Chapter 14 – Error Handling and Exception Handling

# Objectives

* Introduction to Error Handling and Exception Handling
* Unstructured Error Handling
* Modern Hip Exception Handling
* What Exceptions Exist?
* Creating Your Own Exception Handler
* The Using Statement

This chapter will concentrate exclusively on showing you how to handle the various errors that arise when your program is running. We will examine both the pre-.NET error handling model as well as the new .NET exception handling model. You may be wondering why you should learn about error handling. The answer is simple: I still run across code that has been ported from some ancient version of VB and error handling is still employed, so I at least want you to have a cursory understanding of what it does

You will find that the modern version of error handling, exception handling, is commonly employed in many other languages and it's easy to implement. By the time you are done with this chapter, you ought to be able to write fairly robust applications that can deal with unexpected runtime issues.

## 

# Introduction to Error Handling and Exception Handling

Error handling and exception handling refers to gracefully handling runtime problems within your own program, rather than relying on the operating system to do it. Runtime problems are just that: they are errors that arise during the execution of a program. This contrasts with a syntax error which is caught by the compiler before you have a working program to execute. Remember that syntax errors result from some lines of code not being structured according to the rules of the language. Runtime errors result from unexpected conditions that the computer finds itself in – this may be caused by things outside of your control, like a hardware device not being ready, or it could be from bad/faulty logic in your program (What? You don’t ever have that do you?). You might wonder “Why not let the operating system deal with the problem rather than you?” and the answer is quite simple – the O/S will just display a generic error message, which is usually cryptic in nature and then terminate your program.

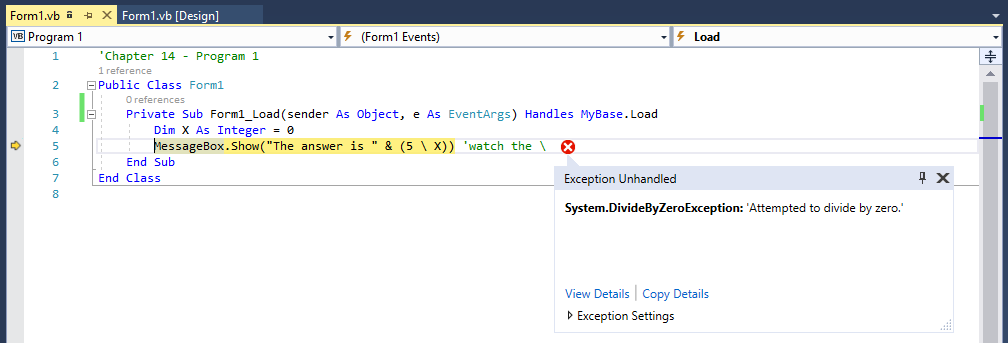
Thus, many inexperienced programmers' idea of dealing with errors is to simply allow the program to crash or else just let the operating system deal with it. Neither of these options is acceptable in commercial grade application systems. How happy would you be to use a program that crashes if you type in the wrong filename instead of providing you a "couldn't find that file" type error message? Or better yet, the O/S pops up a message that says “Invalid stream access 0x154E4743?” No, you expect better – so start providing better error support!

This section will address how to perform basic error handling in VB. Here’s a simple example of an error occurring in VB. Type the following code into your program’s Form\_Load handler subroutine and then run the program:

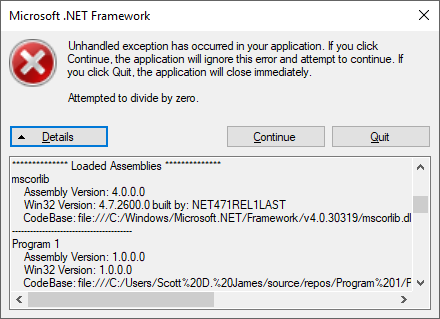
'Chapter 14 - Program 1

Dim X As Integer = 0

MessageBox.Show("The answer is " & (5 \ X)) 'watch the \



As you can see, the default error handling/exception handling in VB is to stop the program (it crashes and then gets terminated) and then provide a fairly useless error message. I don’t like crashing programs, regardless if an error message is printed or not, so we need to learn how to protect our programs from errors. By the way, what you see above is a fairly nice environment provided for you by the Visual Studio IDE. If you end user was running the compiled executable program outside of the IDE, they’d get this wonderful dialog instead:



That’s the O/S dealing with your errors! Boy, isn’t the detailed information with assembly and Win32 versioning so helpful? As an end user, I know exactly what went wrong…

The last item that I would like to mention in this introduction is exceptions versus errors. Up until object-oriented programming (OOP) languages became commonplace, programs had runtime errors. A runtime error occurred when something happened that the runtime environment couldn't deal with, such as dividing by zero or a request to read from a non-existent file. With the onset of OOP languages, exceptions get thrown; i.e. there is now a division by zero exception rather than a division by zero error. As you might expect exceptions are objects by nature so they fit nicely into the OOP paradigm.

Does it really matter that we differentiate between errors and exceptions? Not really. In fact, if you want to think of them as being the same entity, that's okay with me. They really are two different critters. What’s relevant is that VB.NET provides you with both error and exception handling. More importantly either mechanism can be used to write solid applications that can recover from problem by themselves, proving end users with a pleasant experience in using the software! Obviously, we will spend more time on exception handling since it is the preferred style in most modern programming languages. However, the next section will focus on the older Error Handling aspects of Visual Basic.

## 

# Unstructured Error Handling

This section will discuss using the On Error statement to catch errors within your programs. The On Error statement was the only utility that pre-.NET versions of Visual Basic had and it still is available in VB.NET. In fact, VBA (Visual Basic for Applications) and VBScript still only support On Error handling. We do realize that the more modern way to handle errors in VB.NET is to use exception handling -- we'll discuss that topic in the next section.

Error handling is quite simple: we can capture errors by enabling error traps using an On Error GoTo trap. The way that this works is that the On Error Goto line of code is placed before any operation that could potentially cause a problem. If a problem does occur, control of the program is transferred to the label that follows the GoTo statement.

Wherever the program is redirected to go, following an error, is known as an error handler. The error handler would then perform whatever work it needs to carry out in response to knowing that an error just occurred. Typically, the last task that would be carried out in the error handler would be to turn error trapping back off. This can be done with the line of code On Error GoTo 0.

