last example, `MyBackup` procedure can generate one of many exceptions. For example, perhaps `mytable` cannot be found and therefore the `USE` command will fail. Or perhaps the table is found but it cannot be opened exclusively.

The course of action to take in the caller program will likely depend on the type of exception generated. A file-not-found exception might be a serious exception and might require us to e-mail an administrator telling him about the problem. If on the other side the exception is that the file cannot be opened exclusively perhaps we can take a less dramatic action and tell the user to try again at a later time.

The following code shows how this could be achieved by using multiple catch statements.

Caller Program	**Called Routine**
`TRY`	`PROCEDURE MyBackup`
`   DO MyBackup`	`   USE mytable EXCLUSIVE`
`CATCH TO oEx WHEN oEx.ErrorNo=1`	`   PACK`
`   ? "backup failed"`	`   COPY TO myothertable`
`   ? "MyTable was not found"`	`RETURN`
`   ? "contact tech support"`	
`CATCH TO oEx WHEN oEx.ErrorNo=3`	
`   ? "backup failed"`	
`   ? "file is still in use"`	
`   ? "try again later"`	
`CATCH TO oEx`	
`   ? "backup failed"`	
`   ? oEx.Message`	
`ENDTRY`	

Catch statements are evaluated akin to Case statements in a Do Case structure. VFP will evaluate each Catch individually until the When condition is true. Once a When condition is true, VFP will skip the other Catch statements. A Catch without a When statement works as a catch-all statement and it is equivalent to using When .T.

The Finally Clause

There is an extra statement that can be used with Try/Catch. The Finally clause. Finally is typically used to code actions that need to be taken to clean up the environment or restore it to a previous state.

It is very important to note that the code in the Finally section will always be executed, regardless of whether an exception is thrown or not.

The following code shows an example on how to use Finally clause to ensure that our backup routine returns to the original folder.

Caller Program

```
? “before calling back up”
TRY
   DO MyBackup
CATCH
   ? “something went wrong”
ENDTRY
? “after calling back up”
```

Called Routine

```
PROCEDUDE MyBackup
   ? “about to start backing up...”
   TRY
      cOldFolder = curdir()
      CD C:\temp
      USE mytable EXCLUSIVE
      PACK
      COPY TO myothertable
   FINALLY
      CD (cOldFolder)
      IF used( “myTable” )
         USE IN mytable
      ENDIF
   ENDTRY
   ? “we are done backing up”
RETURN
```

Notice that there is no Catch statement in the back up routine. We are assuming that the caller program will catch the exceptions, and it will since it has a Catch-all exceptions statement.

A common misunderstanding is that the Finally clause is unnecessary because code inside it is always executed. Some people get confused and think that code written after the Endtry has the same scope as code in the Finally block. This is not true. Let’s see why not.

Assume that `mytable` cannot be opened exclusively in our previous example. At that point VFP will throw an exception and look for a Catch statement in the current procedure. Since there is no catch statement in our example, VFP will execute the code in the Finally block and then it will unwind the stack looking for a catch statement that can handle the exception. VFP will find the catch statement in the caller program and will continue executing the caller program after that. Notice that VFP never executed the line that prints the `“we are done backing up”` message in the Backup routine.

Nesting Try/Catch Structures

Try/Catch structures can be nested. This allows you to catch very specific exceptions at lower levels of the stack and other more general errors at upper levels.

The following code shows an example on how this work. This code runs a routine called `Procedure1` which in turn calls another routine called `Procedure2` which in turn calls the `MyBackup procedure`. For the sake of an example, let's assume that `MyBackup` routine has the same code as the one used in our last example.

Notice that `Procedure2` knows how to handle exceptions where the `ErrorNo` is equal to 1 (file not found) or 3 (file cannot be open for exclusive use.) If the backup routine generates any other type of exception (e.g. `ErrorNo` 202, invalid path) then `Procedure2` will not be able to handle it. When this happens, VFP will continue unwinding the stack and find a handler for the exception in `Procedure1` (remember that a catch statement without a When condition works as a catch-all.)

```
PROCEDURE Procedure1
      TRY
            DO Procedure2
      CATCH TO oEx
            ? "Something went wrong and we could"
            ? "not handle it in other CATCH statements"
      ENDTRY
RETURN

PROCEDURE Procedure2
      TRY
            DO MyBackUp.prg
      CATCH TO oEx WHEN oEx.ErrorNo=1
            ? "File to backup does not exist"
      CATCH TO oEx WHEN oEx.ErrorNo=3
            ? "File to backup could not be opened"
            ? "for exclusive use."
      ENDTRY
RETURN
```

Throwing Your Own Exceptions

In addition to the exceptions thrown by VFP, you can throw your own exceptions at any given point. To throw an exception, you just need to use the Throw command.

Throwing exceptions allows you to take advantage of structured exception handling for conditions that cannot be detected by VFP per se. For example, let's say you want to prevent users from saving employee records without a last name. The fact that the employee's last name is empty is not an exception that VFP can detect[3]. Nevertheless,

[3] Unless you have a field rule at the database level. For the sake of simplicity we will assume that such rule does not exist in our Employee table.

you can evaluate this condition yourself and throw an exception with the Throw command.

The following code shows how the THROW command can be used in a typical save routine. Notice that the value passed in the THROW command is retrieved via the `oEx.UserValue` property.

Save routine with traditional error handling	**Save routine using structured exception handling and the THROW command.**
`cError = “”`	`TRY`
`IF SaveActionBeforeSave()`	`   SaveActionBeforeSave()`
`   IF SaveAction()`	`   SaveAction()`
`      IF SaveActionAfter()`	`   SaveActionAfter()`
`      ENDIF`	`CATCH TO oEx`
`   ENDIF`	`   ? “Save process failed”`
`ENDIF`	**`   ? oEx.UserValue`**
`IF not empty( cError )`	`ENDTRY`
`   ? “Save process failed”`	
`   ? cError`	`RETURN`
`ENDIF`	
`RETURN`	
`PROCEDURE SaveActionBeforeSave()`	`PROCEDURE SaveActionBeforeSave()`
`   lRetVal = .T.`	`   IF empty( Employee.LastName )`
`   IF empty( Employee.LastName )`	**`      THROW “Empty last name”`**
`      cError = “Empty last name”`	`   ENDIF`
`      lRetVal = .T.`	`RETURN`
`   ENDIF`	
`RETURN lRetVal`	

Has Try/Catch Been Tried Before?

Although Try/Catch is new in VFP 8, Try/Catch has been available in other programming languages for quite a while.

C++ introduced Try/Catch in the late eighties. Other languages that currently support Try/Catch include Java, Delphi, VB.NET, and C#. In all these years, and across all these languages, Try/Catch has proven to be an excellent mechanism to handle exceptions.

Go ahead; don’t be afraid to give it a try.

Recommended Readings

- Castaño, Antonio. VFP 8: Try/Catch and Exceptions. Universal Thread Magazine. October 2002. http://www.universalthread.com/Magazine/October2002/Page61.asp

- Hening, Doug. CATCH Me if you Can. FoxTalk. January 2003. http://www.foxtalknewsletter.com
- MacNeill, Andrew. FoxPro Advisor. April 2003. http://foxproadvisor.com