CS608 Lecture Notes

Visual Basic.NET Programming

Object-Oriented Programming

Classes & Objects – Advanced Topics Part III (Error Handling)

(Part III of III)

(Lecture Notes 2C)

Prof. Abel Angel Rodriguez

CHAPTER 7 CLASSES & OBJECTS -ADVANCED CONCEPTS (CONT)....................................... 3

7.1 Error Handling in VB.NET.. **3**
7.1.1 General Discussion on Errors.. 3
Syntax Errors 3
Logical Errors 3
Run-Time Errors 3

7.2 Exception in VB.NET **6**
7.2.1 Exceptions Basics in VB.NET 6
The Exit Try Statement 8
7.2.2 The VB.NET Exception Class 10
Introduction to the VB.NET Exception Class... 10
Trapping all Errors using the VB.NET Exception Class.. 11
7.2.3 Trapping for Specific Error Using other Exception Classes Provided by VB.NET 15
Trapping for Specific Error using the Exception Classes 16
7.2.4 Trapping for Many Error using Multiple Catch Blocks 22
7.2.5 The Throw Statement.. 24
Error Handling Summarized 24
Raising our own errors.. 24
Using the Throw Statement.. 25

7.3 Shared Methods **30**
7.3.1 Overview & Discussion 30
7.3.2 Shared Methods 31

7.4 Shared Variables.. **32**
7.4.1 Overview.. 32

Chapter 7 Classes & Objects -Advanced Concepts (Cont)

7.1 Error Handling in VB.NET

7.1.1 General Discussion on Errors

- In our review of VB.NET we discussed the various types of errors that can occur in a program.
- There are three varieties of programming errors:

  1. Syntax
  2. Logical
  3. Run-Time

Syntax Errors

- **Syntax Errors –** Errors generated due to the code not following the rules of the language
  - These errors usually involve improper punctuation, spelling or format
  - These problems are easier to solve since the compiler identifies the erroneous statement by displaying a **BLUE Wiggly Line** under the incorrect VB.NET Statement.
  - To resolve this problem, you just have to review the rules of the language for that statement and make sure you are writing the statement properly

Logical Errors

- **Logical Error –** The program is not doing what is supposed to do. .
  - The algorithm fails to solve the problem the program was written to resolve
  - These problems are very difficult to solve since you need to re-think and go back to the planning phase and review the Algorithm.

Run-Time Errors

- **Run Time Errors –** These are programs that occur when the program is executing. The program passed the syntax or compiler test but fails during execution. .
  - These errors can be caused by the user entering a wrong data type via keyboard or performing some illegal operation, program logic not doing what is supposed to do, or system errors.
  - These problems are usually caused by improper arithmetic execution, attempting to access recourses that are not available etc
  - These problems are difficult to solve since they only show up when the program runs.

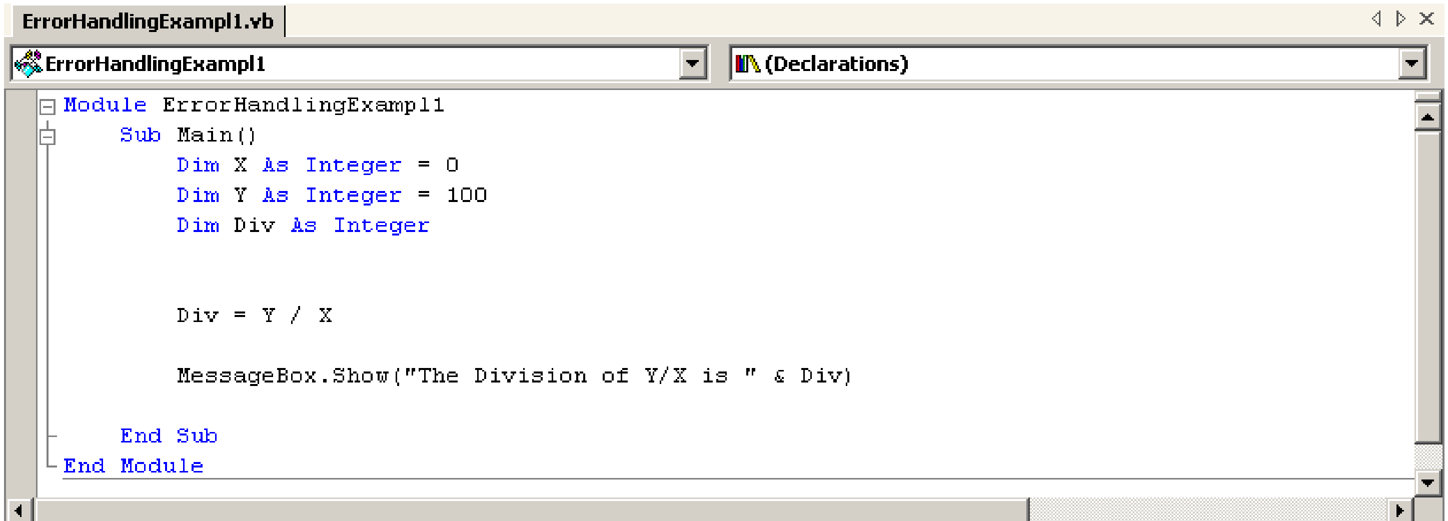
  - When a run-time error is generated, **the built-in default Error handling mechanism of VB.NET will trap the error and terminate the execution of the program**.

  - **It is the responsibility of the programmer to provide a alternate mechanism to trap or handle these errors in order to prevent the default VB.NET mechanism from terminating the program execution**.

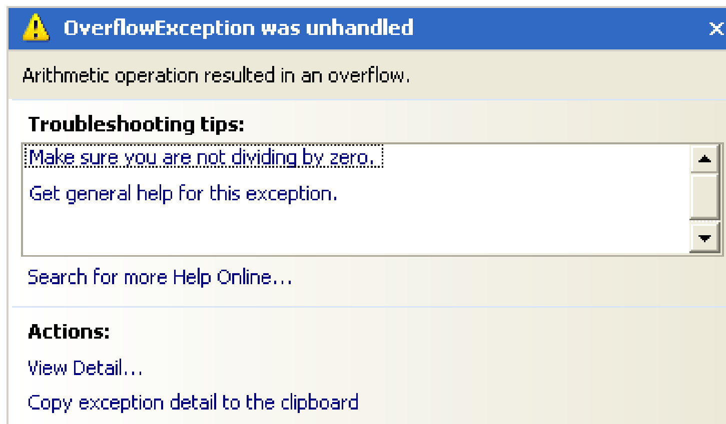
- Run time errors can be a **nuisance** since the default error mechanism of VB.NET cause the program execution to stop.
- Once execution is halted, VB.NET will display a message indicating the nature of the error.

Example 7-1:

- For example the code below attempts to divide a number by 0:



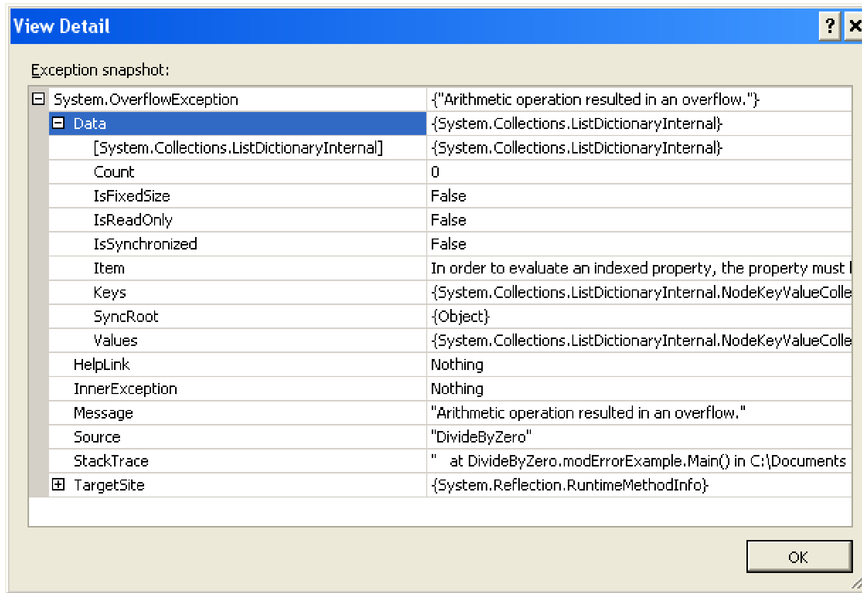
- We know this is illegal and will cause a problem. Once this error is encountered, the program execution will stop and the following message is displayed:



- At this point the execution of the program will halt. You have the following choices:

  - **Troubleshooting Tips –** Access to Microsoft on-line help to get information on these types of errors.
  - **Get general Help for this exception –** Access to additional on-line help to get information on these types of errors.
  - **Search for more Help Online –** Additional help Online.

- **View Details –** View details on the actual error, such as data type, location of program etc..



- In summary, if the programmer does not provide a mechanism to handle such errors during run-time, then the program will halt and display the message to the user. The user of the application will have to make these decisions and obviously a user won't know what to do at this point.
- Also, not handling errors will make our programs unprofessional.

