Unit 10 Exception Handling in VB.NET

Structure:

10.0 Objectives:

After studying this unit, you will be able to:

- describe and discuss the importance of error handling mechanism in Visual Basic
- discuss various exceptions in VB.Net
- list and discuss Exception handling keywords
- explain the keywords try, catch, throw and finally

explain the properties of Exceptions

10.1 Introduction

All programming languages should have the provision to handle unexpected conditions. In programming languages before Microsoft .NET, this was named as ***error handling***. Unexpected conditions generated error codes, which were confined by programming logic that took appropriate action. The CLR (Common Language Runtime) in .NET does not generate error codes. When an unexpected condition occurs, the CLR creates a special object called an ***exception***. This object contains properties and methods that describe the unexpected condition in detail and communicate various items of useful information about what went wrong.

Because .NET deals with exceptions instead of errors, the term “error handling” is occasionally used in the .NET world. Instead, the term
“exception handling” is preferred. This term refers to the techniques used in
.NET to detect exceptions and take appropriate action.

This unit covers how exception handling works in Visual Basic 2005. There are many improvements over pre-.NET versions of Visual Basic. This unit discusses the common language runtime (CLR) exception handler in detail and the programming methods that are most efficient in catching errors.

10.2 Introduction to Error Handling in VB6

For compatibility, Visual Basic 2005 and other .NET versions of Visual Basic still support the old-style syntax for error handling that was used in Visual Basic 6 and earlier versions. That means you can still use the syntax presented in this review. However, it is strongly recommended that you avoid using this old-style syntax in favor of the exception handling features that are native to .NET. Using the new Try...Catch syntax (presented after this review) will give you more flexibility and better code structure.

The old-style syntax in VB6 was handed down from DOS versions of BASIC. The On Error construct was created in an era when line labels and GoTo statements were commonly used. Such error handling is difficult to use and has limited functionality compared to more modern alternatives.

In VB6, a typical routine with error handling code looks like this:

```
Private Function OpenFile(sFileName As String) As Boolean On Error GoTo
ErrHandler:
Open sFileName For Random As #1 OpenFile = True
Exit Sub ErrHandler:
Select Case Err.Number Case 53 ' File not
   found
      MessageBox.Show "File not found" Case Else
       MessageBox.Show "Other error" End Select
OpenFile = False End Function
```

The top of the routine points to a section of code called an *error handler,* which is usually placed at the bottom of the routine. The error handler takes control as soon as an error is detected in the routine, and it checks the error number to determine what action to take. The error number is available as a property of the Err object, which is a globally available object that holds error information in VB6. There are several other error handling syntax options not included in the preceding error-handling code. If the error handler can take care of the error without breaking execution, then it can resume execution with the line of code that generated the error (Resume), the one after that (Resume Next), or at a particular location (Resume {LineLabel}).

There's a much better way to manage errors in VB 2005, called *structured exception handling*. The rest of this unit discusses this new way to work with code errors, and uses the term structured exception handling throughout.

10.3 Exceptions in .NET

.NET implements a system wide, comprehensive approach to exception handling. As noted in the chapter introduction, instead of an *error number*, there is an *exception object*. This object contains information relevant to the error, exposed as properties of the object. Later you'll see a summary of the properties and the information they expose in a table. Such an object is an instance of a class that derives from a class named System. Exception. As shown later, a variety of subclasses of System. Exception are used for different circumstances.

Important Properties and Methods of an Exception

The Exception class has properties that contain useful information about the exception, as shown in the following table 10.1:

Table 10.1: Exception property

Property	Description
HelpLink	A string indicating the link to help for this exception
InnerException	Returns the exception object reference to an inner (nested) exception
Message	A string that contains a description of the error, suitable for displaying to users
Source	A string containing the name of an object that generated the error
StackTrace	A read-only property that holds the stack trace as a text string. The stack trace is a list of the pending method calls at the point at which the exception was detected. That is, if MethodA called MethodB, and an exception occurred in MethodB, the stack trace would contain both MethodA and MethodB.
TargetSite	A read-only string property that holds the method that threw the exception