It is important to shut error handling off when you don't need it. The reason being is that if an unrelated error occurs anywhere else in your program and error handling is not turned off, the error will be caught by the last error trap that you set up. Therefore, it's possible to set up an error handler to catch division by zero errors. If you forget to turn the error handling off and a weird, unrelated error like “file not found” occurs, your program will still jump to the division by zero error handler, which is obviously not at all what you want.

Now that we've discussed error handling, let's look at it in action. Here is our division by zero program again, still doing bad math in Form\_Load:

'Chapter 14 - Program 2

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

Dim X As Integer = 0

'Start trapping errors by turning error handling on via

'an On Error GoTo statement. From this point forward, if an

'error occurs, the program will jump to the handler BadNumber

'which is a label that appears later in this subroutine.

'Notice the two subsequent lines both attempt division by zero.

On Error GoTo BadNumber

MessageBox.Show("The answer is " & (5 \ X))

MessageBox.Show("The answer is " & (6 \ X))

'If we make it this far, then everything executed okay and there

'was no division by zero. We really don't need error handling

'code anymore so shut off the error handling and end the Sub.

On Error GoTo 0

Exit Sub

'This is our error handler, if we get here, then there

'was an error caught by our error trap

BadNumber:

'Here's where we write the code to deal with the error that

'triggered the GoTo to this point.

'Let the user know what happened

MessageBox.Show("You tried to divide by Zero – no no no!")

'Shut error handling back off since we've dealt with the

'error

On Error GoTo 0

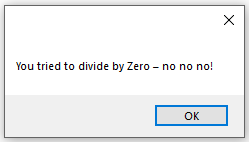
'We don't have to exit the Sub manually since the next line

'of code does it automatically.

End Sub

End Class

If you run the program again now, here’s what happens when we try to divide by zero:



As you can see, our program no longer blows up when an error occurs. We caught the error and handled it internally in our program. The O/S wasn’t ever made aware that anything bad happened; this is exactly what well behaved applications ought to do. You also saw that there wasn't a terrible amount of code required to do this. What’s nice about this technique is that what happens in the IDE and what the end user sees when running the actual application is identical! No more cryptic messages. Furthermore, if we click the OK button, the application stays running – nothing terminated it. This means that we caught the error and could build in recovery mechanisms to allow the user to try putting in a different number…

The On Error statement is very useful for enabling or disabling error handling within a procedure. You even have the ability to use multiple On Error statements to try to catch a multitude of different problems. For example, we could write some code similar to this, in which we are looking for different errors types that could occur:

Sub DoMath()

'.

'.

'.

On Error GoTo BadInputDataHandler

'Try to get valid data – if we get bad input, then go to the

'BadInputDataHandler error handler

'.

'.

'.

'If we get this far, we must have received good input...

On Error GoTo DivisionByZeroHandler

'Try to do some math – we are no longer worried about bad input, now

'we shift our focus to a different error handler for Division by Zero

'.

'.

'.

'If we get this far, the math must have been good, so we can shut off

'the error handling and get out of the subroutine

On Error GoTo 0

Exit Sub

'Here is where all of our error handlers would be placed...

BadInputDataHandler:

'Code goes here

Exit Sub

'Obviously we could have multiple handlers here – just remember to end

'each one with an Exit Sub otherwise you will cascade from one error

'handler into the next.

DivisionByZeroHandler:

'Code goes here

'We don't need an Exit Sub here since the next line exits the sub

'automatically.

End Sub

## 

## The Resume Statement

The Resume statement in VB can be used to continue program execution when an error-handling routine is complete. This again is a pre-.NET remnant that is still currently available.

If you look over the previous example in which we attempted the twin division by zero statements, as soon as the first statement triggered the error trap, the handler fired and we left the subroutine. We never went back up and tried to execute any more lines after the point at which the error occurred. The Resume statement corrects that mistake.

There are three different versions of the Resume statement:

* *Resume* – If an error occurs, transfer control to the same line that caused the error.
* *Resume Next* – If an error occurs, transfer control to the next line following the line that caused the error.
* Resume *label* – If an error occurs, transfer control to the specified label. The only limitation here is that the label must be in the same procedure as the error handler.

Error handlers obviously do not have to include Resume statements – nothing we’ve looked at so far has used any form of a Resume. Let's modify our working program so that the code simply continues executing the program after any line that errors out without printing any message by using the Resume Next command:

'Chapter 14 - Program 3

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

Dim X As Integer = 0

'Start trapping errors – by using On Error Resume Next

'if an error occurs, simply go to the line following the

'error

On Error Resume Next

MessageBox.Show("The answer is " & (4 \ 3))

MessageBox.Show("The answer is " & (6 \ X))

MessageBox.Show("The answer is " & (X \ 6))

On Error GoTo 0

End Sub

If you run this version of the program, you will see that if there is an error, we just ignore it and proceed to the next line. When this program is run we are shown “The answer is 1” (from the first MessageBox.Show) and “The answer is 0” (from the third MessageBox.Show). No noise is made at all about the actual division by 0 line (the second MessageBox.Show).

We could have also used the Resume Next statement in the old error handling routine that we had. This would have allowed us to give the user some indication as to what was wrong, but still trap the errors. The following code shows the revised program:

'Chapter 14 - Program 4

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

Dim X As Integer = 0

'Start trapping errors

On Error GoTo BadNumber

MessageBox.Show("The answer is " & (4 \ 3))

MessageBox.Show("The answer is " & (6 \ X))

MessageBox.Show("The answer is " & (X \ 6))

On Error GoTo 0

Exit Sub

'Here's the error handler – we print out a message

'so the user knows what went wrong and then we send

'control back up to the line right after the error

'causing line.

BadNumber:

MessageBox.Show("You can't divide by zero!")

Resume Next

End Sub

End Class

Now if you run the program you will receive the following messages: “The answer is 1”, “You cannot divide by zero!” and “The answer is 0.” Error handling is relatively easy and pretty neat stuff… You get a lot of power with very little coding effort expended.

So why did error handling have a bad rap and get replaced in .NET?

1. The GoTo statement – it could be a line number, it could be a label. The bigger issue was the spaghetti code it creates, especially with the resume statement’s jumping around. Extra work had to be taken to make sure that the error handling architecture was clean and made sense. Even then, as with the examples shown above, notice how there were multiple Exit Sub/End Sub statements? That’s bad programming practice – we like a single entry point and a single exit point. Bailing out wherever seems very kludgy.
2. It wasn’t object oriented – this means that creating classes of related errors hierarchically wasn’t possible in error handling. In addition to being able to create is-a error classes, modern exception handling will allow exceptions to bubble up to the appropriate level to be processed. In other words, if a child exception isn’t handled directly, a parent exception handler could still deal with the exception so the program doesn’t terminate.
3. Creating your own error values was more difficult – modern exception handling allows you to extend the exception model allowing you to add in whatever kinds of exceptions you need and that make sense to you to use.