7.2 Exception in VB.NET

7.2.1 Exceptions Basics in VB.NET

- Once again:

  - **IT IS THE RESPONSIBILITY OF THE PROGRAMMER TO ADD THE REQUIRED ERROR HANDLING MECHANISM TO A PROGRAM IN ORDER TO PREVENT THE BUILT-IN VB.NET ERROR HANDLING MECHANISM FROM STOPING THE EXECUTION OF A PROGRAM DURING AN ERROR.**

- Exceptions handling is the mechanism provided by **VB.NET** to allow programmers to trap **Run-time** errors.
- An Exception is an anomaly or error that occurs during the execution of a program. This error can be caused by the user, logic or simply the system itself.
- VB.NET provides the following Keywords to handle Exceptions:

  - Try
  - Catch
  - Finally
  - Exit Try
  - Throw

- The general form to a Try –catch-Finally block is as follows:

```
'General Try-Catch-Finally Block Syntax:
Try

        'Code Statement which can cause the exception
        'Your regular code goes here!

Catch
        '[Optional} Code Statement for handling the exception
        'You can have as many Catch statements as necessary to handle the exceptions
        'Also we can specify what type of error to look for

Finally

        '[Optional] Code that will always executed regardless of whether and exception is caught

End Try
```

- The idea is as follows:

  - Within the **Try Block** statement, you place the code where you think an exception can occur.
  - If any exception or error occurs inside the try block, the control of the program is transferred to the appropriate **Catch block** (Remember there can be more than one Catch Block) to handle or take care of the exception.
  - After the Catch block handles the exception, control is transferred to the **Finally Block**, which will always executes by default.

- Keep the following rules in mind when working with exceptions:

  - Both catch and finally blocks are optional.
  - The Catch Block is where we handle the error. VB.NET provides various error definitions that we can catch using the Catch with parameters for the unique error we are trapping for. Or simply trap all errors using the Catch block with no parameters.
  - The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks.

- If no exception occurred inside the try block, the control directly transfers to finally block. The statements inside the finally block are executed always.
- Note that it is an error to transfer control out of a finally block by using the keywords **break**, **continue**, **return** or **goto**

Example 7-2:

- Let's look at the previous example where we tried to divide a number by 0 and generated the default VB.NET error mechanism which stopped execution of the program. In this case we will add our own error handling mechanism via the Try-Catch-Finally Block statement to handle the error ourselves:

ErrorHandlingExampl1.vb*

ErrorHandlingExampl1 (Declarations)

```
    Sub Main()
        Dim X As Integer = 0
        Dim Y As Integer = 100
        Dim Div As Integer

        'Set the Div variable to a Default value for example purpose only
        Div = -1

        'Begin Error trapping Section
        Try

            Div = Y / X
            MessageBox.Show("The Division of Y/X is " & Div)

        Catch
            MessageBox.Show("Exception was raised")
            MessageBox.Show("Code Inside the Catch Statement is executing")
        Finally
            MessageBox.Show("Code Inside the Finally Statement is executing")

            MessageBox.Show("The Division of Y/X is " & Div)

        End Try  'End error trapping section

    End Sub
End Module
```

Discussion of Example:

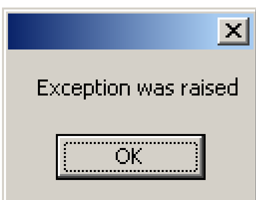
- This example results in the following:

  - We set Div = -1 simply to show at the end that this variable was not set since the error handling routine prevented the computation by 0 to take place.
  - We begin the error trapping section by adding the code which we suspect that will generate error inside the **Try block**:

```
Div = Y / X
MessageBox.Show("The Division of Y/X is " & Div)
```

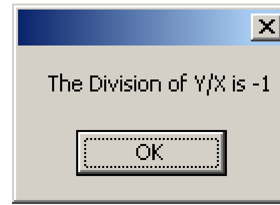
  - When the program attempts to execute this code, an error is generated and the **Catch block** code is executed to handle the error:

```
MessageBox.Show("Exception was raised")
MessageBox.Show("Code Inside the Catch Statement is executing")
```



- Once this code is executed, control is passed to the Finally Block for any cleanup code to be executed.

```
MessageBox.Show("Code Inside the Finally Statement is executing")
MessageBox.Show("The Division of Y/X is " & Div)
```



- Note that we once again display the value of Div. and its content is -1 since it was never modified.

❖ The important point here is that **THE PROGRAM DID NOT STOP RUNNING AND END UNEXPECTALLY**. The built-in VB.NET error handling mechanism did not stop the program but simply passed controlled to our error handling mechanism, the Catch & Finally blocks for execution of the code inside the these blocks.

The Exit Try Statement

- The **Exit Try** Statement will allow you to break out of the **Try** or **Catch** Block and continue to the **Finally** block.
- This statement gives us flexibility to exit under certain conditions if we wish to.
- Lets look at the previous example modified to exit out of the Catch Block under a certain condition:

Example 7-3:

- Modify the Example 7-2 to exit out of the Catch Block when a divide by zero error is trapped:

```
ErrorHandlingExampl2.vb
ErrorHandlingExample2                Main

Module ErrorHandlingExample2
    Sub Main()
        Dim X As Integer = 0
        Dim Y As Integer = 100
        Dim Div As Integer

        'Set the Div variable to a Default value for example purpose only
        Div = -1

        'Begin Error trapping Section
        Try
            Div = Y / X
            MessageBox.Show("The Division of Y/X is " & Div)
        Catch
            If X = 0 Then
                MessageBox.Show("Caught a divide by zero error")
                Exit Try
            Else
                MessageBox.Show("Error Not divide by 0")
            End If
        Finally
            MessageBox.Show("Code Inside the Finally Statement is executing")
            MessageBox.Show("The Division of Y/X is " & Div)

        End Try  'End error trapping section

    End Sub
End Module
```

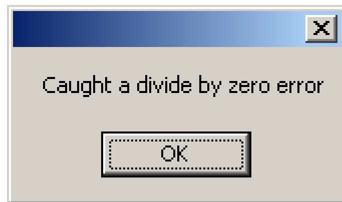
Discussion of Example:

- This example results in the following:

  - In the Catch block we test to verify if the error was a divide by 0 error, if so we break out of the Catch Block, other wise we inform the user that is not a divide by zero error:

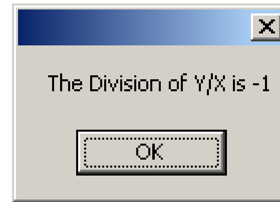
```
If X = 0 Then
    MessageBox.Show("Caught a divide by zero error")
    Exit Try
Else
    MessageBox.Show("Error Not divide by 0")
End If
```

  - Of course since we are always dividing by zero in this example the result will always be:



  - When we break out of the Catch block, control is passed to the Finally Block for any cleanup code to be executed.

```
MessageBox.Show("Code Inside the Finally Statement is executing")
MessageBox.Show("The Division of Y/X is " & Div)
```



  - Note that we once again display the value of Div. and its content is -1 since it was never modified.

7.2.2 The VB.NET Exception Class

- Now is when things get a bit more interesting.
- The Error handling features of VB.NET is powerful and offers a variety of features, but can be confusing.
- In this course we will touch upon the basics and try to keep it as simple as possible

Introduction to the VB.NET Exception Class

- In VB.NET an exception is really an *Object* which is either directly or indirectly derived from the **Exception Class**.
- In the ***System Namespace***, there is a built-in ***Exception Class***
- This is the Main Error handling Class from which all other ***Exception classes*** are derived from.
- Yes that is correct, all other ***Exception Classes***. What this means is that VB.NET provides a list of Exception classes each specifically design to trap a particular error.
- So in addition to the **System.Exception Class**, VB.NET provides many standard exception classes for us to use in order to trap for a particular exception.
- We can build our own custom Exception Classes based on the built-in Exception Classes.

- The **System.Exception Class** contains Properties and methods that we can access with information about the error.
- Note that the main **System.Exception Class** traps all possible errors.

  - The point here is that the Try-Block-Finally Statement really returned an Object of the Exception Class.
  - But using the Catch Block alone will not give us access to the object's properties and method; we would need to specify an Object of the exception class ourselves.

- Keeping in mind that an exception is really an object, lets look at another form or the syntax for the Try –catch-Finally block that is automatically generated when working with the VB.NET development environment:

```
'General Try-Catch-Finally Block Syntax:
Try
        'Code Statement which can cause the exception
        'Your regular code goes here!

Catch ex As Exception
        '[Optional} Code Statement for handling the exception
        'Exception is the main Exception class provided by VB.NET
        'ex is a variable referencing the  exception object, this is a variable so we can name it what ever we want.
        'You can have as many Catch statements as necessary to handle the exceptions

Finally

        '[Optional] Code that will always executed regardless of whether and exception is caught

End Try
```

- In this case, an object of the Exception Class is created in the Catch block.
- This allows us to now use some of the properties and methods of the Exception Class.
- Note that since all other exception classes are derived from the **System.Exception Class** this class traps for all errors.

- The Exception Class contains the following Properties & Methods:

Properties of the Exception Class:

Property	Description
HelpLink	String indicating the link to the help for this exception
Message	A string that contains the error
Source	A string containing the name of an object that generated the error
Other…….	I will not list the other properties. For further info. See VB.NET documentation.

Methods of the Exception Class:

Method	Description
GetType	This method returns an Object which contains additional information about the type of error.
ToString	Similar to the Message Property, but returns the a string containing the error and includes a lot more information about the error, such as path etc.
Other…….	I will not list the other methods. For further info. See VB.NET documentation.

