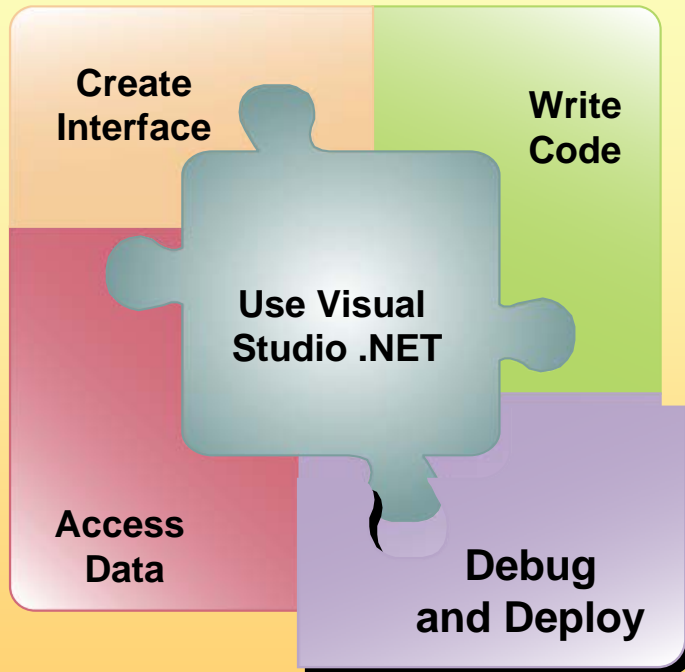


**Module 8:  
Handling Errors and  
Exceptions**

# Overview



- Types of Errors
- Using the Debugger
- Handling Exceptions

# Lesson: Types of Errors

- What Are Syntax Errors?
- What Are Run-time Errors?
- What Are Logic Errors?

# What Are Syntax Errors?

```
Public Sub PerformAction()  
    Dim newValue As xyz  
    MessageBox.Show("Test")  
End Sub  
End Class
```

**Syntax Error** (points to `xyz`)

**Syntax Error** (points to `MessageBox`)

Name 'MessageBox' is not declared.

View errors in the Task List:

| Task List - 2 Build Error tasks shown (filtered) |                                     |                                    |                           |      |
|--|-------------------------------------|------------------------------------|---------------------------|------|
| !  | <input checked="" type="checkbox"/> | Description                        | File                      | Line |
| Click here to add a new task                     |                                     |                                    |                           |      |
| !  |                                     | Type 'xyz' is not defined.         | C:\Documents ...\Form1.vb | 46   |
| !  |                                     | Name 'MessageBox' is not declared. | C:\Documents ...\Form1.vb | 47   |

# What Are Run-time Errors?

**Speed = Miles/Hours**

' If *Hours* = 0, the statement is syntactically correct, but the division is an invalid operation

## Microsoft Development Environment



An unhandled exception of type 'System.OverflowException' occurred in RuntimeError.exe

Additional information: Arithmetic operation resulted in an overflow.

Break

Continue

Ignore

Help

# What Are Logic Errors?

- Definition: An error that causes an application to produce incorrect results
  - Might not generate an error message
  - Found by testing the application and analyzing the results

```
Dim x As Integer = 2  
Do While x < 10  
  ' Code statements  
  x -= 1  
Loop
```

