Structured Exception Handling Sample Help

Sample Description: Structured Exception Handling

Points of Interest

- Nested Throw
- Multiple Catch
- Nested Catch/ReThrow
- User Exceptions

Control

For Help on Help, Press F1

Structured Exception Handling

Envelop provides a strong error handling system. This error handling is more structured than Visual Basic's "On Error" protocol. Objects defined by Envelop have as part of their characteristics, like properties and methods, the exceptions that they throw. Exceptions are like function calls which, after being thrown, ripple up the call stack until a matching catch block is encountered.

This sample highlights how to handle system exceptions, how to catch more than one exception and handle each one independently, how to throw a user defined exception, and how to rethrow an exception that has been caught.

Structured exceptions are more powerful because:

1. Developers may define error conditions of their own, an important facet of building first class objects.
2. Error conditions raised within a method can be handled by any method in the call stack of the erring method, increasing flexibility and promoting greater structure.

Nested Throw

In the Nested Throw example, the following method is executed when the "Test 1" button is clicked.

```
Sub TestThrow
    Try
        ThrowPoint1
    Catch TestThrowError(error_location As String)
        InfoBox.Message("", "I was thrown from '" & error_location & "'")
    End Try
End Sub
```

A Try/Catch is set up to handle an exception. A method ThrowPoint1 is called. If the option button beside "Sub ThrowPoint1" is clicked on, the following throw is executed.

```
Throw TestThrowError("ThrowPoint1")
```

The throw is then caught by the TestThrow method's Catch block, which presents an InfoBox with the name of the place where the throw occurred.

This sample works by having the main routine, TestThrow, call methods ThrowPoint1, ThrowPoint2, and ThrowPoint3. Each method makes a further nested call to another routine. The option button that is clicked, determines which method the Throw is initiated from. The Catch block is capable of recognizing the exact method the Throw occurred from, based on the message passed with the Throw.

Multiple Catch

The Multiple Catch example shows how more than one exception may be thrown and handled independently of each other. In this example, a method named GenerateSystemException is called to generate a random system exception. Based on which system exception is generated and Thrown, a corresponding Catch is executed to handle the exception.

```
Sub TestSystemException
    ' Try a function which will generate a system exception
    ' and in the catch block you would implement code to recover
    ' from the situation.
    Try
      GenerateSystemException
    Catch TooFewArguments
      LblTooFewArguments.ForeColor = 255
      InfoBox.Message("", "Too few arguments were submitted.")
    Catch TooManyArguments
      LblTooManyArguments.ForeColor = 255
      InfoBox.Message("", "Too many arguments were submitted.")
    Catch NotFound(error_description as String)
      LblNotFound.ForeColor = 255
      InfoBox.Message("", "The test function cannot find '" & error_description & "'")
    Catch FileError
      LblFileNotFound.ForeColor = 255
      InfoBox.Message("", "The file was not found.")
    End Try
End Sub
```

Notice how some of the Catch blocks may receive descriptive information from the Throw exception.

Nested Catch/ReThrow

In structured exception handling, you can Catch exceptions at any point within an execution sequence. In some cases, you may wish to catch the exception very close to a specific function call. If a Throw is detected at that point, you may wish to handle the exception at that point and then run the function a second time.

The following is an example of a nested catch. Click "Test 3" to run the following TestReThrow method.

```
Sub TestReThrow
    ' This illustrates how various routines can catch certain exceptions
    ' and handle them separately. Exceptions not caught go up the stack
    ' for further evaluation and handling.
    Try
      CatchPoint1
    Catch
      ' Catch any exception that was not previously caught and rethrow
      ' the exception. The only exception that can reach this
      ' point is the unhandled exception for a file not being found
      LblTestReThrow.ForeColor = 255
      Throw
    End Try
End Sub
```

This method calls a series of methods named CatchPoint1, CatchPoint2, and CatchPoint3. The CheckPoint3 method calls GenerateSystemException to generate a random system exception Throw. Each CatchPoint method has one specific Catch block to catch a specific type of Throw.

```
Sub CatchPoint1
    Try
      CatchPoint2
    Catch TooFewArguments
      LblCatchPoint1.ForeColor = 255
      InfoBox.Message("", "CatchPoint1: There were too few arguments.")
    End Try
End Sub
```

There is one exception that does not have a corresponding Catch block. This is an exception where the system cannot find the file specified. If an exception is not caught by one of the nested Catch blocks, the exception may be rethrown up the call stack and left unhandled. This generally results in an error dialog being displayed.

User Exceptions

User exceptions can be generated easily in Envelop. Like system exceptions, user exceptions can be used to check application data or user input. The method GenerateUserException shown below is an example of how several types of user exceptions may be generated. Like system exceptions, user exceptions may be caught in Catch blocks and handled independently.

```
Sub GenerateUserException
  Dim random_number as Integer
  random_number = int(4 * rnd() + 1)
  Select Case random_number
    Case 1
      ' This number is too small
      Throw NumberTooSmall(1)
    Case 2
      ' This number is even
      ' No arguements are being passed back
      Throw NumberEven
    Case 3
      ' This number is odd
      ' We throw the an execption titled "NumberOdd"
      ' with several arguments of different data types
      Throw NumberOdd(3.0, "$5.75")
    Case 4
      ' This number is too large
      Throw NumberTooLarge("4")
  End Select
End Sub
```

Feedback data may also be passed in the Throw statement of user exceptions. This information can be used to help a user understand the nature of a problem they are experiencing. In the example shown above, a random number between 1-4 is generated in order to randomly Throw a corresponding user exception.