So now that we’ve talked about how we arrived at exception handling, let’s take a look at it!

# Modern Hip Exception Handling

Exception handling is the new style of error control that you should be using with VB. Exception handling consists of Try and Catch statements similar to what you should have seen in C++ or Java. We wrap a try block around code that may cause an exception to be thrown. The catch statements allow us to state which exceptions we want our program to handle internally.

If we don't handle a particular exception internally, our program will then pass the exception up to the operating system. This means that we'll get the default system error message just like we saw in our first error handling program. Let's slowly build up our exception abilities by looking at how to handle various exceptions.

## Handling a Specific Exception

In the examples that we have been looking at up to this point, we knew that we were creating a Divide by Zero error/exception. Let's write exception handling code that specifically tries to deal with that particular exception:

'Chapter 14 - Program 5

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

Dim X As Integer = 0

'Wrap any code which you think might cause an exception that

'you would want your program to handle inside of a Try block

Try

MessageBox.Show("The answer is " & (4 \ 3))

MessageBox.Show("The answer is " & (6 \ X))

'This line won't execute because an exception is

'generated above and control will be transferred to

'the appropriate Catch handler

MessageBox.Show("The answer is " & (X \ 6))

'Here is where we will catch a DivisionByZeroException if that

'is the type of exception that was thrown in the try block

Catch Ex As DivideByZeroException

'Here's where we put the exception handler's code

MessageBox.Show("You cannot divide by zero")

'Notice the code that could cause an exception and the

'handler are all sandwiched within the Try block...the

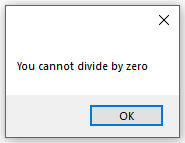
'next line of code "shuts off" the exception handling

End Try

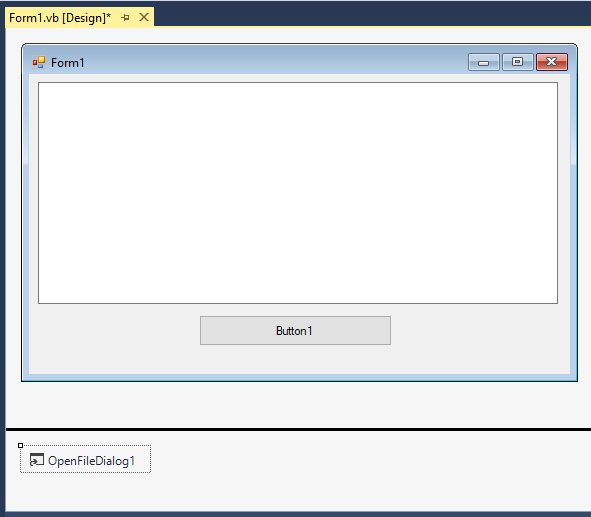
End Sub

End Class

Now when your program runs, you have said that if the specific exception “Division by Zero” occurs in the Try block of code, you want to handle the exception with a handler internal to your program. Here's what the program looks like when it's run – you must admit it's a lot nicer to catch the exception than to crash when it occurs:



While we are on the subject of catching some simple exceptions, let's take a look at another common example. We will write an exception handler that catches bad filenames when we use an OpenFileDialog. Create a form that has a multiline TextBox, a Command button and an OpenFileDialog control on it all with the default names. The next figure shows an example of the form:



Here's the source code behind the form:

Imports System.IO

'Chapter 14 - Program 6

Public Class Form1

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

'The only thing in this form that we respond to is the

'clicking of the command button

Dim OpenFileDialog1 As New OpenFileDialog()

Dim SourceFile As StreamReader

'Set up the OpenFileDialog which we dynamically created

'above

OpenFileDialog1.Filter = "All Files | \*.\* | Text Files | \*.txt"

OpenFileDialog1.FilterIndex = 2

OpenFileDialog1.InitialDirectory = "C:\"

OpenFileDialog1.DefaultExt = "txt"

'Don't let the dialog box check for file exists

'otherwise our exception handling example won't work

OpenFileDialog1.CheckFileExists = False

'Since we don't know if a file exists and the user

'types in a bad name, we will generate an exception

'if we blindly try to read from that file using a

'StreamReader object

If (OpenFileDialog1.ShowDialog() = DialogResult.OK) Then

'If we got here, the user clicked the OK button on

'the dialog -- so try to load the file. However,

'since we don't know if the filename the user entered

'is any good -- throw a Try block up just in case...

Try

SourceFile = New StreamReader(OpenFileDialog1.FileName)

TextBox1.Text = SourceFile.ReadToEnd()

SourceFile.Close()

'The only exception to worry about is the FileNotFound.

'So here's the handler

Catch Ex As FileNotFoundException

MessageBox.Show("File not found!")

End Try

Else

'The user didn't click the OK button in the OpenFileDialog

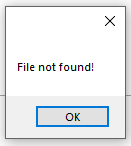
MessageBox.Show("User selected cancel")

End If

End Sub

End Class

Now, even though the OpenFileDialog doesn't look for a bad filename itself, a bad filename still won't blow up the program due to our own exception handling:



## Testing for Different Exceptions

In the previous example, we only looked for one specific exception that could occur while working with files. Obviously, there are a lot of other things that a user could do that would generate many different types of exceptions. Your catch block can simply be extended to handle the various exceptions that you want to process.

Here’s a snippet of code that shows catching both FileNotFound and DirectoryNotFound exceptions within the same Try block:

Try

SourceFile = New StreamReader(OpenFileDialog1.FileName)

TextBox1.Text = SourceFile.ReadToEnd()

SourceFile.Close()

Catch Ex As FileNotFoundException

MessageBox.Show("File not found!")

Catch Ex As DirectoryNotFoundException

MessageBox.Show("Directory not found!")

End Try

## What about Handling Exceptions that I Don't Know About?

There are many exceptions that you might not plan on occurring, but they could still get thrown. How can you write defensive code against things you aren't planning on? You can simply create a generic catch statement to handle exceptions that arise that you didn't specifically plan to catch.

You might want to spend some time exploring the MSDN help on the Exception class. Every exception that can be thrown is a child of the Exception class. One of the most useful properties in the class is .Message, which contains the exception message detailing the exception that was thrown. The syntax of both the generic handler and the .Message property is illustrated in the following code snippet:

Try

SourceFile = New StreamReader(OpenFileDialog1.FileName)

TextBox1.Text = SourceFile.ReadToEnd()

SourceFile.Close()

Catch Ex As FileNotFoundException

MessageBox.Show("File not found!")