# Demonstration: Types of Errors

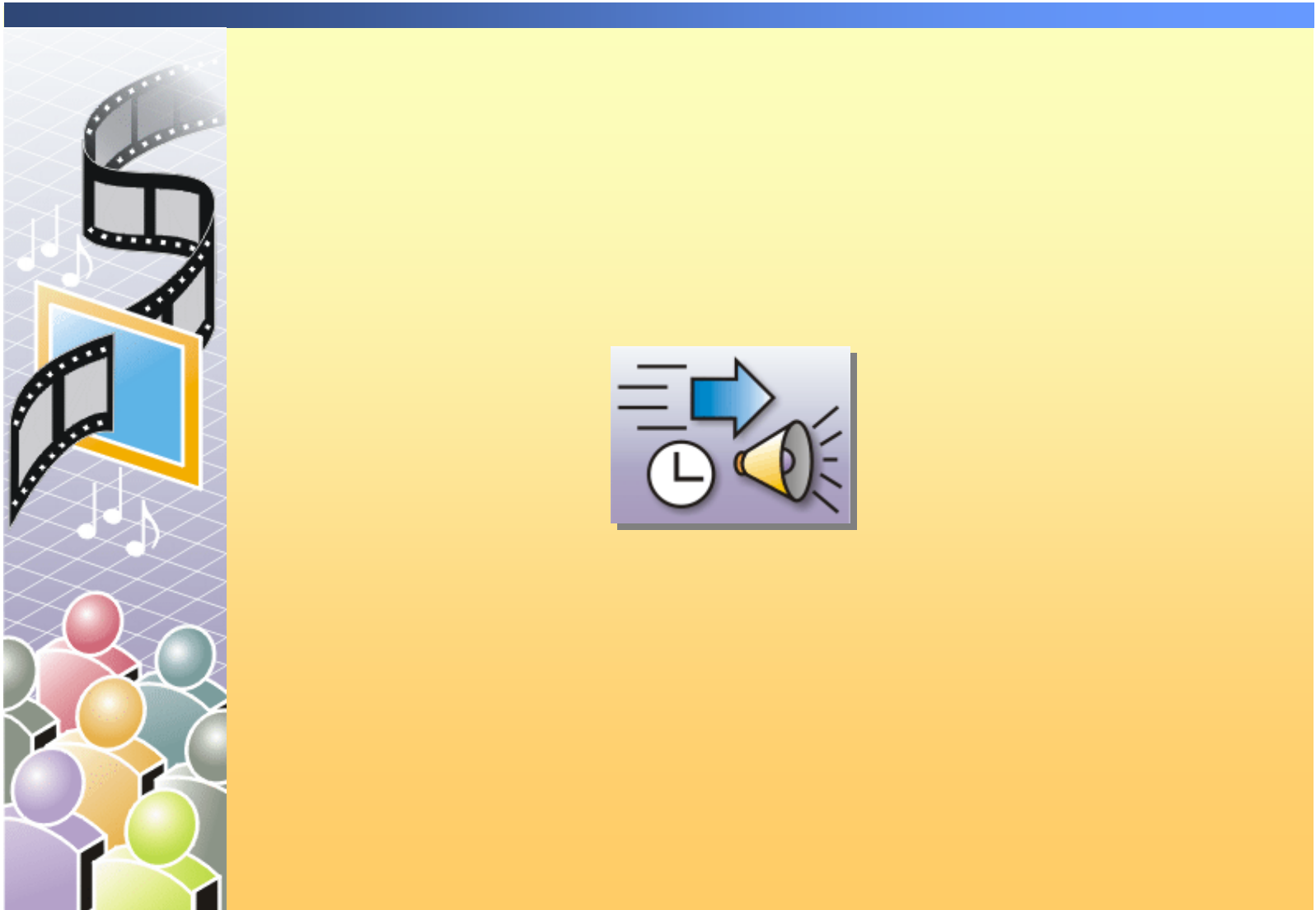


- In this demonstration, you will learn about the following types of errors, and you will see an example of each error in the Visual Studio .NET environment:
  - Syntax error
  - Run-time error
  - Logic error

# Lesson: Using the Debugger

- What Is Break Mode?
- How to Use Breakpoints
- How to Modify Breakpoints
- The Debug Toolbar
- How to Step Through Code
- How to Use Debugging Windows
- How to Use the Command Window

# Multimedia: How to Debug an Application



# What Is Break Mode?

- Pauses the operation of an application
- In break mode, you can:
  - Step through your code line by line
  - Determine the active procedures that have been called
  - Watch the values of variables, properties, and expressions
  - Use debugging windows to change the values of variables and properties
  - Change program flow
  - Run code statements

# How to Use Breakpoints

- A breakpoint is a marker in your code that causes Visual Basic to pause code execution at a specific line
- You cannot place a breakpoint on non-executable code

## Breakpoints

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
    Dim maxLen As Integer = 0  
    Dim longestName As String = ""  
    Dim index As Integer  
  
    For index = 0 To names.GetLength(0)  
        If names(index).Length > maxLen Then  
            longestName = names(index)  
        End If  
    Next  
  