Trapping all Errors using the VB.NET Exception Class

- Since the **Exception Class** is used as the main class from which all other Exceptions classes are generated from, the **System.Exception Class** traps all possible errors
- Let's look at some example of using the Exception class to trap all errors

Example 7-4: Trapping for all errors using object of Exception Class

- Let's look at the previous example but this time we will create an object of the exception class to trap all errors:

```
ErrorHandlingExampl3.vb
ErrorHandlingExample3          (Declarations)

Module ErrorHandlingExample3
    Sub Main()
        Dim X As Integer = 0
        Dim Y As Integer = 100
        Dim Div As Integer

        'Set the Div variable to a Default value for example purpose only
        Div = -1

        'Begin Error trapping Section
        Try
            Div = Y / X
            MessageBox.Show("The Division of Y/X is " & Div)

        Catch objException As System.Exception
            MessageBox.Show("Caught an Exception")

        Finally
            MessageBox.Show("Code Inside the Finally Statement is executing")
            MessageBox.Show("The Division of Y/X is " & Div)

        End Try  'End error trapping section

    End Sub
End Module
```

Discussion of Example:

- This example results in the following:

  - In this example, we create an object of the Exception Class in the **Catch block** to trap for any:

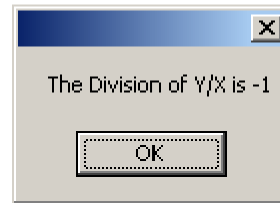
```
Catch objException As System.Exception
```

  - When the error is trapped the **Catch block** code is executed to handle the error:

```
MessageBox.Show("Caught an Exception")
```

  - Once this code is executed, control is passed to the Finally Block for any cleanup code to be executed.

```
MessageBox.Show("Code Inside the Finally Statement is executing")
MessageBox.Show("The Division of Y/X is " & Div)
```



  - At this point you may ask yourself what is the difference between our previous examples where we used the Catch Block statement alone:

```
Catch
MessageBox.Show("Exception was raised")
MessageBox.Show("Code Inside the Catch Statement is executing")
```

  And this version:

```
Catch objException As System.Exception
MessageBox.Show("Caught an Exception")
```

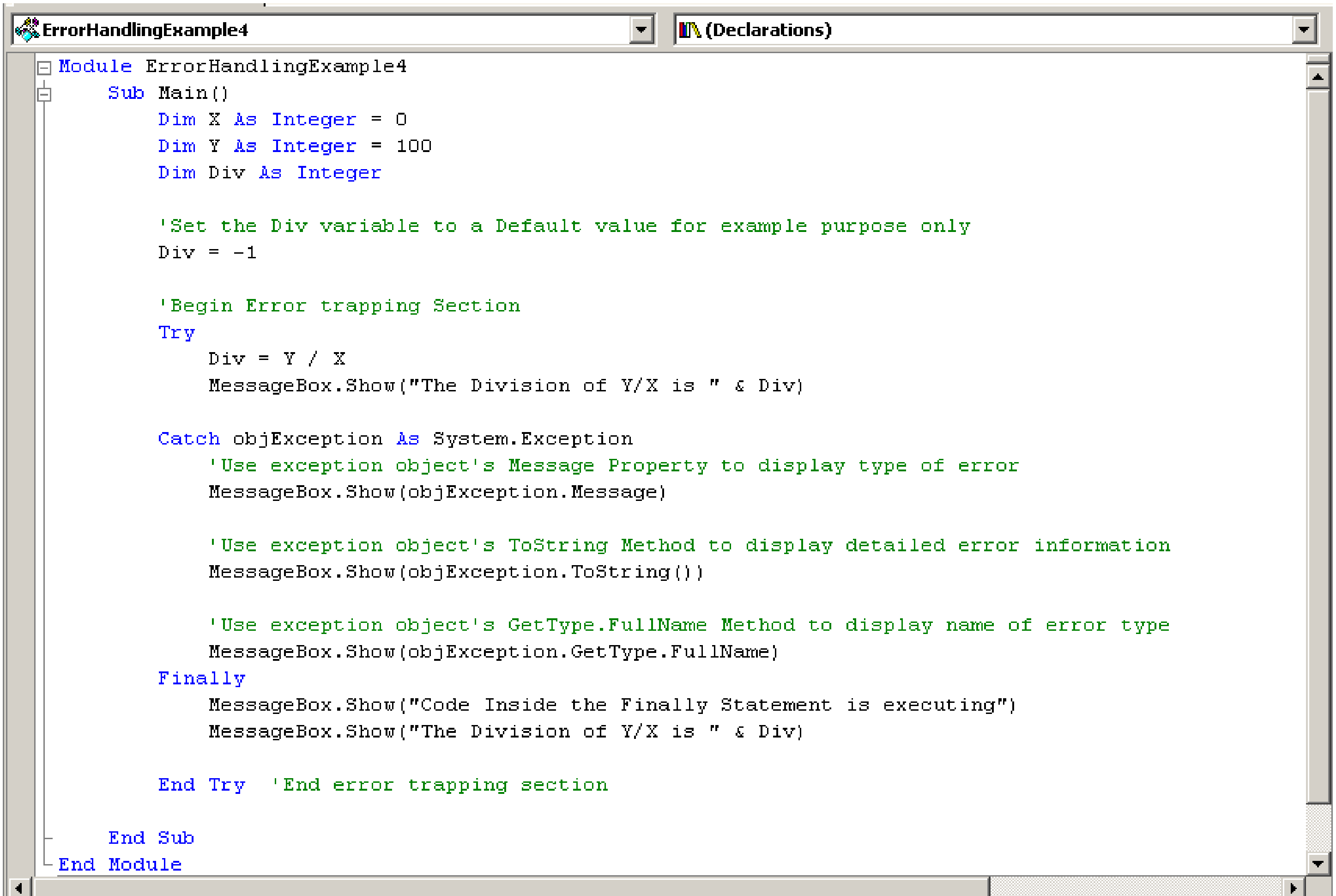
❖ The answer is NONE!! Both these code trap for all errors. The difference is that with the second version using the object, we can access the object's properties and methods. As shown in the next example.

Example 7-5: Trapping for all Errors using Object of Exception Class and accessing Properties & Methods

- Let's look at the previous example but this time we will create an object of the exception class to trap all errors and we will use the following properties and methods:

    - **Message** Property – Returns a string describing the type of error
    - **ToString()** Method – Returns a string describing the type of error and additional information such as path in code were error was generated from etc.
    - **GetType()** Method – Returns an object which contains more detailed information about the error such as the full name of the built-in VB.NET class for this particular error.

- The code is as follows:



```
ErrorHandlingExample4                    (Declarations)

Module ErrorHandlingExample4
    Sub Main()
        Dim X As Integer = 0
        Dim Y As Integer = 100
        Dim Div As Integer

        'Set the Div variable to a Default value for example purpose only
        Div = -1

        'Begin Error trapping Section
        Try
            Div = Y / X
            MessageBox.Show("The Division of Y/X is " & Div)

        Catch objException As System.Exception
            'Use exception object's Message Property to display type of error
            MessageBox.Show(objException.Message)

            'Use exception object's ToString Method to display detailed error information
            MessageBox.Show(objException.ToString())

            'Use exception object's GetType.FullName Method to display name of error type
            MessageBox.Show(objException.GetType.FullName)
        Finally
            MessageBox.Show("Code Inside the Finally Statement is executing")
            MessageBox.Show("The Division of Y/X is " & Div)

        End Try  'End error trapping section

    End Sub
End Module
```

Discussion of Example:

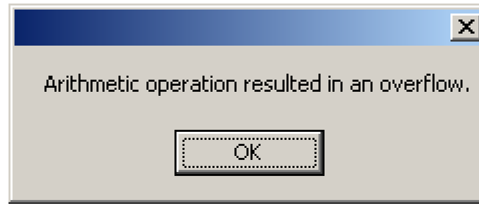
- This example results in the following:

    - In this example, we create an object of the Exception Class in the **Catch block** to trap for any:

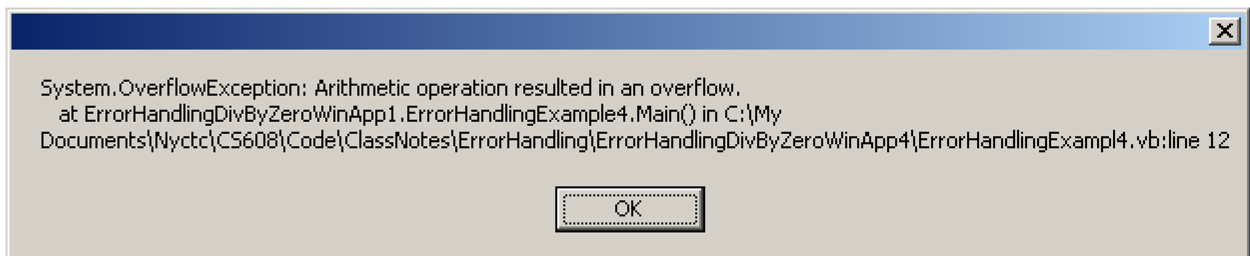
```
Catch objException As System.Exception
```

- When the error is trapped the **Catch block** code is executed to handle the error, but in this case we use the properties of the Exception Object to describe the error:

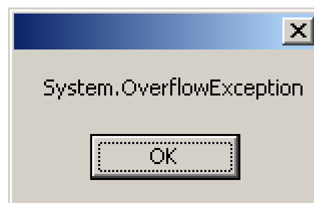
```
'Use exception object's Message Property to display type of error
MessageBox.Show(objException.Message)
```




```
'Use exception object's ToString Method to display detailed error information
MessageBox.Show(objException.ToString())
```

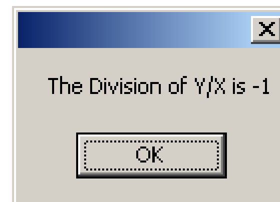



```
'Use exception object's GetType.FullName Method to display name of error type
MessageBox.Show(objException.GetType.FullName)
MessageBox.Show("Caught an Exception")
```



- Once this code is executed, control is passed to the Finally Block for any cleanup code to be executed.

```
MessageBox.Show("Code Inside the Finally Statement is executing")
MessageBox.Show("The Division of Y/X is " & Div)
```