Catch Ex As DirectoryNotFoundException

MessageBox.Show("Directory not found!")

'Notice this Exception is non-specific – this is our

'generic exception handler and IT MUST BE LAST in the

'list of those exception we want to catch.

Catch Ex As Exception

MessageBox.Show(Ex.Message)

End Try

## Cleaning Up After an Exception Occurs

You have already seen that you can write multiple catches for a single Try. There is another statement that you need to be aware of when working with exceptions and that is the Finally statement. The purpose of Finally is to allow you to specify a block of code which will be performed regardless of whether an exception occurs or not. That's right, the Finally block will be executed if any exception occurs and even if no exception occurs.

Here's a code snippet showcasing the Finally statement. This is the same example that we looked at above except that there is a Finally statement placed in the Try block. This code would disable the textbox regardless of whether we were able to successfully read in the text from the filename provided by the user:

Try

SourceFile = New StreamReader(OpenFileDialog1.FileName)

TextBox1.Text = SourceFile.ReadToEnd()

SourceFile.Close()

Catch Ex As FileNotFoundException

MessageBox.Show("File not found!")

Catch Ex As DirectoryNotFoundException

MessageBox.Show("Directory not found!")

Catch Ex As Exception 'this is generic

MessageBox.Show(Ex.Message)

Finally

'Lock the textbox, so it essentially is

'only a file viewer, not an editor!

'This will trigger regardless if there

'is an exception or not...

TextBox1.Enabled = False

End Try

Let's now look at a more complicated example with multiple Try blocks and a Finally that really does something useful. In this case, we try to open a file up using a StreamReader object. If we are successful in opening the file, then we want to read the entire file in and place it in a textbox. Regardless if we are successful in the read operation or not, we want to make sure that the file is closed before continuing with the program – therefore, we will use a Finally statement. Here’s the code sample:

Try

SourceFile = New StreamReader(OpenFileDialog1.FileName)

Catch Ex As Exception

MessageBox.Show("Error: " & Ex.Message)

End Try

If (Not SourceFile Is Nothing) Then

'We have connected the StreamReader to a valid file

Try

TextBox1.Text = SourceFile.ReadToEnd()

Catch Ex As Exception

MessageBox.Show("Error: " & Ex.Message)

Finally

'Since StreamReader was open, we want to try to close

'it even if we weren't able to read from it.

SourceFile.Close()

End Try

End If

## Using the When Filter

The When filter is an optional clause that will allow you to catch exceptions that occur but only when a criteria that you specify is true. The syntax is for using the When filter is Catch Exc As Exception When (criteria is true). Let's see an example of using When in the following code:

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles Me.Load

Dim X, Y As Integer

Dim isDone As Boolean

Try

X = 1

Y = 0

MessageBox.Show("Answer is :" & (5 / X))

MessageBox.Show("Answer is :" & (5 / Y))

'This is called when a division by zero exception

'occurs and X is equal to 0.

Catch Ex1 As DivideByZeroException When X = 0

MessageBox.Show("can't divide by x of 0")

'This is called when a division by zero exception

'occurs and Y is equal to 0.

Catch Ex2 As DivideByZeroException When Y = 0

MessageBox.Show("Can't divide by y of 0")

Finally

'You can still add Finally statements with When

isDone = True

End Try

End Sub

Some of the real power of the When filter is achieved by hooking it up with the old pre-.NET error object, which is accessed through its name Err. Err is what was responsible for holding any error condition that occurred while a VB program was running. The Err object has several properties and methods available. The two properties that we are interested in when an error occurs are the Err.Number and Err.Description items which contain the error number and the description of the error respectively.

Again, keep in mind that the Err object is a throwback from the pre-.NET days. You may not want to put a whole lot of effort into using it since it may not always be around as the language moves forward. However, since many pre-.NET programs were written using the Err object, understanding how to utilize it in modern VB makes porting older programs a bit easier. Here are some examples of using the Err object and the When filter:

'Chapter 14 - Program 7

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

Dim myArray(5) As Integer

Try

myArray(10) = 100

'VB6's Err object would be set to Error 9 whenever you tried to

'move past the upper boundary of an array...

Catch Ex As IndexOutOfRangeException When Err.Number = 9

MessageBox.Show(Ex.ToString)

'.

'.

'.

End Try

'A second exception of the same kind, but catching it via the Err object

Try

myArray(10) = 100

'.

'.

'.

'We can also write catch statements for errors that arise without

'using exceptions...notice that we are still within the Try block

'though. This example will catch any array boundary errors.

Catch When Err.Number = 9

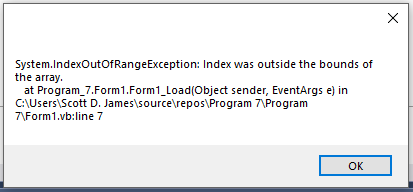
MessageBox.Show(Err.Description)

End Try

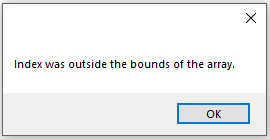
End Sub

End Class

Here’s the output we get from the first exception handler:



Here’s the output from the second exception handler:



## Throwing an Exception

Now that we have worked with exceptions, the next question you may have is "Can I throw an exception?" There may be times when you want to be able to do this yourself. In VB, the answer to your question is "yes."

Let's say, for example, that you have checked the denominator of a number and you know that it is 0, which will result in a DivisionByZeroException. Let's take the initiative to throw the exception ourselves rather than waiting for the runtime environment to do it for us. Here's some code that shows how to do this.

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles Me.Load

Dim Denominator As Integer

Try

If Denominator <= 0 Then

'If we know that the denominator is 0 and we are planning to do

'division, there will be a DivideByZeroException. In this example

'rather than blindly trying to do the division and getting the

'exception, we will manually be responsible for throwing the

'exception.

'Notice the syntax...you simply say Throw followed by

'a New instance of whatever exception type it is that you want to

'throw from your program.

Throw New DivideByZeroException

End If

Catch Ex As DivideByZeroException

MessageBox.Show(Ex.Message)

End Try

End Sub

You also have the ability to catch an exception, perform some processing and then rethrow that same exception. Here's a snippet of code that shows how to do this:

'Chapter 14 - Program 8

Public Class Form1

Private Sub MySub()

Dim Denominator As Integer = 0

Try

'This is sure to blow up...