    MsgBox("Longest name is " & longestName)  
End Sub  
End Class
```

# How to Modify Breakpoints

The image shows a 'New Breakpoint' dialog box with the following fields and options:

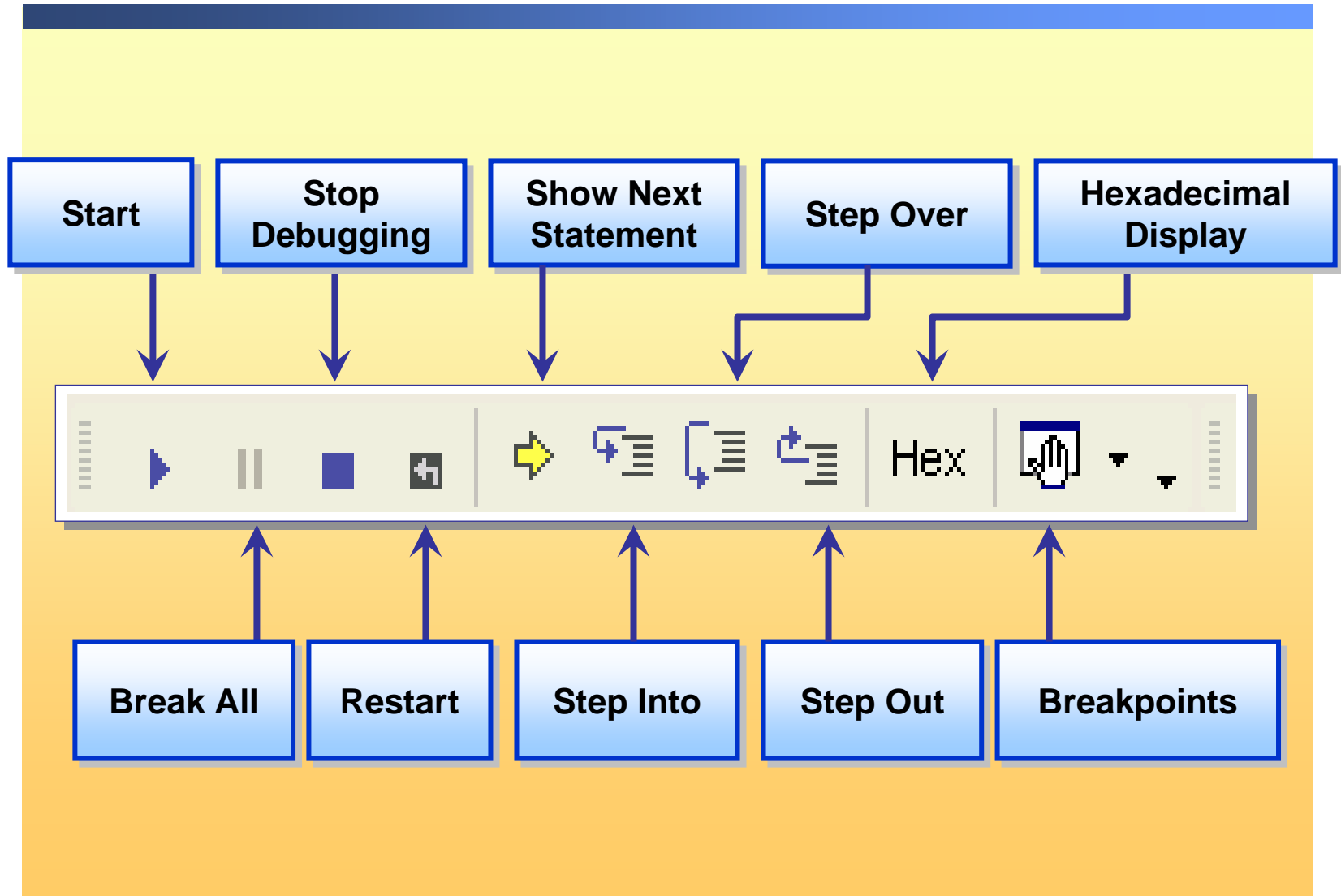
- Function: FilterIndex
- Line: 1
- Language: Basic
- Character: 1
- Condition...: when 'accountBalance < 0' is true
- Hit Count...: when hit count is equal to 3

Annotations:

- Condition Property**: A pink box with an arrow pointing to the 'Condition...' button.
- Hit Count Property**: A pink box with an arrow pointing to the 'Hit Count...' button.

Buttons at the bottom: OK, Cancel, Help.

# The Debug Toolbar



# How to Step Through Code

- **Step Into or Step Over:** executes the next line of code. If the next line contains a procedure call:
  - **Step Into:** executes only the call, and then halts at the first line of code inside the procedure
  - **Step Over:** executes the procedure, and then halts at the first line of code outside the procedure
- **Step Out:** resumes execution until the procedure returns, and then breaks at the return point in the calling procedure
- **Run To Cursor:** the debugger runs your application until it reaches the insertion point that you set

# Demonstration: How to Use Debugging Windows



- In this demonstration, you will learn how to debug an application by using:
  - The Autos, Locals, Watch, and Command windows
  - Breakpoints
  - The **Run To Cursor** command

# How to Use Debugging Windows

| Window            | Use this window to...  |
|-------------------|--|
| <b>Autos</b>      | View variables in the current statement and three statements before and after the current statement  |
| <b>Call Stack</b> | View the history of calls to the line of code being debugged   |
| <b>Locals</b>     | View and modify local variables  |
| <b>Watch</b>      | <ul style="list-style-type: none"><li>▪ Create a custom list of variables and expressions to monitor</li><li>▪ View and manipulate any watch expressions</li></ul> |

# How to Use the Command Window

- Use the Command window to:
  - Issue commands (Command mode)