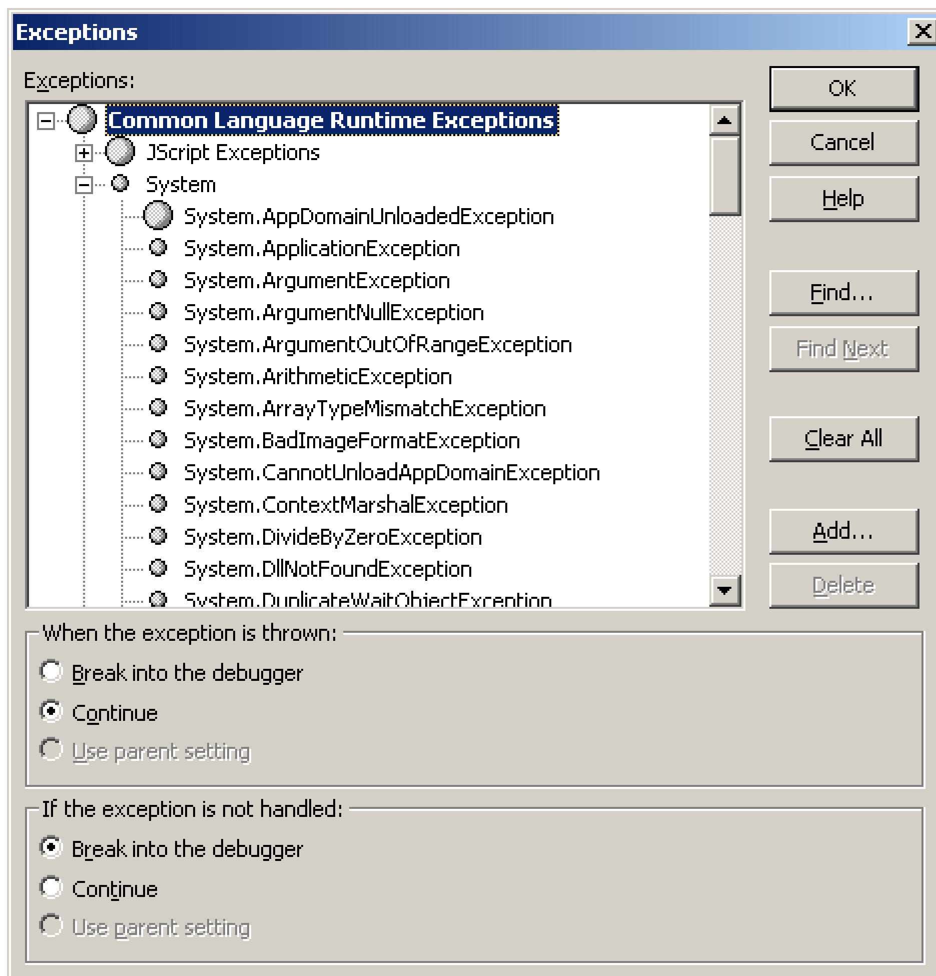
❖ In this example we see that we were not only able to trap for all errors but we were able to use the properties and methods of the Exception Class for information about the error or to perform what ever functionality is available for us by the Exception Class.

7.2.3 Trapping for Specific Error Using other Exception Classes Provided by VB.NET

- In addition to the **System.Exception Class**, VB.NET provides many standard exception classes for us to use in order to trap for a specific error.
- In this case we may have code that performs some specific task and we suspect that this task may result in a specific error. We may want to trap for this error so that we can perform some corrective process etc.
- VB.NET provides a variety of Classes for this purpose. Note that all these classes are derived from the main Exception Class.
- Examples of such Exception Classes are as follows:

  - System.OutOfMemoryException
  - System.NullReferenceException
  - Syste.InvalidCastException
  - Syste.ArrayTypeMismatchException
  - System.IndexOutOfRangeException
  - System.ArithmeticException
  - System.DevideByZeroException
  - System.OverFlowException

- Each of these classes contains properties and methods for the specific error in which they were created to identify.
- You can view a listing of all the Exception Classes in the VB.NET documentation or using the IDE and the keyboard you can enter **Ctrl+Alt+E** to displays the Exceptions dialog box:



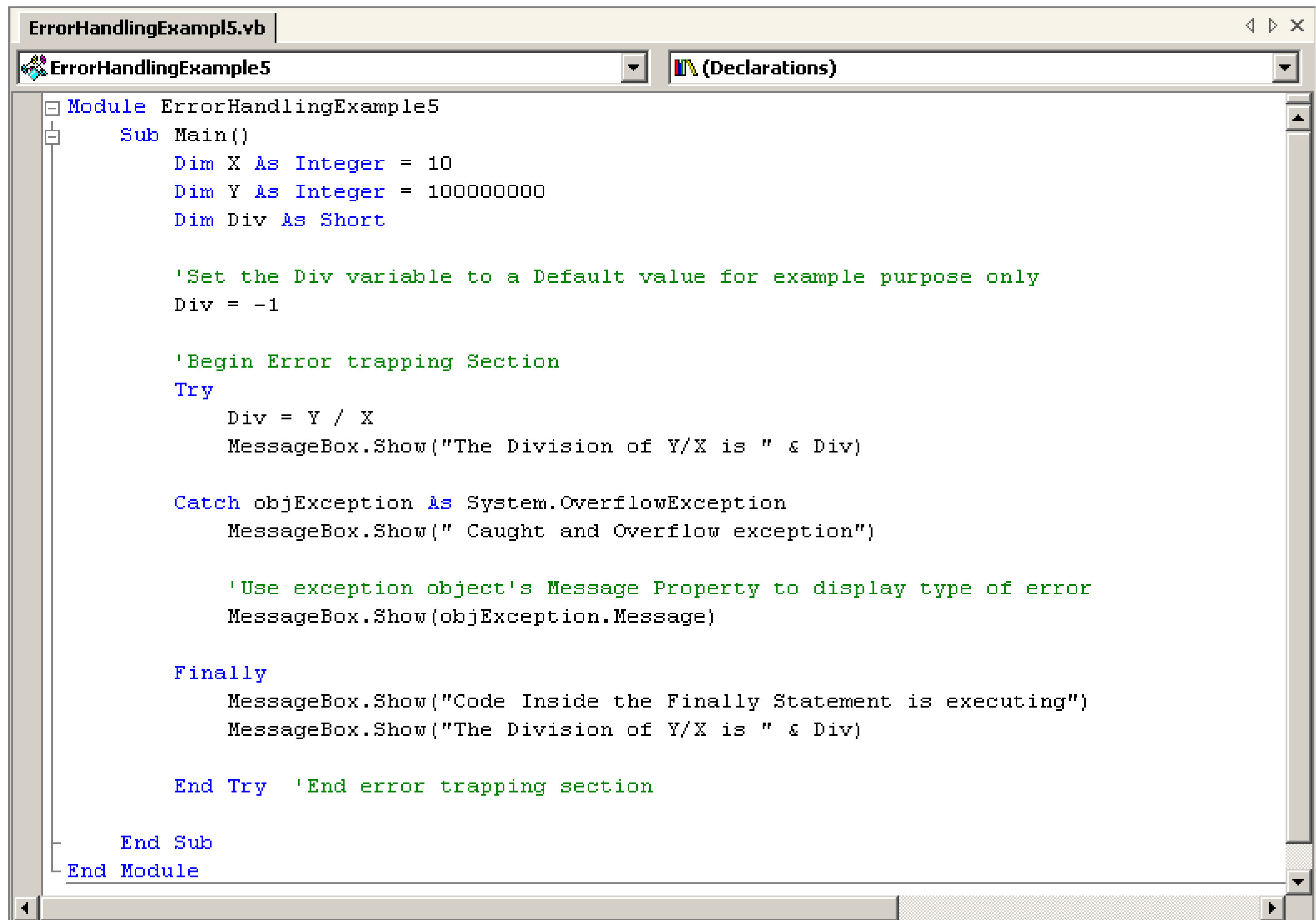
Trapping for Specific Error using the Exception Classes

- Let's look at some examples of using a particular Exception class to trap for a unique error

Example 7-5: Trapping for an Overflow Exception.

- An Overflow is when we perform a mathematical operation and the result is beyond the range of the data type.
- In this example we will divide two integer variables and place the result in a short variable, but the result of the division will exceed the limits of an short.
- We will use the **System.OverflowException** Class to trap for this specific error only.

- The code is as follows:




```
Module ErrorHandlingExample5
    Sub Main()
        Dim X As Integer = 10
        Dim Y As Integer = 100000000
        Dim Div As Short

        'Set the Div variable to a Default value for example purpose only
        Div = -1

        'Begin Error trapping Section
        Try
            Div = Y / X
            MessageBox.Show("The Division of Y/X is " & Div)

        Catch objException As System.OverflowException
            MessageBox.Show(" Caught and Overflow exception")

            'Use exception object's Message Property to display type of error
            MessageBox.Show(objException.Message)

        Finally
            MessageBox.Show("Code Inside the Finally Statement is executing")
            MessageBox.Show("The Division of Y/X is " & Div)

        End Try  'End error trapping section

    End Sub
End Module
```

Discussion of Example:

- This example results in the following:

    - In this example, we create three variables, where two are initialize with numbers and the third we will use to hold the result of a division. The third variable is of a short data type which is smaller than an integer:

```
Dim X As Integer = 10
Dim Y As Integer = 100000000
Dim Div As Short
```

- We begin the error trapping section by adding the code which we suspect that will generate error inside the **Try block**. Since we the result of this division is large and will be placed in a short variable, this will yield an error:

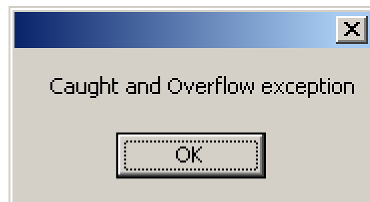
```
Try
Div = Y / X
MessageBox.Show("The Division of Y/X is " & Div)
```

- We then create an object of the **OverflowException Class** in the **Catch block** to trap for this specific type of error, where the result of an operation will overflow or exceed the limitation of the data type:

```
Catch objException As System.OverflowException
```

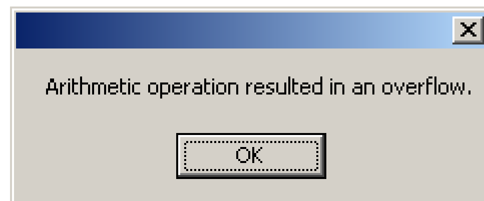
- In the **Catch block** when only this specific error is trapped, we display a message box indication that an exception was raised:

```
MessageBox.Show(" Caught and Overflow exception")
```



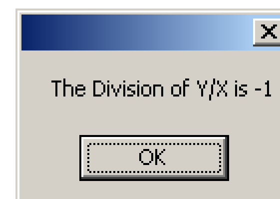
- In the **Catch block** we also use the *Message Property* of the Overflow Object to display the information about the exception:

```
'Use exception object's Message Property to display type of error
MessageBox.Show(objException.Message)
```



- Once this code is executed, control is passed to the Finally Block for any cleanup code to be executed.

```
MessageBox.Show("Code Inside the Finally Statement is executing")
MessageBox.Show("The Division of Y/X is " & Div)
```



- ❖ In this example we were able to trap for this specific error only.
- ❖ Note that if another error occurs, we are not trapping for it, therefore the built-in VB.NET error mechanism will take over and stop the program. Point is we need to be aware of this when trapping for specific errors.

