Programming VB.NET: A Guide for Experienced Programmers

GARY CORNELL AND JONATHAN MORRISON

Apress™

Programming VB.NET: A Guide for Experienced Programmers

Copyright ©2002 by Gary Cornell

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-893115-99-2

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Karen Watterson

Technical Reviewers: Ken Getz, Tim Walton

Managing Editor and Production Editor: Grace Wong

Copy Editors: Steve Wilent, Tracy Brown Collins

Compositor: Susan Glinert Stevens

Artist: Allan Rasmussen

Indexer: Valerie Haynes Perry

Cover Designer: Karl Miyajima

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010

and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA 94710.

Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 7

Error Handling the VB .NET Way: Living with Exceptions

UP TO NOW, WE HAVE PRETENDED that bad things do not happen to our programs. But bad things happen to good programs all the time: a network connection may be down or the printer may run out of paper, for instance. It is not your fault as the programmer when this happens, but you also cannot blame the user if your program crashes because the network goes down! At the very least, your program must *not* end abruptly when these kinds of things happen. Your program must:

- Log or somehow notify the user of the problem.
- Let the user save his or her work if appropriate.
- Let the user gracefully exit the program if necessary.

This is not always easy. The code to open a network connection is usually not attached to the objects whose state you need to maintain. You often need some way to transfer control as well as to inform other objects what happened so they can deal with the situation.

NOTE *Bad things also happen to bad programs. If you do not validate data before you use it, you may find yourself dividing by zero or stuffing too much data into a container that cannot hold that much stuff. Your job as a programmer is to make sure your program does not do this. Whatever form of error handling you choose, it is not supposed to be a substitute for validating data before using it!*

In any case, good programmers know they live in a world where exceptional behavior often does not seem all that exceptional. This chapter will bring you into the real world.

The idea is that VB .NET finally supports *structured exception handling* (or simply, exception handling) for dealing with common errors. In this chapter, we not only show you the syntax used to add exception handling to a VB .NET application, but we show you the benefits of using it for error handling. For example, with exception handling, even the more or less socially acceptable use of the `GoTo` we showed you back in Chapter 3 is no longer necessary. But because power always comes at a cost, we also alert you to the gotchas that you will encounter when using exception handling.

> **NOTE** *For readers coming from earlier versions of VB, using the older* `On Error` *syntax is still possible. We think, however, that it would be almost foolish to continue using it for new programs. We feel strongly that is has taken far too long for VB to lose an archaic way of treating errors that goes back to the early days of computing! (You also cannot mix the two methods in the same procedure.)*

Error Checking vs. Exception Handling

Traditional error checking (such as that used in earlier versions of VB or in traditional COM or Windows programming) is done by checking the return value of a given function and reacting based on that value. This usually involves the equivalent of a giant switch statement that checks the value returned by the function. And, of course, this return value tends to be random: a 1 is good sometimes and bad other times; a 0 can mean success or just as often failure. Or, as is the case in the VB6 example code given here, the value returned seems truly random:

```
Select Case ErrorNumber
    Case 57
      MsgBox "Your printer may be off-line."
    Case 68
      MsgBox "Is there a printer available?"
   'more cases…
    Case Else
      'eeks
End Select
```

Now, this kind of code gets the job done, but it is hard to read and even harder to maintain. We think it is also fair to say that there is a lot of room for programmer error in this scheme. For instance, suppose that you had made a mistake with one of the error values or, as is all too common, you forgot to check all the possible

return values of the error function. Beyond this, it is a pain to write the same error-checking code every time you use a Windows API function in your code. Although there are times you will have to check the return value of a function regardless of what type of error handling scheme you are using, you do not want to do it everywhere. For example, one key benefit is efficiency: exception handling code costs you less time in writing, less time in maintenance, and often less time in executing!

First Steps in Exception Handling

Before we start writing the code that shows you some exception handlers at work, here are some things to keep in mind. First, when you use structured exception handling, you are providing an alternative path for your code that will be executed *automatically* when something bad happens. More precisely, you can create in any VB .NET code an alternate path for code execution when the code cannot complete its work in the normal path. Also, when you enable exception handling, VB .NET automatically creates an object that encapsulates the error information.