The two most important methods of the Exception class are shown in table 10.2

Table 10.2: Methods of exception class

Method	Description
GetBaseException	Returns the first exception in the chain
ToString	Returns the error string, which might include as much information as the error message, the inner exceptions, and the stack trace, depending on the error

How Exceptions Differ from the Err Object in VB6

Since an exception contains all of the information needed about an error, structured exception handling does not use error numbers and the Err object. The exception object contains all the relevant information about the error. However, whereas there is only one global Err object in VB6, there are many types of exception objects in VB 2005. For example, if a divide by zero is done in code, then an OverflowException is generated. There are several dozen types of exception classes in VB 2005, and in addition to using the ones that are available in the .NET Framework, you can inherit from a class called ApplicationException and then create your own exception classes.

In .NET, all exceptions inherit from System.Exception. Special-purpose exception classes can be found in many namespaces. The following table 10.3 lists four representative examples of the classes that extend Exception:

Table 10.3: Exception classes

Namespace	Class	Description
System	InvalidOperation Exception	Generated when a call to an object method is inappropriate because of the object's state
System	OutOfMemory Exception	Results when there is not enough memory to carry out an operation
System.XML	XmlException	Often caused by an attempt to read invalid XML
System.Data	DataException	Represents errors in ADO.NET components

It is common for an exception class to reside in a namespace with the classes that typically generate the exception. For example, the DataException class is in System.Data, with the ADO.NET components that often generate a DataException instance. Having many types of exceptions in VB 2005 enables different types of conditions to be trapped with different exception handlers.

10.4 Structured-Exception-Handling Keywords

Structured exception handling depends on several new keywords in VB 2005:

- **Try** – Begins a section of code in which an exception might be generated from a code error. This section of code is often called a Try block. In some respects, this would be the equivalent of an On Error statement in VB6. However, unlike an On Error statement, a Try statement does not indicate where a trapped exception should be routed. Instead, the exception is automatically routed to a Catch statement.

- **Catch** – Begins an exception handler for a type of exception. One or more Catch code blocks follow a Try block, with each Catch block catching a different type of exception. When an exception is encountered in the Try block, the first Catch block that matches that type of exception receives control. A Catch statement is analogous to the line label used in a VB6 On Error statement, but the ability to route different types of exceptions to different Catch statements is a radical improvement over VB6.
- **Finally** – Contains code that runs when the Try block finishes normally, or when a Catch block receives control and then finishes. That is, the code in the Finally block always runs, regardless of whether an exception was detected. Typically, the Finally block is used to close or dispose of any resources, such as database connections, that might have been left unresolved by the code that had a problem. There is no equivalent of a Finally in VB6.
- **Throw** – Generates an exception. This is similar to Err.Raise in VB6. It's usually done in a Catch block when the exception should be kicked back to a calling routine or in a routine that has itself detected an error such as a bad argument passed in.

The Try, Catch, and Finally Keywords

Here is an example showing some typical simple structured exception handling code in VB 2005. In this case, the most likely source of an error is the items argument. If it has a value of zero, then this would lead to dividing by zero, which would generate an exception.

First, create a Windows Application in Visual Basic 2005 and place a button on the default Form1 created in the project. In the button's click event, place the following two lines of code:

```
Dim sngAvg As Single sngAvg = GetAverage(0,
100)
```

Then put the following function in the form's code:

```
Private Function GetAverage(iItems As Integer, iTotal As Integer) as Single
' Code that might throw an exception is wrapped in a Try block Try
    Dim sngAverage As Single


    ' This will cause an exception to be thrown if iItems = 0 sngAverage =
    CSng(iTotal \ iItems)
    ' This only executes if the line above generated no error
    MessageBox.Show("Calculation successful")
    Return sngAverage
  Catch excGeneric As Exception
    ' If the calculation failed, you get here MessageBox.Show("Calculation unsuccessful
  - exception caught")
    Return 0 End Try
```

End Function

This code traps all the exceptions with a single generic exception type, and you don't have any Finally logic. Run the program and press the button. You will be able to follow the sequence better if you place a breakpoint at the top of the GetAverage function and step through the lines.