Example 7-6: Trapping for a DivideByZero Exception.

- An **DivideByZero Exception** occurs when we attempt to divide a Decimal Number by 0. Note that this only occurs with a floating point decimal number, not the integer data type. If we divide by zero using the Integer data type the exception raised is the *OverflowException* not the *DivideByZeroException*.
- In this example we will divide two Decimal variables and place the result in another Decimal variable, but we will attempt to divide by 0.
- We will use the **`DivideByZeroException`** Class to trap for this specific error only.
- The code is as follows:

ErrorHandlingExampl6.vb

ErrorHandlingExample5 | (Declarations)

```
Module ErrorHandlingExample5
    Sub Main()
        Dim X As Decimal = 0
        Dim Y As Decimal = 100
        Dim Div As Decimal

        'Set the Div variable to a Default value for example purpose only
        Div = -1

        'Begin Error trapping Section
        Try
            Div = Y / X
            MessageBox.Show("The Division of Y/X is " & Div)

        Catch objException As System.DivideByZeroException
            MessageBox.Show(" Caught a divide by zero exception")

            'Use exception object's Message Property to display type of error
            MessageBox.Show(objException.Message)

        Finally
            MessageBox.Show("Code Inside the Finally Statement is executing")
            MessageBox.Show("The Division of Y/X is " & Div)

        End Try  'End error trapping section

    End Sub
End Module
```

Discussion of Example:

- This example results in the following:
    - In this example, we create three Decimal variables, where one is initialize with 0:

```
Dim X As Decimal = 0
Dim Y As Decimal = 100
Dim Div As Decimal
```

- We begin the error trapping section by adding the code which we suspect that will generate error inside the **Try block**:

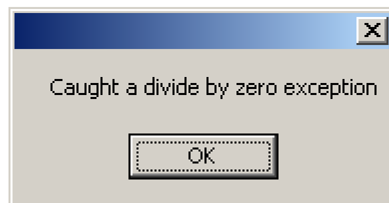
```
Try
Div = Y / X
MessageBox.Show("The Division of Y/X is " & Div)
```

- We then create an object of the **`DivideByZeroException`** **Class** in the **Catch block** to trap for this specific type of error, where the result of an operation will overflow or exceed the limitation of the data type:

```
Catch objException As System.DivideByZeroException
```

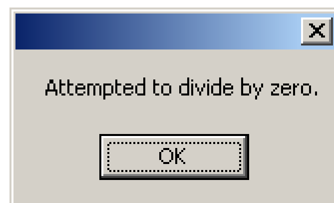
- In the **Catch block** when only this specific error is trapped, we display a message box indication that an exception was raised:

```
MessageBox.Show(" Caught a divide by zero exception")
```



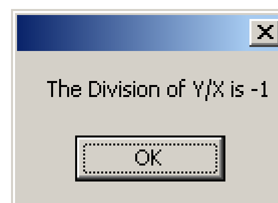
- In the **Catch block** we also use the *Message Property* of the Overflow Object to display the information about the exception:

```
'Use exception object's Message Property to display type of error
MessageBox.Show(objException.Message)
```



- Once this code is executed, control is passed to the Finally Block for any cleanup code to be executed.

```
MessageBox.Show("Code Inside the Finally Statement is executing")
MessageBox.Show("The Division of Y/X is " & Div)
```

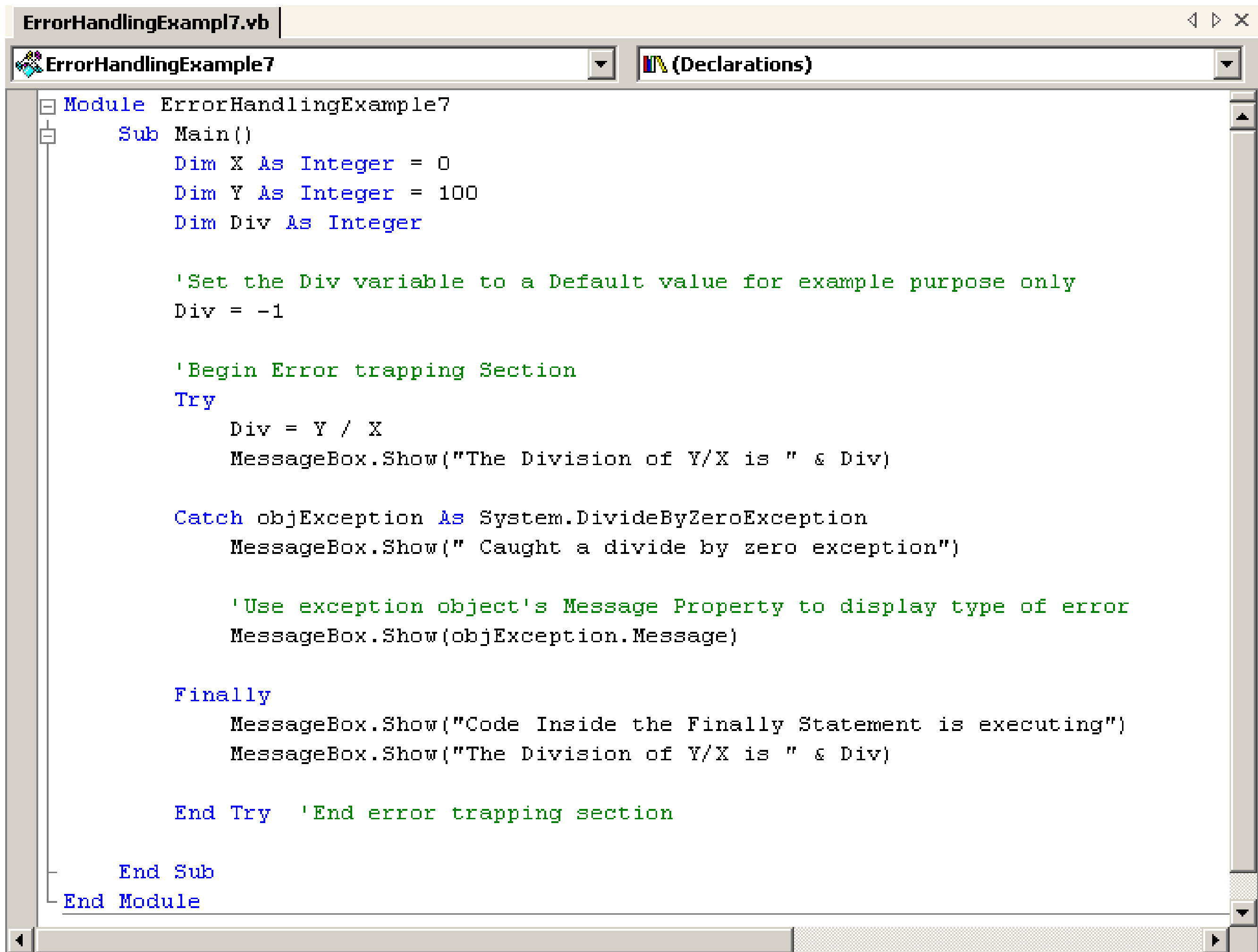


❖ In this example we were able to trap for the **`DivideByZeroException`** exception error only.

Example 7-7: Trapping for the wrong Exception.

- In this example we will trap for the wrong exception. We will make some simple modifications to Example 7.6 to demonstrate what happens when we trap for one specific exception and another exception is raised.
- As with the previous example we will trap for a **`DivideByZeroException.`** Remember this exception is raised when we attempt to divide a Decimal Number by 0. If we divide by zero using the Integer data type the exception raised is the *OverflowException* not the *DivideByZeroException*.
- Again we will use the **`DivideByZeroException`** Class to trap for this specific error only.