Once an exception is triggered, the built-in exception handling mechanism begins its search for a handler that can deal with that particular object (error condition). It is important to keep in mind that what we are describing is not a bunch of `GoTo`s that make for spaghetti code—it is more like the service road that runs parallel to the main highway with various exits. Next, keep in mind that, in a way, this service road is the dream of all drivers that are stuck in traffic. It is a smart service road—if something goes wrong, you will automatically be shunted to the exception handling code sequence. (Well, you will be if you wrote the code for the exception handler.) Once you are on the service road, the code in the exception handler can deal with the problem using *exception handlers* or, optionally, let it bubble up through the call chain.

The actual mechanism for doing this in VB .NET is called a `Try-Catch` block. Here is an example: Suppose you build a console application called `ProcessFile`. The user is supposed to use this application from the command line by typing something like:

```
ProcessFile nameoFile
```

where the argument on the command line is the name of the file. As users are prone to do, the user can do one of many annoying things to your nice program, such as:

- Forget to give you a filename
- Give you the name of a nonexistent file
- Ask you to work with a file that is locked for this operation

We have to write our code in a way that takes into account all the possible ways the user of our program can go wrong. Here is an example of the simple `Try-Catch` block in a VB .NET application that could be part of the `ProcessFile` application:

```
Module Exception1
  Sub Main()
  Dim args() As String
    Try
       args = Environment.GetCommandLineArgs()
       ProcessFile(args(1))
    Catch
       Console.WriteLine("ERROR")
    End Try
  Console.WriteLine("Press enter to end")
  Console.ReadLine()
  End Sub

  Sub ProcessFile(ByVal fileName As String)
    'process file code goes here
    Console.WriteLine("Am processing " & fName)
  End Sub
End Module
```

The code in the `Try` section of the `Try-Catch` block is assumed to be "good" code—in this case, a call to `ProcessFile`. (The reason the call to `Environment.GetCommandLineArgs()` in the exception handler is it can also throw an exception if your code is running on a box that does not support command line arguments.)

The code in the `Catch` section of the `Try-Catch` block is there because, well, users are users and do not always follow directions. In this code snippet, if the user forgets to enter a filename, then the code would try to reference the name of the file, which would trigger an `IndexOutOfRangeException`, because the array would not have an entry in the cited position. Triggering this exception causes the code flow to move down the alternate pathway (the `Catch` block), which in this case simply prints out `ERROR` in the console window.

> **NOTE** *As with most VB .NET control flow constructs such as* `For` *and* `Do`*, there is a way to exit from a* `Try` *block on demand. With a* `Try` *block, put an* `Exit Try` *inside of any* `Try` *block to exit from it immediately. We think using* `Exit Try` *is generally a poor programming practice.*

Analyzing the Exception

The next step is to catch the exception and to analyze it. This is done by modifying the `Catch` line to read something like:

```
Catch  excep As Exception
```

(You can use any variable here, of course, because it is being declared in the `Catch` clause.) Now, the exception object referenced by `excep` contains a lot of information. For example, change the code in the `Catch` clause to read:

```
Catch excep As Exception
  Console.WriteLine(excep)
```

to take advantage of the built-in `ToString` method of the exception object `excep` and you will see something like:

```
System.IndexOutOfRangeException: An exception of type_
System.IndexOutOfRangeException was thrown.
at Exception_1.Exception1.Main() in C:\Documents and_
Settings\x20\My Documents\Visual Studio
        Projects\ConsoleApplication14\Exception1.vb:line 6
```

This message shows there was an error in accessing the array element at line 6. (Not that we recommend printing this information out—unless you want to scare the user—but it is very useful for debugging.)

Finally, while reading this code we hope you are thinking ahead. Suppose the user supplies a filename but the `ProcessFile` method cannot process it. What then? Is there a way to differentiate between exceptions? As you will see shortly, you can make the `Catch` clause more sophisticated to check for different kinds of exceptions. You can even have a `Catch` clause `Throw` an exception object back to the code that called it that encapsulates what went wrong in its cleanup work.

Multiple Catch Clauses

The .NET runtime allows multiple `Catch` clauses. Each clause can trap for a specific exceptions, using objects that inherit from the base `Exception` class to identify the particular errors. For example, consider the following code:

```
Sub Main()
  Dim args(), argument As String
  Try
     args = Environment.GetCommandLineArgs()
     ProcessFile(args(1))
  Catch indexProblem As IndexOutOfRangeException
    Console.WriteLine("ERROR - No file name supplied")
  Catch ioProblem As System.IO.IOException
    Console.WriteLine("ERROR - can't process file named " & args(1))
  Catch except As Exception
    'other exception
  End Try
  Console.WriteLine("Press enter to end")
  Console.ReadLine()
End Sub
```

In this case, the exception handler looks inside the `Try-Catch` block, attempting to match all of the `Catch` blocks *sequentially* to find a match. If the user leaves out the filename, it will match the first clause. Presumably it will match the second clause if the `ProcessFile` call cannot process the file. (More on why this happens in a moment.) If not, the last `Catch` clause will catch any other kind of exception.