Here is a more complex example that traps the divide-by-zero exception explicitly. This second version of the GetAverage function (notice that the name is GetAverage2) also includes a Finally block:

```
Private Function GetAverage2(iItems As Integer, iTotal As Integer) as Single
  ' Code that might throw an exception is wrapped in a Try block Try
    Dim sngAverage As Single
    ' This will cause an exception to be thrown. sngAverage =
    CSng(iTotal \ iItems)
    ' This only executes if the line above generated no error.
    MessageBox.Show("Calculation successful")
    Return sngAverage
  Catch excDivideByZero As DivideByZeroException
    ' You'll get here with an DivideByZeroException in the Try block
  MessageBox.Show("Calculation generated DivideByZero Exception")
    Return 0
  Catch excGeneric As Exception
    ' You'll get here when any exception is thrown and not caught in
    ' a previous Catch block.
¦ MessageBox.Show("Calculation failed - generic exception caught") Return 0
¦ Finally
'  ' Code in the Finally block will always run. MessageBox.Show("You always get here,
,  with or without an error")
¦ End Try
'  End Function
```

'
'
' This code contains two Catch blocks for different types of exceptions. If an exception is generated, then .NET will go down the Catch blocks looking for a matching exception type. That means the Catch blocks should be arranged with specific types first and more generic types after. Place the code for GetAverage2 in the form, and place another button on Form1. In the Click event for the second button, place the following code:
' *Dim sngAvg As Single*
' *sngAvg = GetAverage2(0, 100)*
'
' Run the program again and press the second button. As before, it's easier to follow if you set a breakpoint early in the code and then step through the code line by line.
'

The Throw Keyword

Sometimes a Catch block is unable to handle an error. Some exceptions are so unexpected that they should be "sent back up the line" to the calling code, so that the problem can be promoted to code that can decide what to do with it. A Throw statement is used for that purpose. A Throw statement, like an Err.Raise, ends execution of the exception handler – that is, no more code in the Catch block after the Throw statement is executed. However, Throw does not prevent code in the Finally block from running. That code still runs before the exception is kicked back to the calling routine.

You can see the Throw statement in action by changing the earlier code for GetAverage2 to look like this:

```
Private Function GetAverage3(iItems As Integer, iTotal as Integer) as Single
' Code that might throw an exception is wrapped in a Try block Try
    Dim sngAverage As Single
    ' This will cause an exception to be thrown.
    sngAverage = CSng(iTotal \ iItems)
    ' This only executes if the line above generated no error.
    MessageBox.Show("Calculation successful")
    Return sngAverage
  Catch excDivideByZero As DivideByZeroException
    ' You'll get here with an DivideByZeroException in the Try block.
MessageBox.Show("Calculation generated DivideByZero Exception")
     Throw excDivideByZero
  MessageBox.Show("More logic after the throw − never executed") Catch excGeneric
  As Exception
    ' You'll get here when any exception is thrown and not caught in
    ' a previous Catch block.
  MessageBox.Show("Calculation failed - generic exception caught") Throw excGeneric
  Finally
    ' Code in the Finally block will always run, even if
    '  an exception was thrown in a Catch block. MessageBox.Show("You always get
 here, with or without an error")
  End Try
End Function
```

Here is some code to call GetAverage3. You can place this code in another button's click event to test it out:

```
Try
```

Dim sngAvg As Single

sngAvg = GetAverage3(0, 100) Catch exc As Exception

MessageBox.Show("Back in the click event after an error") finally

MessageBox.Show("Finally block in click event") End Try

10.5 Throwing a New Exception

Throw can also be used with exceptions that are created on-the-fly. For example, you might want your earlier function to generate an ArgumentException, since you can consider a value of items of zero to be an invalid value for that argument.