- The code is as follows:




```
Module ErrorHandlingExample7
    Sub Main()
        Dim X As Integer = 0
        Dim Y As Integer = 100
        Dim Div As Integer

        'Set the Div variable to a Default value for example purpose only
        Div = -1

        'Begin Error trapping Section
        Try
            Div = Y / X
            MessageBox.Show("The Division of Y/X is " & Div)

        Catch objException As System.DivideByZeroException
            MessageBox.Show(" Caught a divide by zero exception")

            'Use exception object's Message Property to display type of error
            MessageBox.Show(objException.Message)

        Finally
            MessageBox.Show("Code Inside the Finally Statement is executing")
            MessageBox.Show("The Division of Y/X is " & Div)

        End Try  'End error trapping section

    End Sub
End Module
```

Discussion of Example:

- This example results in the following:

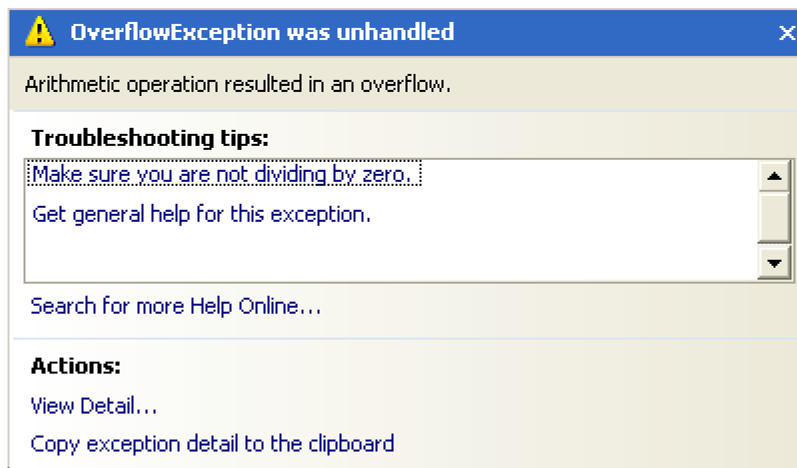
    - In this example, we modify the variables by making them integers and initialize them accordingly:

```
Dim X As Integer = 0
Dim Y As Integer = 100
Dim Div As Integer
```

- We begin the error trapping section by adding the code which we suspect that will generate error inside the **Try block** and the code that will handle the error in the **Catch Block** using the object of the **`DivideByZeroException`** **Class**:

```
Try
        Div = Y / X
MessageBox.Show("The Division of Y/X is " & Div)

Catch objException As System.DivideByZeroException

         MessageBox.Show(" Caught a divide by zero exception")

        'Use exception object's Message Property to display type of error
        MessageBox.Show(objException.Message)
```

- When we execute the program we get the following message:



- The code in the In the **Catch block** is never executed AND PROGRAM ENDS!!

❖ In this example we attempted to trap for the **`DivideByZeroException`** exception error only but what we generated was an Overflow Exception since when we divide an integer by zero the result is infinity which overflows the Div integer variable.

❖ The lesson here is that we need to be aware of the which error we are trapping for and make sure we take into account other errors that may occur.

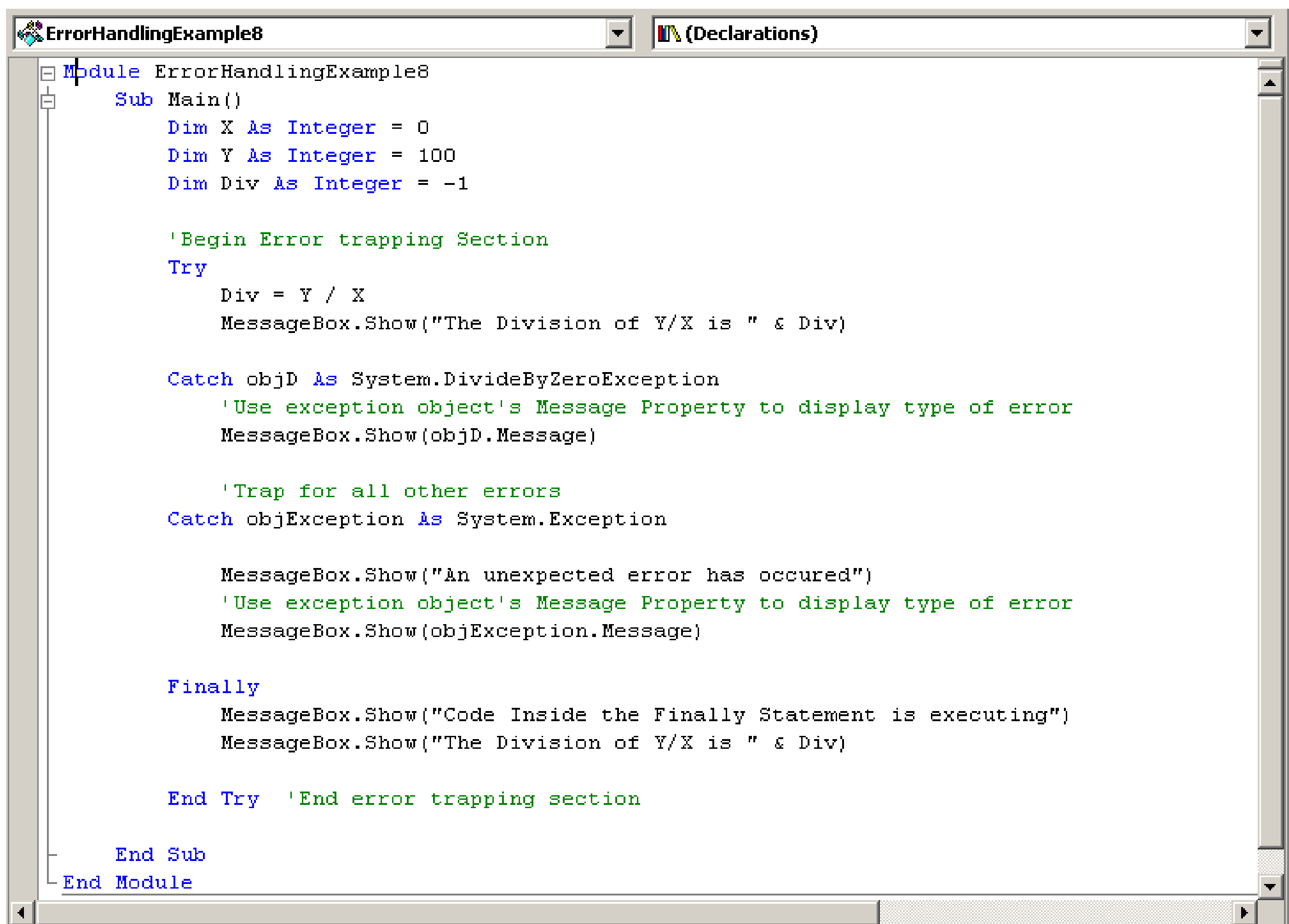
7.2.4 Trapping for Many Error using Multiple Catch Blocks

- In example 7.7 we saw how trapping for a specific error runs the risk of not trapping for another.
- To resolve this problem we can create multiple Catch Blocks, one for each of the specific error we wish to trap and one to trap all errors.
- Lets look at an example

Example 7-8: Trapping for Multiple Errors.

- An In this example we use the wrong exception class but we will add an additional Catch block to trap any other error. This is good programming practice to trap for a particular error but then protect the code by trapping for any other error in case.
- In this example we will divide two Decimal variables and place the result in another Decimal variable, but we will attempt to divide by 0.
- We will use the **`DivideByZeroException`** Class to trap for this specific error and we will use the only main **`Exception`** Class to trap all other errors that may occur.

- The code is as follows:



```
ErrorHandlingExample8                    (Declarations)

Module ErrorHandlingExample8
    Sub Main()
        Dim X As Integer = 0
        Dim Y As Integer = 100
        Dim Div As Integer = -1

        'Begin Error trapping Section
        Try
            Div = Y / X
            MessageBox.Show("The Division of Y/X is " & Div)

        Catch objD As System.DivideByZeroException
            'Use exception object's Message Property to display type of error
            MessageBox.Show(objD.Message)

            'Trap for all other errors
        Catch objException As System.Exception

            MessageBox.Show("An unexpected error has occured")
            'Use exception object's Message Property to display type of error
            MessageBox.Show(objException.Message)

        Finally
            MessageBox.Show("Code Inside the Finally Statement is executing")
            MessageBox.Show("The Division of Y/X is " & Div)

        End Try  'End error trapping section

    End Sub
End Module
```

Discussion of Example:

- This example results in the following:

  - In this example, we create three integer variables and initialize them accordingly:

```
Dim X As Integer = 0
Dim Y As Integer = 100
Dim Div As Integer = -1
```

- We begin the error trapping section by adding the code which we suspect that will generate error inside the **Try block**:

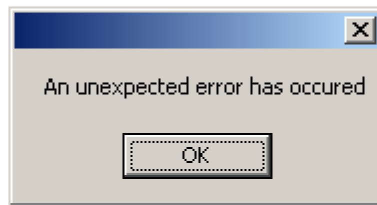
```
Try
Div = Y / X
MessageBox.Show("The Division of Y/X is " & Div)
```

- We then create an object of the **`System.DivideByZeroException` Class** in the **Catch block** to trap for this specific type of error, we also use the *Message Property* of the *DivideByZeroException* Object to display the information about the exception:

```
Catch objException As System.DivideByZeroException

'Use exception object's Message Property to display type of error
MessageBox.Show(objException.Message)
```

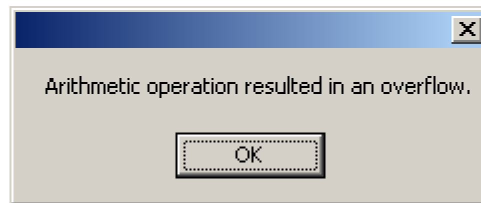
- We then create another Catch Block using an object of the **`System.Exception` Class** in the **Catch block** to trap for all other errors, and we display a message box indicating of the unexpected error:

```
Catch objException As System.Exception

MessageBox.Show("An unexpected error has occured")
'Use exception object's Message Property to display type of error
```



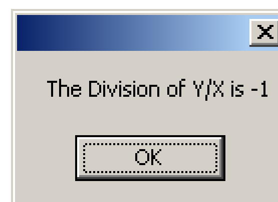
- We also use the *Message Property* of the *Exception* Object to display the information about the exception:

```
'Use exception object's Message Property to display type of error
MessageBox.Show(objException.Message)
```



- Once this code is executed, control is passed to the Finally Block for any cleanup code to be executed.

```
MessageBox.Show("Code Inside the Finally Statement is executing")
MessageBox.Show("The Division of Y/X is " & Div)
```



❖ The important thing to notice here is that by using multiple blocks we were able protect the code from other unexpected errors.