> **CAUTION** *Once a* `Catch` *clause that matches the exception is found, VB processes the code in that* `Catch` *block but will not process any other* `Catch` *block.*

Note that a match that is a `Catch` clause is an exception that is either is of the same type or of a type that *inherits* from that type. For example, the `FileNotFoundException` class inherits from `IOException`, so you should not write code that looks like this:

```
  Try
    ProcessFile(args(1))
  Catch indexProblem As IndexOutOfRangeException
    Console.WriteLine("ERROR - No file name supplied")
  Catch ioProblem As System.IO.IOException
    Console.WriteLine("ERROR - can't process file named " & args(1))
  Catch fileNotfound As System.IO.FileNotFoundException
  'will never be triggered
  End Try
```

because the more general `FileNotFoundException` clause will be headed off by the clause that caught its parent I/O exception.

CAUTION *This means a clause that says*

`Catch e As Exception`

kills all the remaining `Catch` *clauses. Using this clause as the first* `Catch` *block will cause you no end of grief. (Using* `Catch` *without specifying an exception is the equivalent of a* `Catch e As Exception` *clause, by the way.) Also note that if you use* `Catch e As Exception` *and do not put any code in the block, it will act much like the very dangerous* `On Error Resume Next` *from earlier versions of VB.*

In spite of the dangers of a `Catch e As Exception` line of code, a good rule of thumb is to actually have a general `Catch e As Exception` clause as the final `Catch` clause in any `Try` block—especially during the development process. This allows you to better isolate errors. We suggest printing out a stack trace to the console or a log file if all else fails. You can do this using the `StackTrace` method in the generic `Exception` class. For example:

```
Try
  ProcessFile(args(0))
Catch indexProblem As IndexOutOfRangeException
  Console.WriteLine("ERROR - No file name supplied")
Catch fnf As System.IO.FileNotFoundException
  Console.WriteLine("ERROR - FILE NOT FOUND")
Catch ioProblem As System.IO.IOException
  Console.WriteLine("ERROR - can't process file named " & args(1))
Catch e As Exception
  Console.WriteLine("Please inform the writer of this program of this message")
  Console.Write(e.StackTrace)
End Try
```

What happens if there is no `Catch` block that corresponds to the specific type of exception that is thrown, and if there is also no `Catch e As Exception` clause in the code you are trying? Well, when this happens, the exception bubbles up to any `Try` clauses that surrounds the code of an inner `Try` clause. And if there is no outer `Try` block with a matching `Catch` clause, then the exception bubbles up to the calling method and looks for an exception handler there. This is presumably what would happen in the `ProcessFile` method in the code you saw earlier—the `ProcessFile` method would pass on any unhandled exceptions (in the form of an `Exception` *object*) to `Sub Main`.

CAUTION *If no* `Try` *clause in the method catches the exception, execution processes to any* `Finally` *clauses, and then jumps immediately out of the method. This explains why you should think of exception handling as an awfully powerful (but smart)* `GoTo`*. It is smart because it will be able to perform cleanup code automatically through use of* `Finally` *clauses.*

In general, if your code does not handle an exception even when you go all the way up to the code in the entry point for the application, then .NET displays a generic message with a description of the exception and a stack trace of all the methods in the call stack when the exception occurred.

TIP *VB .NET allows you to add a* `When` *clause to a* `Catch` *clause to further specify its applicability. The syntax looks like this:*

```
Catch badNameException When theName = String.Empty
```

Throwing Exceptions

We said that the `ProcessFile` method would simply propagate the exception back to the code in `Sub Main` that called it. This code, in the `Main` procedure, in turn is inside a `Try` block, so the exception handling we wrote should handle it. But this is actually a little bit naïve and perhaps even becomes dangerous when you write classes that will be reused by other people. (And even if it is not dangerous, people who use your code will not be happy with you if you propagate exceptions willy-nilly without attempting to handle them.)