In such a case, a new exception must be instantiated. The constructor allows you to place your own custom message into the exception. To show how this is done, let's change the above-mentioned example to throw your own exception instead of the one caught in the Catch block:

```
Private Function GetAverage4(iItems As Integer, iTotal as Integer) as Single
   If iItems = 0 Then
      Dim excOurOwnException As New _
         ArgumentException("Number of items cannot be zero")
       Throw excOurOwnException
   End If

 ' Code that might throw an exception is wrapped in a Try block. Try
      Dim sngAverage As Single


      ' This will cause an exception to be thrown. sngAverage =
      CSng(iTotal \ iItems)


      ' This only executes if the line above generated no error.
      MessageBox.Show("Calculation successful")
      Return sngAverage
   Catch excDivideByZero As DivideByZeroException
      ' You'll get here with an DivideByZeroException in the Try block.
      MessageBox.Show("Calculation generated DivideByZero Exception") Throw
      excDivideByZero
      MessageBox.Show("More logic after the thrown - never executed") Catch excGeneric
   As Exception
      ' You'll get here when any exception is thrown and not caught in
      ' a previous Catch block.
      MessageBox.Show("Calculation failed - generic exception caught") Throw excGeneric
   Finally
      ' Code in the Finally block will always run, even if
      '  an exception was thrown in a Catch block. MessageBox.Show("You always get here,
      with or without an error")
   End Try
End Function
```

This code can be called from a button with similar code for calling GetAverage3. Just change the name of the function called to GetAverage4.

This technique is particularly well suited to dealing with problems detected in property procedures. Property Set procedures often do checking to ensure that the property is about to be assigned a valid value. If not, throwing a new ArgumentException (instead of assigning the property value) is a good way to inform the calling code about the problem.

10.6 The Exit Try Statement

The Exit Try statement will break out of the Try or Catch block and continue at the Finally block under a given circumstance. In the following example, you are going to exit a Catch block if the value of iItems is 0, because you know that your error was caused by that problem:

```
Private Function GetAverage5(iItems As Integer, iTotal as Integer) As Single
  ' Code that might throw an exception is wrapped in a Try block. Try
    Dim sngAverage As Single
    ' This will cause an exception to be thrown. sngAverage =
    CSng(iTotal \ iItems)
    ' This only executes if the line above generated no error.
    MessageBox.Show("Calculation successful")
    Return sngAverage
  Catch excDivideByZero As DivideByZeroException
    ' You'll get here with an DivideByZeroException in the Try block. If iItems = 0 Then
      Return 0
      Exit Try Else
      MessageBox.Show("Error not caused by iItems") End If
    Throw excDivideByZero
  MessageBox.Show("More logic after the thrown - never executed")
  Catch excGeneric As Exception
    ' You'll get here when any exception is thrown and not caught in
    ' a previous Catch block.
  MessageBox.Show("Calculation failed - generic exception caught") Throw excGeneric
  Finally
    ' Code in the Finally block will always run, even if
    ' an exception was thrown in a Catch block. MessageBox.Show("You always get
here, with or without an error")
  End Try
End Sub
```

In your first Catch block, you have inserted an If block so that you can exit the block given a certain condition (in this case, if the overflow exception was caused by the value of intY being 0). The Exit Try goes immediately to the Finally block and completes the processing there:

```
If iItems = 0 Then Return 0
  Exit Try Else
  MessageBox.Show("Error not caused by iItems") End If
```

Now, if the overflow exception is caused by something other than division by zero, then you'll get a message box displaying Error not caused by iItems.

Nested Try Structures

In some cases, particular lines in a Try block may need special exception processing. Moreover,

errors can occur within the Catch portion of the Try structures and cause further exceptions to be thrown. For both of these scenarios, nested Try structures are available. You can alter the example under the section "**The Throw Keyword**" to demonstrate the following code:

```
Private Function GetAverage6(iItems As Integer, iTotal as Integer) As Single

    ' Code that might throw an exception is wrapped in a Try block. Try
      Dim sngAverage As Single
      ' Do something for performance testing....
      Try LogEvent("GetAverage")
      Catch exc As Exception MessageBox.Show("Logging function
        unavailable")
      End Try
      ' This will cause an exception to be thrown. sngAverage =
      CSng(iTotal \ iItems)

      ' This only executes if the line above generated no error.
      MessageBox.Show("Calculation successful")
      Return sngAverage

    Catch excDivideByZero As DivideByZeroException
      ' You'll get here with an DivideByZeroException in the Try block.
      MessageBox.Show("Error not divide by 0")
      Throw excDivideByZero
   MessageBox.Show("More logic after the thrown - never executed") Catch excGeneric
    As Exception
      ' You'll get here when any exception is thrown and not caught in
      ' a previous Catch block.
   MessageBox.Show("Calculation failed - generic exception caught") Throw excGeneric
    Finally
      ' Code in the Finally block will always run, even if
      '  an exception was thrown in a Catch block. MessageBox.Show("You always get
  here, with or without an error")
    End Try
End Function
```