MessageBox.Show("Answer is " & (5 \ Denominator))

Catch Ex As DivideByZeroException

'We caught it here, but we don't want to deal with it

'at this level, so throw it again, and let whatever

'called us handle it (hopefully)...

'

'We will rethrow a new divide by zero exception but we'll

'put in a custom message...

Throw New DivideByZeroException("No! Bad programmer! " &

"No zero dividing", Ex)

End Try

End Sub

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

Try

'Could be weird code in MySub that might go bad, so we'd

'best wrap it in a try

MySub()

Catch Ex As Exception

'Don't know what kind of exception might come back, so grab

'anything and handle it generically...

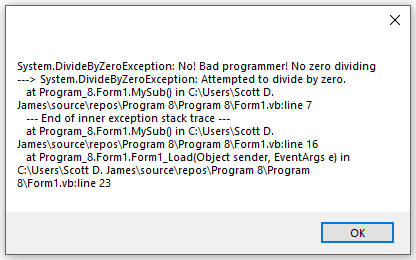
MessageBox.Show(Ex.ToString)

End Try

End Sub

End Class

Here’s the output. Notice the message that we sent in MySub bubbled back up to the Load routine:



# What Exceptions Exist?

As I just previously mentioned, you really need to consult the MSDN documentation on each class that you use in your application to determine what exceptions can be raised by that particular class. Here's a list of the exception classes that are defined in .NET – understand that each one of these classes may have more exceptions defined within it.

List of exception classes defined under the System.SystemException namespace:

|  |
| --- |
| [System.AccessViolationException](http://msdn.microsoft.com/en-us/library/system.accessviolationexception.aspx) |
| [System.Activities.ValidationException](http://msdn.microsoft.com/en-us/library/system.activities.validationexception.aspx) |
| [System.AppDomainUnloadedException](http://msdn.microsoft.com/en-us/library/system.appdomainunloadedexception.aspx) |
| [System.ArgumentException](http://msdn.microsoft.com/en-us/library/system.argumentexception.aspx) |
| [System.ArithmeticException](http://msdn.microsoft.com/en-us/library/system.arithmeticexception.aspx) |
| [System.ArrayTypeMismatchException](http://msdn.microsoft.com/en-us/library/system.arraytypemismatchexception.aspx) |
| [System.BadImageFormatException](http://msdn.microsoft.com/en-us/library/system.badimageformatexception.aspx) |
| [System.CannotUnloadAppDomainException](http://msdn.microsoft.com/en-us/library/system.cannotunloadappdomainexception.aspx) |
| [System.Collections.Generic.KeyNotFoundException](http://msdn.microsoft.com/en-us/library/system.collections.generic.keynotfoundexception.aspx) |
| [System.ComponentModel.Design.Serialization.CodeDomSerializerException](http://msdn.microsoft.com/en-us/library/system.componentmodel.design.serialization.codedomserializerexception.aspx) |
| [System.ComponentModel.LicenseException](http://msdn.microsoft.com/en-us/library/system.componentmodel.licenseexception.aspx) |
| [System.ComponentModel.WarningException](http://msdn.microsoft.com/en-us/library/system.componentmodel.warningexception.aspx) |
| [System.Configuration.ConfigurationException](http://msdn.microsoft.com/en-us/library/system.configuration.configurationexception.aspx) |
| [System.Configuration.Install.InstallException](http://msdn.microsoft.com/en-us/library/system.configuration.install.installexception.aspx) |
| [System.ContextMarshalException](http://msdn.microsoft.com/en-us/library/system.contextmarshalexception.aspx) |
| [System.Data.DataException](http://msdn.microsoft.com/en-us/library/system.data.dataexception.aspx) |
| [System.Data.DBConcurrencyException](http://msdn.microsoft.com/en-us/library/system.data.dbconcurrencyexception.aspx) |
| [System.Data.OperationAbortedException](http://msdn.microsoft.com/en-us/library/system.data.operationabortedexception.aspx) |
| [System.Data.SqlTypes.SqlTypeException](http://msdn.microsoft.com/en-us/library/system.data.sqltypes.sqltypeexception.aspx) |
| [System.DataMisalignedException](http://msdn.microsoft.com/en-us/library/system.datamisalignedexception.aspx) |
| [System.Deployment.Application.DeploymentException](http://msdn.microsoft.com/en-us/library/system.deployment.application.deploymentexception.aspx) |
| [System.DirectoryServices.AccountManagement.PrincipalException](http://msdn.microsoft.com/en-us/library/system.directoryservices.accountmanagement.principalexception.aspx) |
| [System.Drawing.Printing.InvalidPrinterException](http://msdn.microsoft.com/en-us/library/system.drawing.printing.invalidprinterexception.aspx) |
| [System.EnterpriseServices.RegistrationException](http://msdn.microsoft.com/en-us/library/system.enterpriseservices.registrationexception.aspx) |
| [System.EnterpriseServices.ServicedComponentException](http://msdn.microsoft.com/en-us/library/system.enterpriseservices.servicedcomponentexception.aspx) |
| [System.ExecutionEngineException](http://msdn.microsoft.com/en-us/library/system.executionengineexception.aspx) |
| [System.FormatException](http://msdn.microsoft.com/en-us/library/system.formatexception.aspx) |
| [System.IdentityModel.LimitExceededException](http://msdn.microsoft.com/en-us/library/system.identitymodel.limitexceededexception.aspx) |
| [System.IdentityModel.SecurityMessageSerializationException](http://msdn.microsoft.com/en-us/library/system.identitymodel.securitymessageserializationexception.aspx) |
| [System.IdentityModel.Tokens.SecurityTokenException](http://msdn.microsoft.com/en-us/library/system.identitymodel.tokens.securitytokenexception.aspx) |
| [System.IndexOutOfRangeException](http://msdn.microsoft.com/en-us/library/system.indexoutofrangeexception.aspx) |
| [System.InsufficientExecutionStackException](http://msdn.microsoft.com/en-us/library/system.insufficientexecutionstackexception.aspx) |
| [System.InvalidCastException](http://msdn.microsoft.com/en-us/library/system.invalidcastexception.aspx) |
| [System.InvalidOperationException](http://msdn.microsoft.com/en-us/library/system.invalidoperationexception.aspx) |
| [System.InvalidProgramException](http://msdn.microsoft.com/en-us/library/system.invalidprogramexception.aspx) |
| [System.IO.InternalBufferOverflowException](http://msdn.microsoft.com/en-us/library/system.io.internalbufferoverflowexception.aspx) |
| [System.IO.InvalidDataException](http://msdn.microsoft.com/en-us/library/system.io.invaliddataexception.aspx) |
| [System.IO.IOException](http://msdn.microsoft.com/en-us/library/system.io.ioexception.aspx) |
| [System.Management.ManagementException](http://msdn.microsoft.com/en-us/library/system.management.managementexception.aspx) |
| [System.MemberAccessException](http://msdn.microsoft.com/en-us/library/system.memberaccessexception.aspx) |
| [System.MulticastNotSupportedException](http://msdn.microsoft.com/en-us/library/system.multicastnotsupportedexception.aspx) |
| [System.NotImplementedException](http://msdn.microsoft.com/en-us/library/system.notimplementedexception.aspx) |
| [System.NotSupportedException](http://msdn.microsoft.com/en-us/library/system.notsupportedexception.aspx) |
| [System.NullReferenceException](http://msdn.microsoft.com/en-us/library/system.nullreferenceexception.aspx) |
| [System.OperationCanceledException](http://msdn.microsoft.com/en-us/library/system.operationcanceledexception.aspx) |
| [System.OutOfMemoryException](http://msdn.microsoft.com/en-us/library/system.outofmemoryexception.aspx) |
| [System.Printing.PrintSystemException](http://msdn.microsoft.com/en-us/library/system.printing.printsystemexception.aspx) |
| [System.RankException](http://msdn.microsoft.com/en-us/library/system.rankexception.aspx) |
| [System.Reflection.AmbiguousMatchException](http://msdn.microsoft.com/en-us/library/system.reflection.ambiguousmatchexception.aspx) |
| [System.Reflection.ReflectionTypeLoadException](http://msdn.microsoft.com/en-us/library/system.reflection.reflectiontypeloadexception.aspx) |
| [System.Resources.MissingManifestResourceException](http://msdn.microsoft.com/en-us/library/system.resources.missingmanifestresourceexception.aspx) |
| [System.Resources.MissingSatelliteAssemblyException](http://msdn.microsoft.com/en-us/library/system.resources.missingsatelliteassemblyexception.aspx) |
| [System.Runtime.InteropServices.ExternalException](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.externalexception.aspx) |
| [System.Runtime.InteropServices.InvalidComObjectException](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.invalidcomobjectexception.aspx) |
| [System.Runtime.InteropServices.InvalidOleVariantTypeException](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.invalidolevarianttypeexception.aspx) |
| [System.Runtime.InteropServices.MarshalDirectiveException](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.marshaldirectiveexception.aspx) |
| [System.Runtime.InteropServices.SafeArrayRankMismatchException](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.safearrayrankmismatchexception.aspx) |
| [System.Runtime.InteropServices.SafeArrayTypeMismatchException](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.safearraytypemismatchexception.aspx) |
| [System.Runtime.Remoting.RemotingException](http://msdn.microsoft.com/en-us/library/system.runtime.remoting.remotingexception.aspx) |
| [System.Runtime.Remoting.ServerException](http://msdn.microsoft.com/en-us/library/system.runtime.remoting.serverexception.aspx) |
| [System.Runtime.Serialization.SerializationException](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.serializationexception.aspx) |
| [System.Security.Authentication.AuthenticationException](http://msdn.microsoft.com/en-us/library/system.security.authentication.authenticationexception.aspx) |
| [System.Security.Cryptography.CryptographicException](http://msdn.microsoft.com/en-us/library/system.security.cryptography.cryptographicexception.aspx) |
| [System.Security.HostProtectionException](http://msdn.microsoft.com/en-us/library/system.security.hostprotectionexception.aspx) |
| [System.Security.Policy.PolicyException](http://msdn.microsoft.com/en-us/library/system.security.policy.policyexception.aspx) |
| [System.Security.Principal.IdentityNotMappedException](http://msdn.microsoft.com/en-us/library/system.security.principal.identitynotmappedexception.aspx) |
| [System.Security.SecurityException](http://msdn.microsoft.com/en-us/library/system.security.securityexception.aspx) |
| [System.Security.VerificationException](http://msdn.microsoft.com/en-us/library/system.security.verificationexception.aspx) |
| [System.Security.XmlSyntaxException](http://msdn.microsoft.com/en-us/library/system.security.xmlsyntaxexception.aspx) |
| [System.ServiceModel.CommunicationException](http://msdn.microsoft.com/en-us/library/system.servicemodel.communicationexception.aspx) |
| [System.ServiceModel.Dispatcher.InvalidBodyAccessException](http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.invalidbodyaccessexception.aspx) |
| [System.ServiceModel.Dispatcher.MultipleFilterMatchesException](http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.multiplefiltermatchesexception.aspx) |
| [System.ServiceModel.InvalidMessageContractException](http://msdn.microsoft.com/en-us/library/system.servicemodel.invalidmessagecontractexception.aspx) |
| [System.ServiceModel.QuotaExceededException](http://msdn.microsoft.com/en-us/library/system.servicemodel.quotaexceededexception.aspx) |
| [System.ServiceProcess.TimeoutException](http://msdn.microsoft.com/en-us/library/system.serviceprocess.timeoutexception.aspx) |
| [System.StackOverflowException](http://msdn.microsoft.com/en-us/library/system.stackoverflowexception.aspx) |
| [System.Threading.AbandonedMutexException](http://msdn.microsoft.com/en-us/library/system.threading.abandonedmutexexception.aspx) |
| [System.Threading.SemaphoreFullException](http://msdn.microsoft.com/en-us/library/system.threading.semaphorefullexception.aspx) |
| [System.Threading.SynchronizationLockException](http://msdn.microsoft.com/en-us/library/system.threading.synchronizationlockexception.aspx) |
| [System.Threading.ThreadAbortException](http://msdn.microsoft.com/en-us/library/system.threading.threadabortexception.aspx) |
| [System.Threading.ThreadInterruptedException](http://msdn.microsoft.com/en-us/library/system.threading.threadinterruptedexception.aspx) |
| [System.Threading.ThreadStartException](http://msdn.microsoft.com/en-us/library/system.threading.threadstartexception.aspx) |
| [System.Threading.ThreadStateException](http://msdn.microsoft.com/en-us/library/system.threading.threadstateexception.aspx) |
| [System.TimeoutException](http://msdn.microsoft.com/en-us/library/system.timeoutexception.aspx) |
| [System.Transactions.TransactionException](http://msdn.microsoft.com/en-us/library/system.transactions.transactionexception.aspx) |
| [System.TypeInitializationException](http://msdn.microsoft.com/en-us/library/system.typeinitializationexception.aspx) |
| [System.TypeLoadException](http://msdn.microsoft.com/en-us/library/system.typeloadexception.aspx) |
| [System.TypeUnloadedException](http://msdn.microsoft.com/en-us/library/system.typeunloadedexception.aspx) |
| [System.UnauthorizedAccessException](http://msdn.microsoft.com/en-us/library/system.unauthorizedaccessexception.aspx) |
| [System.UriTemplateMatchException](http://msdn.microsoft.com/en-us/library/system.uritemplatematchexception.aspx) |
| [System.Web.Caching.DatabaseNotEnabledForNotificationException](http://msdn.microsoft.com/en-us/library/system.web.caching.databasenotenabledfornotificationexception.aspx) |
| [System.Web.Caching.TableNotEnabledForNotificationException](http://msdn.microsoft.com/en-us/library/system.web.caching.tablenotenabledfornotificationexception.aspx) |
| [System.Web.Management.SqlExecutionException](http://msdn.microsoft.com/en-us/library/system.web.management.sqlexecutionexception.aspx) |
| [System.Web.Services.Protocols.SoapException](http://msdn.microsoft.com/en-us/library/system.web.services.protocols.soapexception.aspx) |
| [System.Windows.Automation.ElementNotAvailableException](http://msdn.microsoft.com/en-us/library/system.windows.automation.elementnotavailableexception.aspx) |
| [System.Windows.Data.ValueUnavailableException](http://msdn.microsoft.com/en-us/library/system.windows.data.valueunavailableexception.aspx) |
| [System.Windows.Markup.XamlParseException](http://msdn.microsoft.com/en-us/library/system.windows.markup.xamlparseexception.aspx) |
| [System.Windows.Media.Animation.AnimationException](http://msdn.microsoft.com/en-us/library/system.windows.media.animation.animationexception.aspx) |
| [System.Windows.Media.InvalidWmpVersionException](http://msdn.microsoft.com/en-us/library/system.windows.media.invalidwmpversionexception.aspx) |
| [System.Workflow.Activities.EventDeliveryFailedException](http://msdn.microsoft.com/en-us/library/system.workflow.activities.eventdeliveryfailedexception.aspx) |
| [System.Workflow.Activities.WorkflowAuthorizationException](http://msdn.microsoft.com/en-us/library/system.workflow.activities.workflowauthorizationexception.aspx) |
| [System.Workflow.Runtime.Hosting.PersistenceException](http://msdn.microsoft.com/en-us/library/system.workflow.runtime.hosting.persistenceexception.aspx) |
| [System.Workflow.Runtime.Tracking.TrackingProfileDeserializationException](http://msdn.microsoft.com/en-us/library/system.workflow.runtime.tracking.trackingprofiledeserializationexception.aspx) |
| [System.Xml.Schema.XmlSchemaException](http://msdn.microsoft.com/en-us/library/system.xml.schema.xmlschemaexception.aspx) |
| [System.Xml.XmlException](http://msdn.microsoft.com/en-us/library/system.xml.xmlexception.aspx) |
| [System.Xml.XPath.XPathException](http://msdn.microsoft.com/en-us/library/system.xml.xpath.xpathexception.aspx) |
| [System.Xml.Xsl.XsltException](http://msdn.microsoft.com/en-us/library/system.xml.xsl.xsltexception.aspx) |