A better tactic is to do what you can locally to try to clean up the mess and then use the keyword `Throw` to send an exception object back to the calling code. For example, you saw in Chapter 4 how VB .NET no longer has deterministic finalization. Thus, if you create an object that has a `Dispose` method, you should dispose of it before throwing an exception. Ditto if you open a file or grab a graphic context. This snippet is the paradigm for this kind of code:

```
Try
  'code that created a local object that has a Dispose method
  ' more code that might throw exceptions
Catch(e As Exception)
  localObject.dispose()
  Throw e;
End Try
```

The point is that, if you do not call the `Dispose` method of your local object, whatever it grabbed will *never* be disposed of. This is because if you only have a reference to an object locally, other code will not have access to its `Dispose` method! On the other hand, whatever caused the exception did *not* go away and it is quite likely that the calling code needs to know that there was a problem, such as in processing the file. The way you do this is to send it an exception object using the `Throw` statement as you can see in the last bold line.

Actually, if you really want to be a good citizen, do not just (re)throw a generic exception as in the preceding code. Instead, make your code as useful as it can be to the calling code by adding information to the exception object you are throwing back. You can do this in three ways:

1. Add a descriptive string to the current exception and rethrow it with the new string added, and hope this helps.

2. Throw a built-in exception that inherits from the given exception that describes the situation better.

3. Create a new exception class that inherits from the given exception class that describes what happened better than any built-in exception class.

Ideally, these are in ascending order of usefulness, with number 3 being what you should always do. In practice, most people use all three methods based on their judgment of what will happen if they do not send every possible piece of information up the call stack.

As an example of how to perform these various tasks, imagine a situation where you are reading a bunch of key/value pairs back from some data source and the last key does not have a corresponding value. Because you assumed that every key has a value and tried to read its associated value, you are presented with an unexpected I/O exception. (See Chapter 9 for how to write code that reads information back from a file.)

Now you want to tell the caller of the code what has happened. You can add a string to an exception by using this version of the constructor in the `Exception` class:

```
Public Sub New(ByVal message As String)
```

For example, here is how you add a new string to the `IOException` that informs the caller that a value is missing for the last key and then throw it:

```
Dim excep as New IOException("Missing value for last key")
Throw excep
```

The code that calls your code is presented with the exception you throw; it can look at the value returned by the `Message` method in the `Exception` class to see what happened.

> **NOTE** *Actually, in the real world it is more likely that you would get an* `EndOfStreamException`*, which inherits from* `IOException`*. But more on streams in Chapter 9.*

The second situation is trivial to implement because of the cardinal rule of inheritance: any subclass must be usable wherever its parent class is. All you have to do is throw an instance of the child class exception that works better.

The final situation requires a bit more work because you have to build a class that extends an existing exception class. For example, suppose we want to build an exception class that inherits from `System.IO.IOException`. The only change is that we add a read-only property that returns the last key for the lost value:

```
Public Class LastValueLostException
   Inherits System.IO.IOException
  Private mKey As String
  Public Sub New(ByVal theKey As String)
    MyBase.New("No value found for last key")
    mKey = theKey
  End Sub
  Public ReadOnly Property LastKey() As String
    Get
      Return mKey
    End Get
  End Property
End Class
```

Note that the name of the newly created `Exception` class ends with the word `Exception`. This is a standard naming convention that we strongly suggest you follow. Someone who is presented with a `LastValueLostException` can use the read-only `LastKey` property that is set in the constructor of this new exception type to find the key that was not paired with a value. We made sure the `Message` method in the parent `Exception` gave the correct information by adding this line:

```
MyBase.New("No value found for last key")
```

which calls the correct constructor in the parent class (ultimately this constructor is in the parent `Exception` class).

You may notice that we do not override any other methods, such as the generic `ToString` method, which comes from `Exception`. Exception objects should always print out the standard message, if and when required.

How would somebody use our class? If the last key was "oops," then this line:

```
Throw New LastValueLostException("oops")
```

would do exactly that.

Exceptions in the Food Chain

We created a new exception class that inherited from `IOException`, because this was clearly the kind of problem we were having. Suppose, however, that you have a more generic situation where there is no obvious class to inherit from except `Exception` itself. Well, not quite—you always have a better choice. We strongly suggest not inheriting from `Exception` itself, but rather using a subclass of `Exception` called `ApplicationException`.

The reason is that the .NET Framework distinguishes between exceptions that arise because of problems caused by the runtime (such as running out of memory or stack space) and those caused by your application. It is the latter exceptions that are supposed to inherit from `ApplicationException`, and therefore this is the class you should inherit from when you create a generic exception in your program.

> **CAUTION** *Be aware that* `IOException`*, like most built-in non-generic exceptions, also splits off from* `Exception` *and not* `ApplicationException`.