7.2.5 The Throw Statement

Error Handling Summarized

- So far the objectives has been for us to trap errors or exceptions caused by our code using the **Try..Catch..Finally** statement.
- A point to focus on is that the Error or Exception that our code causes is raised by the VB.NET engine. So here is the algorithm of what happens when our code generates an error and what we do about it with the **Try** Statement.

1. The code does something wrong or causes the error
2. VB.NET detects the error and **raises or throw** an error and attempts to stop the program
3. We have in our code a **Try..Catch..Finally** Statement to trap or catch the error.
4. In the Catch block, we handle the error appropriately.

   - In summary, the program does not allow VB.NET to stop the program, but simply let the **Try** Statement handle it.
   - The point that VB.NET has the responsibility of raising the error, and we have the responsibility to trap it using the **Try** Statement

Raising our own errors

- It turns out that we can also programmatically raise or throw an error ourselves.
- This means that we, NOT VB.NET are raising an error.
- But this in turn means that if we raise an error then we are also responsible to trap the error using the Try statement to handle it.
- So now we have the responsibility of raising and catching this error.
- We do this using the **Throw** Statement.
- The **Throw** Statement syntax is as follows:

```
'General Throw Statement Syntax:
Throw New System.ExceptionClass(argument)

        'The Exception Class can be any of the Exception Classes provided by VB.NET
        'Which class you use depends on which type of error you want to throw.
        'You can use the regular Exception class or DivideByZeroException etc.
        'The argument is a text message we would like to display when the error is trapped.
        Note that the System keyword is optional, the following statement would be the same:

Throw New ExceptionClass(argument)
```

- You may be asking yourself, why would we want to throw an error?
- It turns out that there are many situations where this is necessary. As we progress with the course, we will see these situations.
- Also, we can throw an error to indicate something that was supposed to happen did not, therefore we can throw an error and then catch it later. When we catch it we can warn the user that things did not happen as expected.

   - Remember that if we throw an error than it is our responsibility to catch it as well.

Using the Throw Statement

- There are two ways to use the Throw statement:

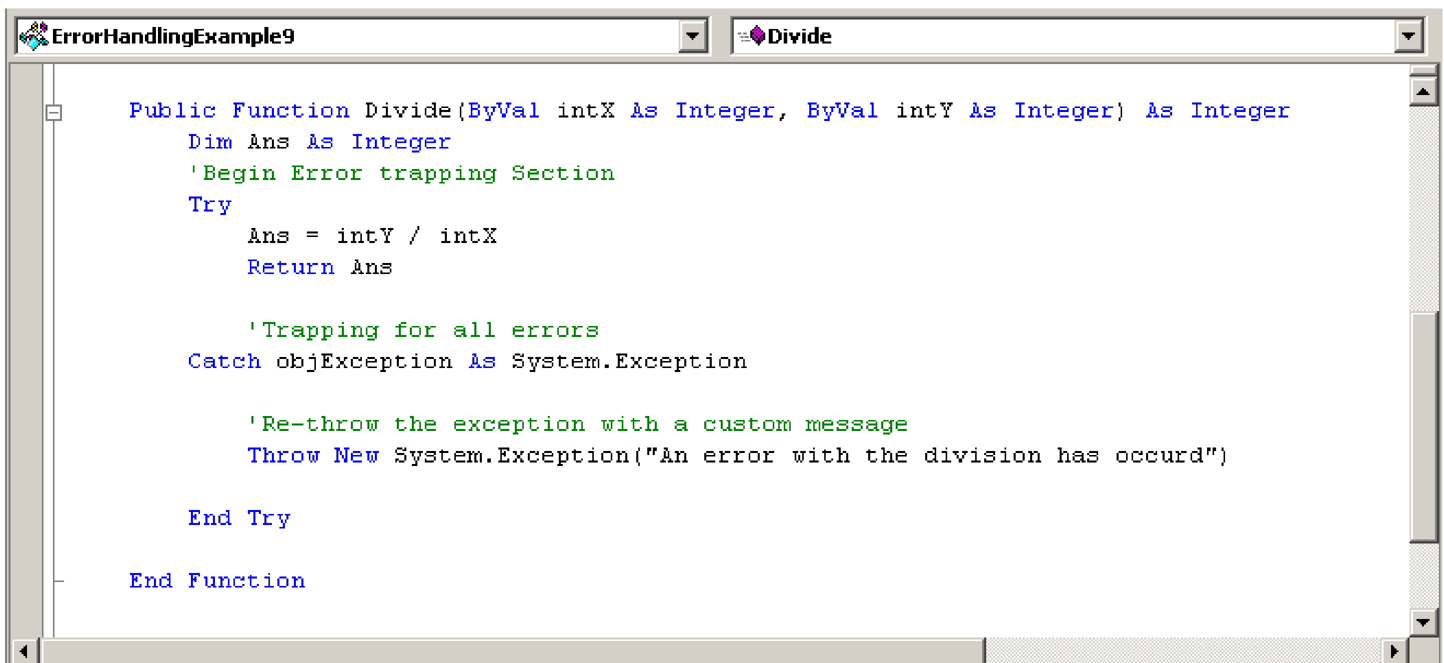
  - To re-throw an exception – Used inside an **Try-Catch-Finally** statement to re-throw the exception that you caught
  - Any other location (out-side an Try-Catch-Finally)

Re-Throwing an Exception

- We can use a **Throw** statement inside a **Try-Catch-Finally** Statement to re-throw the exception caught by the Try Statement.
- This means we can Catch an error, perform what ever necessary processing and then re-throw it so that it is trapped elsewhere or the further on in the program.
- Of course, we need to provide another **Try-Catch-Finally** Statement to catch the exception we just threw.
- The following example demonstrates this concept:

Example 7-9: Re-Throwing

- In this example we extend the divide-by-zero example by creating a function named *Divide(X,Y)* which divides the values passed as arguments.
- Lets first take a look at this function code:



```
ErrorHandlingExample9                    Divide

Public Function Divide(ByVal intX As Integer, ByVal intY As Integer) As Integer
    Dim Ans As Integer
    'Begin Error trapping Section
    Try
        Ans = intY / intX
        Return Ans

        'Trapping for all errors
    Catch objException As System.Exception

        'Re-throw the exception with a custom message
        Throw New System.Exception("An error with the division has occurd")

    End Try

End Function
```

Discussion of Function:

- This example results in the following:

  - The important processing of this function is that it takes the arguments and divides them: **intY/intX** and return the results:

```
Ans = intY / intX
Return Ans
```

  - We begin the error trapping section by adding the code which we suspect that will generate error inside the **Try block**, which is the code the performs the arithmetic:

```
Try
    Ans = intY / intX
    Return Ans
```

- In the **Catch** block, we create an object of the **`System.Exception`** **Class** to trap for all errors. We then call the Throw statement to re-throw the exception again:

```
Catch objException As System.Exception

    'Re-throw the exception with a custom message
    Throw New System.Exception("An error with the division has occurd")
```

- The key point to this example is that in the Catch block, we **Re-Throw** the exception programmatically. We include a custom message in the argument.

- It is now the responsibility of the calling program to trap this error using a **Try-Catch-Finally** Statement; otherwise execution of the program will stop, since the built-in VB.NET mechanism will take over.

  - Note that no **Finally** statement was used. This was simply my choice for this example.

- Lets see how the calling program, in this case Main() handles the exception we just threw:

```
ErrorHandlingExample9                    Main
Module ErrorHandlingExample9
    Sub Main()
        Dim X As Integer = 0
        Dim Y As Integer = 100
        Dim Div As Integer = -1

        Try
            Div = Divide(X, Y)
            MessageBox.Show("The results of division is " & Div)

        Catch objException As System.Exception

            'Using the exception object's Message Property to display type of error
            MessageBox.Show(objException.Message)
        Finally

            MessageBox.Show("The Division of Y/X is " & Div)
        End Try  'End error trapping section

    End Sub
```

Discussion of Main:

- This example results in the following:

  - In this example, we create three integer variables and initialize them accordingly:

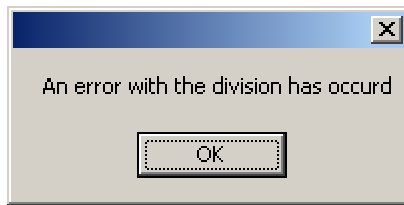
```
Dim X As Integer = 0
Dim Y As Integer = 100
Dim Div As Integer = -1
```

- We begin the error trapping section by adding the code which we suspect that will generate error inside the **Try block**, which is the call to the function Divide(X,Y):

```
Try
    Div = Divide(X, Y)
    MessageBox.Show("The results of division is " & Div)
```

- We then create an object of the **`System.Exception`** **Class** in the **Catch block** to trap for any error. We also use the *Message Property* of the *Exception* Object to display the information about the exception:

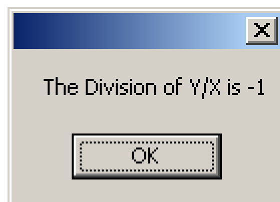
```
Catch objException As System.Exception

 'Using the exception object's Message Property to display type of error
 MessageBox.Show(objException.Message)
```



❖ The important thing to notice here is that the error we are trapping is the error we generated via the **Throw** Statement.