In the preceding example, you are assuming that a function exists to log an event. This function would typically be in a common library, and might log the event in various ways. You will look at logging exceptions in detail later in the chapter, but a simple LogEvent function might look like this:

```
Public Function LogEvent(ByVal sEvent As String)
    FileOpen(1, "logfile.txt", OpenMode.Append)
    Print(1, DateTime.Now & "-" & sEvent & vbCrLf)
    FileClose(1)
End Function
```

In this case, you don't want a problem logging an event, such as a "disk full" error, to crash the routine. The code for the GetAverage function triggers a message box to indicate trouble with the logging function.

A Catch block can be empty. In that case, it has a similar effect as On Error Resume Next in VB6. The exception is ignored. However, execution does not pick up with the line after the line

that generated the error, but instead picks up with either the Finally block or the line after the End Try if no Finally block exists.

10.7 Using Exception Properties

The previous examples have displayed hard-coded messages into message boxes, and this is obviously not a good technique for production applications. Instead, a message box or log entry describing an exception should provide as much information as possible concerning the problem. To do this, various properties of the exception can be used.

The most brutal way to get information about an exception is to use the ToString method of the exception. Suppose that you modify the earlier example of GetAverage2 to change the displayed information about the exception like this:

```
Private Function GetAverage2(ByVal iItems As Integer, ByVal iTotal As Integer) _
    As Single
  ' Code that might throw an exception is wrapped in a Try block. Try
    Dim sngAverage As Single

    '  This will cause an exception to be thrown. sngAverage =
    CSng(iTotal \ iItems)
    'This only executes if the line above generated no error.
     MessageBox.Show("Calculation successful")
    Return sngAverage
  Catch excDivideByZero As DivideByZeroException
    ' You'll get here with an DivideByZeroException in the Try block.
     MessageBox.Show(excDivideByZero.ToString)
    Throw excDivideByZero
   MessageBox.Show("More logic after the thrown - never executed") Catch excGeneric
  As Exception
    ' You'll get here when any exception is thrown and not caught in
    ' a previous Catch block.
    MessageBox.Show("Calculation failed - generic exception caught") Throw excGeneric
  Finally
    ' Code in the Finally block will always run, even if
    '  an exception was thrown in a Catch block. MessageBox.Show("You always get
    here, with or without an error")
  End Try
End Function
```

When the function is accessed with iItems = 0, a message box similar to the one in Figure 10.1 will be displayed.

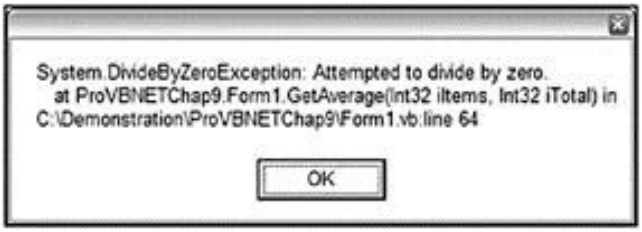

Fig. 10.1: Error message

The Message Property

The message in the dialog shown in Figure 10.1 is helpful to a developer because it contains a

lot of information, but it's not something you would typically want users to see. Instead, a user normally needs to see a short description of the problem, and that is supplied by the Message property.

If the previous code is changed so that the Message property is used instead of To String, then the message box will provide something like what is shown in Figure 10.2.

Fig. 10.2: Message property

The InnerException and TargetSite Properties

The InnerException property is used to store an exception trail. This comes in handy when multiple exceptions occur. It's quite common for an exception to occur that sets up circumstances whereby further exceptions are raised. As exceptions occur in a sequence, you can choose to stack your exceptions for later reference by use of the InnerException property of your Exception object. As each exception joins the stack, the previous Exception object becomes the inner exception in the stack.