The runtime tries to help you by going a little further. It actually splits the exception hierarchy into two as shown in Figure 7-1.

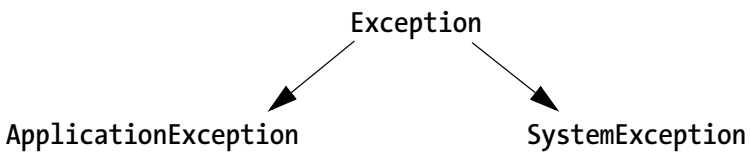

Figure 7-1. The exception hierarchy split into two

The Exception, ApplicationException, and SystemException classes have identical functionality—the existence of the three classes is a convenience that makes the exceptions your programs may cause easier to understand. Here is a summary of the most important members of these classes (which are also important for built-in classes such as IOException that inherit from Exception).

Eliminating the GoTo Using Exceptions

By combining exception handling with building your own exception classes, you can finally eliminate all uses of the GoTo. For example, in the code in Chapter 3 we showed that one possible socially acceptable use of the GoTo was to get out of a deeply nested loop when something bad happened in an inner loop. We would more likely just wrap the whole loop in a Try-Catch block as follows:

```
Sub Main()
  Dim getData As String
  Dim i, j As Integer
  Dim e As System.IO.IOException
  Try
    For i = 1 To 10
      For j = 1 To 100
      Console.Write("Type the data, hit the Enter key between " & _
             "ZZZ to end:  ")
      getData = Console.ReadLine()
      If getData = "ZZZ" Then
        e = New System.IO.IOException("Data entry ended at user request")
        Throw e
      Else
       'Process data
      End If
      Next j
    Next i
  Catch
    Console.WriteLine(e.Message)
    Console.ReadLine()
  End Try
End Sub
```

CAUTION *Do not change the preceding code by eliminating the first line in bold in favor of the following line, which replaces the second line in bold:*

```
Dim e As New System.IO.IOException("Data entry ended at user request")
```

Because of the block visibility rules in VB .NET, the `Catch` *clause would not be able to see the exception object.*

And Finally...Finally Blocks

When you use a `Try-Catch` block, there is often some cleanup code that must be processed in the normal *and* in the exceptional condition. Files should be closed in both cases; for example, `Dispose` methods need to be called, and so on. Even in the simple example that started this chapter, we should have the `ReadLine` code (which keeps the console window up while it waits for the user to press Enter).

You can assure that certain code executes no matter what happens by adding a `Finally` clause as in the bolded code in the following modification of our first example.

```
Sub Main()
  Dim args(), argument As String
  args = Environment.GetCommandLineArgs()
  Try
    ProcessFile(args(1))
  Catch
    Console.WriteLine("ERROR")
  Finally
    Console.WriteLine("Press enter to end")
    Console.ReadLine()
  End Try
End Sub
```

Now the code in bold will always be executed. (And so the DOS window will stay around long enough for the user to see what happened.)

CAUTION *Keep in mind that the code in a* `Finally` *clause will execute before any exceptions get propagated to the calling code and also before a function returns.*

Some Tips for Using Exceptions

Exceptions are cool and people new to them have a natural tendency to overuse them. After all, why go to the trouble to parse what the user enters when setting up an exception for the user's error is so easy? Resist this temptation. Exception handling will make your programs run much slower if misused. Here are four tips on using exceptions—they all come down to variations on the rule that exceptions are supposed to be exceptional:

1. Exceptions indicate an exceptional condition; do not use them as you would a return code for a function. (We have seen code that throws a "SUCCESS_EXCEPTION" every time a function call *does not* fail!)

2. Exception handling is not supposed to replace testing for the obvious. You do not, for example, use exceptions to test for end of file (EOF) conditions.

3. Do not micromanage exceptions by wrapping every possible statement in a `Try-Catch` block. It is usually better to wrap the whole action in a single `Try` statement than to have multiple `Try` statements.

4. Do not squelch exceptions by writing code like

   ```
   Catch e as Exception
   ```

   without a very good reason. This is the equivalent of blindly using `On Error Resume Next` in older VB code and doing so is bad for the same reasons. If an exception happens, handle it or propagate it.

5. Which leads to one final tip—what we like to call the good fellowship rule:

   If you do not handle an exception condition completely and need to rethrow an exception to the calling code, add enough information (or create an new exception class) so that the code you are communicating with knows exactly what happened and what you did to (try to) fix it.