Let's take a look at a couple of the more common exception classes that we have been using and some of the various exceptions that they can throw. Please note that these are not exhaustive lists – they are just provided to show you some of the latitude covered by the exception classes.

List of exceptions thrown by System.ArithmeticException:

|  |
| --- |
| [System.DivideByZeroException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdividebyzeroexceptionclasstopic.htm) |
| [System.NotFiniteNumberException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemnotfinitenumberexceptionclasstopic.htm) |
| [System.OverflowException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemoverflowexceptionclasstopic.htm) |

List of exceptions thrown by System.IO.IOException:

|  |
| --- |
| [System.IO.DirectoryNotFoundException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemiodirectorynotfoundexceptionclasstopic.htm) |
| System.IO.DriveNotFoundException |
| [System.IO.EndOfStreamException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemioendofstreamexceptionclasstopic.htm) |
| [System.IO.FileLoadException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemiofileloadexceptionclasstopic.htm) |
| [System.IO.FileNotFoundException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemiofilenotfoundexceptionclasstopic.htm) |
| [System.IO.PathTooLongException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemiopathtoolongexceptionclasstopic.htm) |
| System.IO.PipeException |

[Visual Basic]

List of exceptions thrown by System.Data.DataException:

|  |
| --- |
| [System.Data.ConstraintException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdataconstraintexceptionclasstopic.htm) |
| [System.Data.DeletedRowInaccessibleException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatadeletedrowinaccessibleexceptionclasstopic.htm) |
| [System.Data.DuplicateNameException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdataduplicatenameexceptionclasstopic.htm) |
| System.Data.EntityException |
| [System.Data.InRowChangingEventException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatainrowchangingeventexceptionclasstopic.htm) |
| System.Data.InvalidCommandTreeException |
| [System.Data.InvalidConstraintException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatainvalidconstraintexceptionclasstopic.htm) |
| [System.Data.InvalidExpressionException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatainvalidexpressionexceptionclasstopic.htm) |
| [System.Data.MissingPrimaryKeyException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatamissingprimarykeyexceptionclasstopic.htm) |
| [System.Data.NoNullAllowedException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatanonullallowedexceptionclasstopic.htm) |
| [System.Data.ReadOnlyException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatareadonlyexceptionclasstopic.htm) |
| [System.Data.RowNotInTableException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatarownotintableexceptionclasstopic.htm) |
| [System.Data.StrongTypingException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatastrongtypingexceptionclasstopic.htm) |
| [System.Data.TypedDataSetGeneratorException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdatatypeddatasetgeneratorexceptionclasstopic.htm) |
| System.Data.UpdateException |
| [System.Data.VersionNotFoundException](file:///C:\Users\Scott%20D.%20James\Documents\svsu\CIS%20311\frlrfsystemdataversionnotfoundexceptionclasstopic.htm) |

You would obviously need to look up some details on the use of each of the exception classes provided above. As you can see, there is already a lot of stuff built in to the .NET framework that you can immediately catch with very little work. Also, remember that some exception classes contain a single exception, while others host a family of related exceptions.