For simplicity, you'll start a new code sample, with just a subroutine that generates its own exception. You'll include code to add a reference to an

InnerException object to the exception you are generating with the Throw method.

This example also includes a message box to show what's stored in the exception's TargetSite property. As shown in the results, TargetSite will contain the name of the routine generating the exception — in this case, HandlerExample. Here's the code:

```
Sub HandlerExample() Dim intX As Integer
 Dim intY As Integer Dim intZ As Integer
 intY = 0
 intX = 5
 ' First Required Error Statement. Try
  ' Cause a "Divide by Zero"
  intZ = CType((intX \ intY), Integer)

 ' Catch the error.
 Catch objA As System.DivideByZeroException Try
    Throw (New Exception("0 as divisor", objA)) Catch objB As
  Exception
   Dim sError As String
   sError = "My Message: " & objB.Message & vbCrLf & vbCrLf sError &= "Inner
   Exception Message: " & _
      objB.InnerException.Message & vbCrLf & vbCrLf
   sError &= "Method Error Occurred: " & objB.TargetSite.Name
   MessageBox.Show(sError)
 End Try Catch
Messagebox.Show("Caught any other errors") Finally
Messagebox.Show(Str(intZ)) End Try
```

End Sub

As before, you catch the divide-by-zero error in the first Catch block, and the exception is stored in objA so that you can reference its properties later. You throw a new exception with a more general message (“0 as divisor”) that is easier to interpret, and you build up your stack by appending objA as the InnerException object using an overloaded constructor for the Exception object:

```
Throw (New Exception("0 as divisor", objA))
```

You catch your newly thrown exception in another Catch statement. Note how it does not catch a specific type of error:

```
Catch objB As Exception
```

Then you construct an error message for the new exception and display it in a message box:

```
Dim sError As String
sError = "My Message: " & objB.Message & vbCrLf & vbCrLf sError &= "Inner
Exception Message: " & _
   objB.InnerException.Message & vbCrLf & vbCrLf
sError &= "Method Error Occurred: " & objB.TargetSite.Name
MessageBox.Show(sError)
```

The message box that is produced is shown in Figure 10.3.

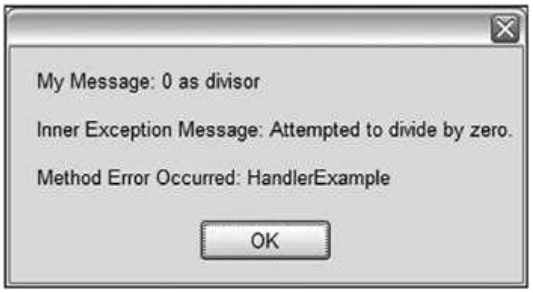

Fig. 10.3: Error message window

First your own message is included, based on the new exception thrown by your own code. Then the InnerException gets the next exception in the stack, which is the divide-by-zero exception, and its message is included. Finally, the TargetSite property gives you the name of the method that threw the exception. TargetSite is particularly helpful in logs or error reports from users that are used by developers to track down unexpected problems.

After this message box, the Finally clause displays another message box that just displays the current value of intZ, which is zero because the divide failed. This second box also occurs in other examples that follow.

10.8 Source and StackTrace

The Source and StackTrace properties provide the user with information regarding where the error occurred. This supplemental information can be invaluable for the user to pass on to the troubleshooter in order to help resolve errors more quickly. The following example uses these two properties and shows the feedback when the error occurs:

```
Sub HandlerExample2() Dim intX As Integer
 Dim intY As Integer Dim intZ As Integer intY
 = 0
```

```
intX = 5
' First Required Error Statement. Try
  ' Cause a "Divide by Zero"
  intZ = CType((intX \ intY), Integer)

 ' Catch the error.
 Catch objA As System.DivideByZeroException objA.Source =
   "HandlerExample2" Messagebox.Show("Error Occurred at :" & _
      objA.Source & objA.StackTrace) Finally
   Messagebox.Show(Str(intZ)) End Try
End Sub
```

The output from the Messagebox statement is very detailed, providing the entire path and line number where the error occurred, as shown in Figure 10.4.