- Once this code is executed, control is passed to the Finally Block for any cleanup code to be executed.

```
 MessageBox.Show("The Division of Y/X is " & Div)
```



- In the previous example we trapped for all errors using the general Exception Object, but we could have also used one of the custom built-in classes to trap for a specific error.
- In the previous example, we could trap specifically for the overflow by using the class System.OverFlowException. This way we trap for an overflow only.
- But remember that if we trap for a specific exception we should also trap for any other exceptions that may happen as a precaution using multiple Catch statements.
- The following example demonstrates this concept:

Example 7-10: Re-Throwing Example Trapping for specific exception

- We will extend the previous example to trap the overflow exception and also trap for all other exception just in case.
- Lets first take a look at this function code:

```
ErrorHandlingExample10                                  Divide

Public Function Divide(ByVal intX As Integer, ByVal intY As Integer) As Integer
    Dim Ans As Integer
    'Begin Error trapping Section
    Try
        Ans = intY / intX

        Return Ans

        'Trapping for Overflow Error Only
    Catch objOvF As System.OverflowException
        'Re-throw the exception with a custom message
    Throw New System.OverflowException("ATTENTION! The Division operation has generated and Overflow error")

        'Trap for all other errors
    Catch objException As System.Exception

        'In case any other error occurs, this message will display
        MessageBox.Show("WARNING! An unexpected error has occurred")
    End Try

End Function
```

Discussion of Function:

- This example results in the following:

  - Again we begin the error trapping section by adding the code which we suspect that will generate error inside the **Try block**, which is the code the performs the arithmetic:

```
Try
    Ans = intY / intX
    Return Ans
```

  - We trap for the overflow exception in the first **Catch** block using an object of the **`System.OverflowException`** **Class** to trap only overflow errors:

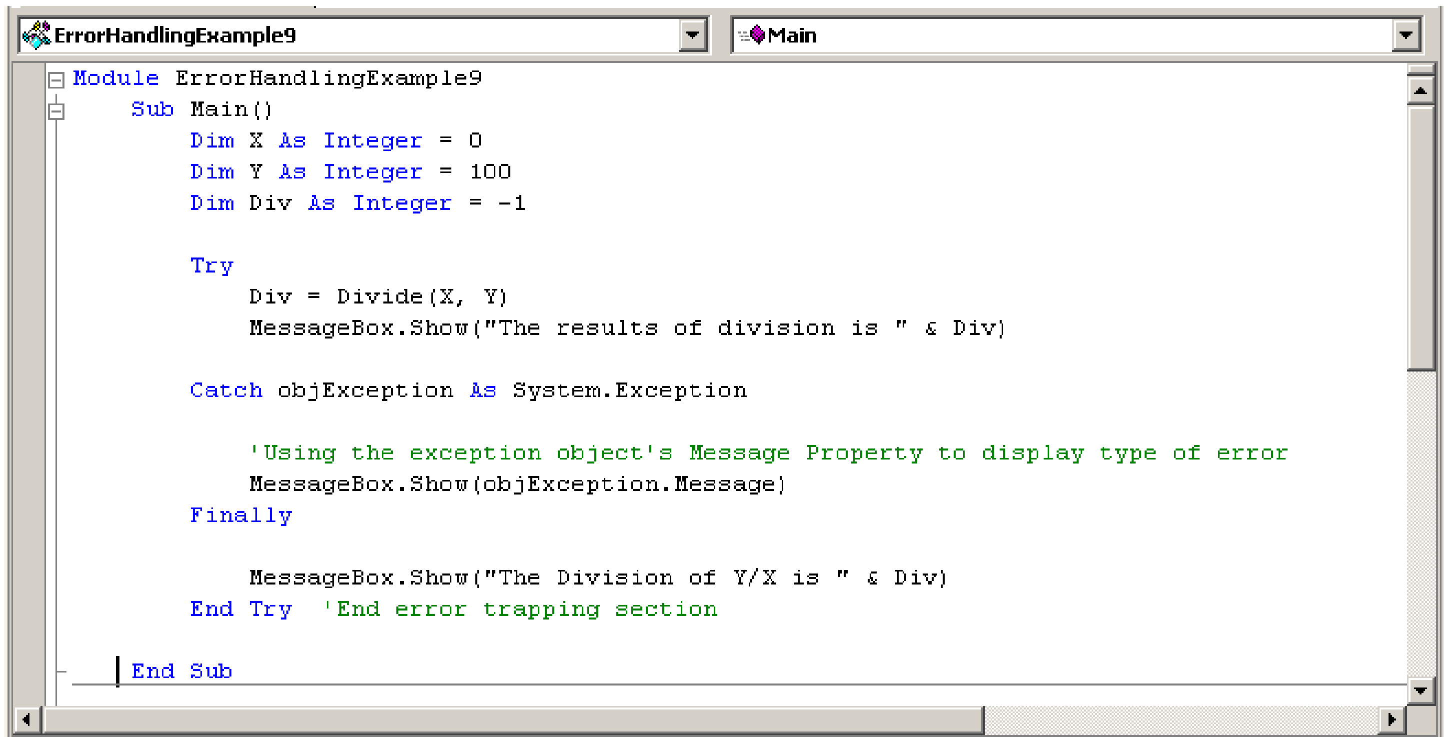
```
Catch objOvF As System.OverflowException
```

  - As the previous example, we will re-throw the exception so that the error is propagated to the calling program:

```
'Re-throw the exception with a custom message
    Throw New System.OverflowException("ATTENTION! The Division operation has
                                        generated and Overflow error")
```

  - It is now the responsibility of the <u>calling program</u> to trap this error using a **Try-Catch-Finally** Statement; otherwise execution of the program will stop, since the built-in VB.NET mechanism will take over.

- As the previous example, the calling program is simply trapping for all errors. Or we could have trapped again only for overflow error. We could do what ever we want:



```
ErrorHandlingExample9                    Main

Module ErrorHandlingExample9
    Sub Main()
        Dim X As Integer = 0
        Dim Y As Integer = 100
        Dim Div As Integer = -1

        Try
            Div = Divide(X, Y)
            MessageBox.Show("The results of division is " & Div)

        Catch objException As System.Exception

            'Using the exception object's Message Property to display type of error
            MessageBox.Show(objException.Message)
        Finally

            MessageBox.Show("The Division of Y/X is " & Div)
        End Try  'End error trapping section

    End Sub
```

Discussion of Main:

- Same as previous example.

Throwing an Exception in other location (NOT inside Try-Catch-Finally)

- We can use a **Throw** statement in other location of a program, not just inside a **Try-Catch-Finally** Statement.
- But remember, we need to have a **Try-Catch-Finally** in the section that follows the code in order to trap it, otherwise the program will stop.
- We will cover this type of approach to using the Throw statement in future lectures.

7.3 Shared Methods

7.3.1 Overview & Discussion

- In the beginning of this course you learned the 3 BASIC STEPS TO AN OBJECT-ORIENTED PROGRAM:

> I. **Create the class specification or Class Module**
>   - Private Data, Properties & Methods
>
> II. **Create Object of the Class**
>
> III. **Use the Object of the Class**
>   - Write the program to manipulate, access or modify the objects data & Call the Methods & and Trigger Events

- We also emphasized on the difference between a Class and the Object.
- A Class is a template which defines what the object will look like and the object is an instance or manifestation of a Class.
- We also pressed the issue that we CANNOT use the class methods and members prior to creating an object. For example supposed we have the following class and object declarations:

Example 1:

```
'Class Declaration:
Public Class clsEmployee

      Public Function Authenticate (sUser As string, sPass As String) As Boolean
            'function code
      End Function

End Class
```

```
'SYNTAX ERROR !!!Using class name to access  members results in Syntax Error:
bolResults = clsEmployee.Authenticate("joe", "111")    'Result in Syntax Error
```

- In the example we called the *Class.Method()* syntax to use the class and we know this is a violation. You don't use the Class but the object of a class. The class is only a template and it's members are only available after an object is created. Let's look a the correct syntax:

Example 2:

```
'Class Declaration:
Public Class clsEmployee

      Public Function Authenticate (sUser As string, sPass As String) As Boolean
              'function code
      End Function

End Class

'Object Declaration:
Public objEmployee1 As New clsEmployee


'Correct Declaration using object:
bolResults = objEmployee1.Authenticate("joe", "111")
```

7.3.2 Shared Methods

- VB.NET actually provides a mechanism called ***Shared Methods*** , which allows us to use member methods (procedures/functions) without the need to create an object.
- I understand that from the start you were taught that this is not allowed and is a violation. Now we take it back. There are circumstances were it may be best to actually be able to call a class method WITHOUT having to instantiate or create an object.

- You have actually used this already without realizing it. For example when using *Console Applications*, to display information to the screen we used the following syntax:

```
Console.WriteLine("Access Granted")
```

- Did you realized that Console is the Class name? and that there is no object declared when we called the *WriteLine()* method? In the Namespace Console is the class that handle console applications and *WriteLine()* is a method of this class. So, you were using the Class.Method() syntax and did not realize it.
- It turns out that *WriteLine()* is a Shared method this is why we were able to do this.

- The Syntax to implement a Share method is as follows:

```
'Syntax for Shared Sub Procedure:
Public Shared Sub ProcedureName ( [argument list] )
        'Body Code!
End Sub

'Syntax for Shared Function Procedure:
Public Shared Function ProcedureName ( [argument list] ) As ReturnType
        'Body Code!
        Return = ReturnValue
End Function
```

- Now lets' look at our previous CLASS DECLARATION example, but this time we make the METHOD A SHARED METHOD:

Example 3:

```
'Class Declaration:
Public Class clsEmployee

Public Shared Function Authenticate(ByVal sUser As String, ByVal sPass As String) As
Boolean
        'function code

End Function

End Class

'GOOD CODE! EXECUTED METHOD WITHOUT OBJECT!
bolResults = clsEmployee.Authenticate("joe", "111")    'RETURNS
```

7.4 Shared Variables

7.4.1 Overview

- In our review of VB.NET we discussed the various types of errors that can occur in a program.
- INCOMPLETE SECTION!