# Creating Your Own Custom Exceptions

We have one more thing that we need to look at regarding exceptions and that is how to create your own custom exceptions and throw them. You can design your own classes that will throw any exceptions that you create (yes, I know we haven't talked about these in detail yet – we will be examining object-orientation in detail in Chapter 16). This is a great way to extend the usefulness of the code you write and keep it consistent with the .NET architecture.

Let's write an application in which we create an Invalid ZipCode Exception. Assuming that our program somehow receives an invalid ZipCode, we would want to throw an InvalidZipCodeException so that we could do whatever processing we need to do when an invalid zip code is encountered. Here's the code for this program:

'Chapter 14 -- Program 9

Public Class Form1

'This is our normal form code. We are having a great

'day in program execution land, handling addresses - thousands

'of addresses. Every time a new address comes along, this

'command button click routine gets fired and it validates the

'address -- maybe it looks up the address' zip code against

'a USPS online service...

Private Sub cmdValidateZipCode\_Click(sender As Object, e As EventArgs)

Handles cmdValidateZipCode.Click

Dim strZipCode As String = "4841X" 'Current address' zip code

Dim blnZipCodeOkay As Boolean

'This time though, there's a bad zip code...

Try

'See if we can validate the zip code

blnZipCodeOkay = IsNumeric(strZipCode)

If Not blnZipCodeOkay Then

'If the zip code is bad, let the user know.

MessageBox.Show("Validate Address GUI: " & vbCrLf &

"Oops! We got a bad zip code...")

'Also throw a ZipCode exception. Keep in mind that

'you need to pass the bad ZipCode value as an argument

'to the exception. In the U.S., no alphabetic characters

'can legally be in a valid ZipCode, so we have a problem.

Throw New InvalidZipCodeException("4841X")

End If

'You can catch custom exceptions exactly the same way that

'you catch the built in exceptions in VB.

Catch Ex As InvalidZipCodeException

MessageBox.Show("Custom Exception Thrown: " & Ex.Message)

End Try

End Sub

End Class

'This is where we define our custom exception.

'It will be in a class called InvalidZipCodeException

Public Class InvalidZipCodeException

'We have to base our exception class off of the parent of all

'exception classes which is System.Exception

Inherits System.Exception

'This is the constructor for our class, which expects a string

'to be passed to it. The string will contain the value of the

'invalid ZipCode

Sub New(ByVal strMessage As String)

'We have to make sure to call our parent's constructor. If

'we look up the Exception class in the MSDN help, we see

'that the constructor for Exception requires a string to

'be passed to it that contains a description of the

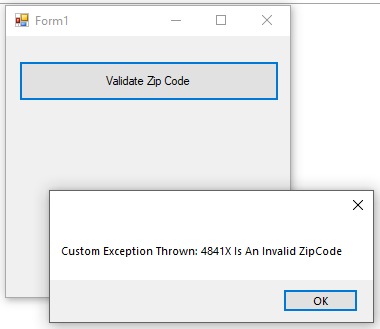
'exception, so here we go:

MyBase.New(strMessage & " Is An Invalid ZipCode")

End Sub

End Class

The next screenshot shows the Custom Exception being caught by our Custom Exception handler:



From this chapter, you can see that catching errors and handling them inside your own program is neither difficult nor time consuming to do. A well written application plans on things going wrong and has defensive code written to check for these issues and deal with them. You should now add error/exception handling code in every VB application that you write to make your application's users experience the best that it can be.

Let’s look at some of the more important properties of the System.Exception class which all Exceptions are based upon:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Data | A collection providing additional exception-specific information |
| InnerException | If an exception is a side effect of another error, the original error appears here |
| Message | A description of the error |
| Source | The name of the application or object that caused the error |
| StackTrace | A string fully documenting the list of all active procedure calls that led to the error causing statement |
| TargetSite | The name of the method that triggered the error |

Finally, just to be complete, I’d like to look at the Err (remember this is the old pre-.NET) object’s methods and properties that are relevant to us. Again, this is especially important to understand in porting old code:

|  |  |
| --- | --- |
| *Item* | *Purpose* |
| Clear | This method resets the Err object back to its default values. Usually we use Err to get the details of an error, but we can also initiate an error using the Raise method |
| Description | A property containing a text description of the error |
| ErrorToString | A method that returns the error message associated with a numeric system error code |
| IsError | A method that returns True if the supplied object argument is a System.Exception derived object |
| Number | A property containing the numeric code for the active error |
| Raise | This method can be called to generate a runtime error with the Err object |
| Source | This property contains the name of the application, class or object that generated the error |

# The Using Statement

This statement, while not directly error related, can help reduce the number of errors in your Visual Basic programs. Basically, the Using statement creates a new instance of an object when you call it, you use the object and when the accompanying End Using statement is encountered, the object is automatically disposed of, freeing up resources.

Here’s a sample program that illustrates the use of the Using statement wrapped around a OpenFileDialog instance and a TextReader instance. When the Command button is pressed, this code will allow the user to select a file via an OpenFileDialog and then read the file’s contents into a ListBox:

'Chapter 14 - Program 10

Public Class Form1

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

Dim strFilename As String

'Create an OpenFileDialog instance

Using myOpenFileDialog As New OpenFileDialog

myOpenFileDialog.ShowDialog()

'What would happen if this went after End Using?

strFilename = myOpenFileDialog.FileName

'We are going to hit the End Using, so we will

'automatically dispose of the OpenFileDialog instance

End Using

If strFilename <> "" Then

ReadFile(strFilename)

End If

End Sub

Private Sub ReadFile(ByVal strFilename As String)

'Create a TextReader instance

Using reader As System.IO.TextReader =

System.IO.File.OpenText(strFilename)

'Do what you want with it...

Dim line As String

line = reader.ReadLine()

Do Until line Is Nothing

ListBox1.Items.Add(line)

line = reader.ReadLine()

Loop

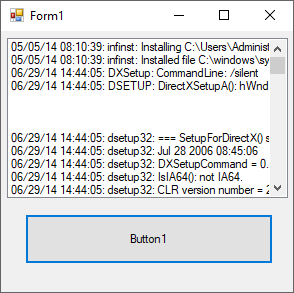
'Dispose of the TextReader instance

End Using

End Sub

End Class

Here’s the output from the program:



The only caveat on using the Using statement is that whatever objects are listed after Using, each must implement the IDisposable inteface – check the MSDN help if in doubt, but a good majority of the built in objects already do implement that interface!