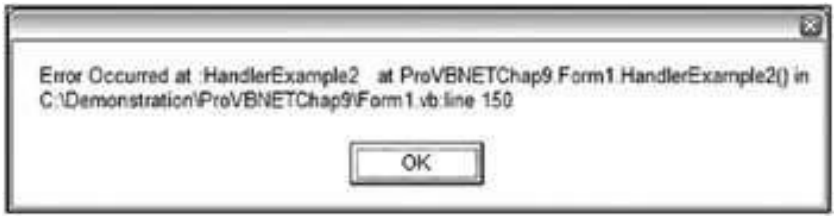

Figure 10.4: Error message with location

Notice that this information is also included in the ToString method examined earlier (refer to Figure 10.1).

10.9 GetBaseException

The GetBaseException method comes in very handy when you are deep in a set of thrown exceptions. This method returns the originating exception, which makes debugging easier and helps keep the troubleshooting process on track by sorting through information that can be misleading:

```
Sub HandlerExample3() Dim intX As Integer
 Dim intY As Integer

 Dim intZ As Integer intY = 0
 intX = 5
 ' First Required Error Statement. Try
  ' Cause a "Divide by Zero"
  intZ = CType((intX \ intY), Integer) ' Catch the error.
 Catch objA As System.DivideByZeroException Try
    Throw (New Exception("0 as divisor", objA)) Catch objB As
   Exception
    Try
      Throw (New Exception("New error", objB)) Catch objC As
     Exception
      Messagebox.Show(objC.GetBaseException.Message) End Try
   End Try Finally
  Messagebox.Show(Str(intZ)) End Try
End Sub
```

The InnerException property provides the information that the GetBaseException method needs, so as your example executes the Throw statements, it sets up the InnerException property. The purpose of the GetBaseException method is to provide the properties of the initial exception in the chain that was produced. Hence, objC.GetBaseException.Message returns the Message property of the original OverflowException message even though you've thrown multiple errors since the original error occurred:

Messagebox.Show(objC.GetBaseException.Message)

To put it another way, the code traverses back to the exception caught as objA and displays the same message as the objA.Message property would, as shown in Figure 10.5.

Fig. 10.5: Error message with GetBaseException

10.10 HelpLink

The HelpLink property gets or sets the help link for a specific Exception object. It can be set to any string value, but is typically set to a URL. If you create your own exception in code, you might want to set HelpLink to some URL describing the error in more detail. Then the code that catches the exception can go to that link. You could create and throw your own custom application exception with code like the following:

Dim exc As New ApplicationException("A short description of the problem") exc.HelpLink = "http://mysite.com/somehtmlfile.htm"
Throw exc

When trapping an exception, the HelpLink can be used to launch a viewer so the user can see the details about the problem. The following example shows this in action, using the built-in Explorer in Windows:

```
Sub HandlerExample4() Try
   Dim exc As New ApplicationException("A short description of the problem")
   exc.HelpLink = "http://mysite.com/somehtmlfile.htm" Throw exc
   ' Catch the error.
Catch objA As System.Exception Shell("explorer.exe " &
   objA.HelpLink)
End Try
End Sub
```

This results in launching Internet Explorer to show the page specified by the

URL. Most exceptions thrown by the CLR or the .NET Framework's classes have a blank HelpLink property. You should only count on using HelpLink if
you have previously set it to a URL (or some other type of link information) yourself.

10.11 Summary

- When an unexpected condition occurs, the CLR creates a special object called an exception
- The error handler takes control as soon as errors are detected in the routine, and determine what action to take.
- The Exception class has properties that contain useful information about the exception
- Try, Catch, and Finally are the Structured-Exception-Handling Keywords
- Throw block will handle the error those are not addressed by the catch block

10.12 Questions and Exercises

1. Describe the concept of Exceptions in .Net environment
2. Describe the structured Exception handling keywords
3. Explain the Exit Try statement with the suitable example
4. Describe the concept of Source and Stacktrace

.

10.13 Suggested Readings:

- http://msdn.microsoft.com/en-us/library/aa289505(v=vs.71).aspx
- http://en.wikipedia.org/wiki/Visual_Basic_.NET
- http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_ exceptions.html