  - Debug and evaluate expressions (Immediate mode)

| Task   | Solution   | Example     |
|--|--|-------------|
| Evaluate expressions                               | Preface the expression with a question mark (?)                      | ?myVariable |
| Switch to Immediate mode from Command mode         | Type <b>immed</b> into the window, without the greater than sign (>) | immed       |
| Switch back to Command mode from Immediate mode    | Type <b>&gt;cmd</b> into the window                                  | >cmd        |
| Temporarily enter Command mode from Immediate mode | Type the command, prefacing it with a greater than sign (>)          | >alias      |

# Practice: Debugging Code



**1** Examine the code in the Click event handler

**2** Build and run the application

**3** Use debugging tools to locate the logic error

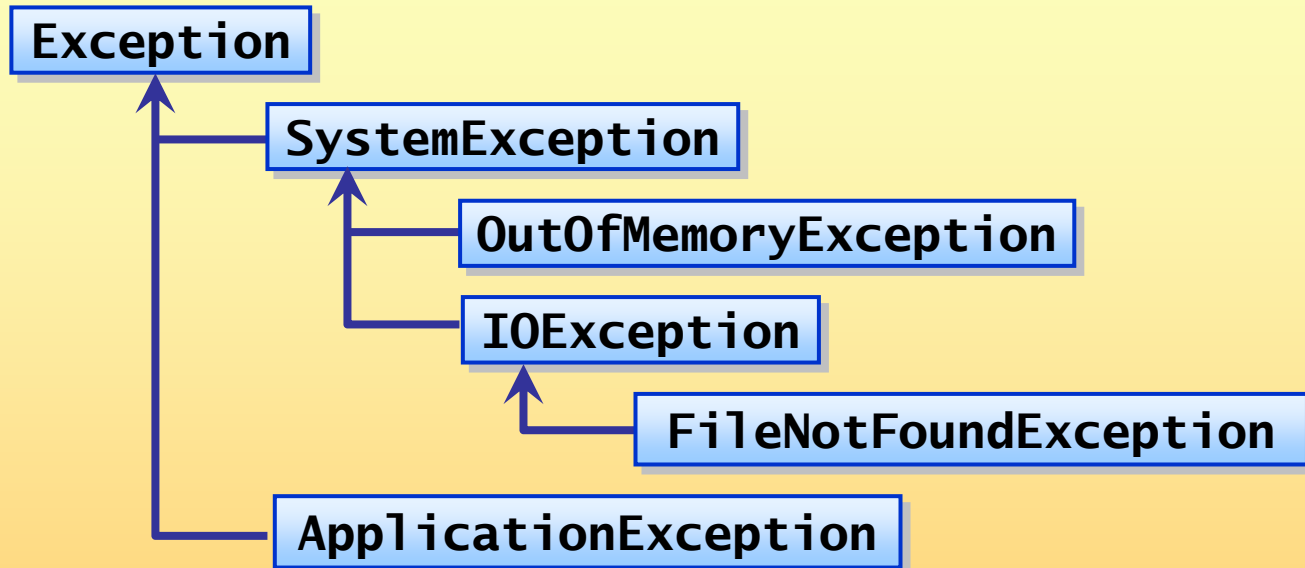
**4** Suggest a way to fix the error

# Lesson: Handling Exceptions

- The Exception Class
- What Is Structured Exception Handling?
- How to Use the Try...Catch Statement
- How to Use the Finally Block
- How to Throw Exceptions
- Guidelines for Using Structured Exception Handling

# The Exception Class

- The .NET Framework provides the following exception object model:



- Exception classes enable you to retrieve information about any exception you encounter
- Properties of the Exception base class enable you to analyze exceptions
  - Key properties: StackTrace, Message, HelpLink, Source

# What Is Structured Exception Handling?

- Detect and respond to errors while an application is running
- Use Try...Catch...Finally to encapsulate and protect blocks of code that have the potential to raise errors
  - Each block has one or more associated handlers
  - Each handler specifies some form of filter condition on the type of exception it handles
- Advantages:
  - Allows separation between logic and error handling code
  - Makes code easier to read, debug, and maintain

# How to Use the Try...Catch Statement

- Put code that might throw exceptions in a Try block
- Handle the exceptions in a separate Catch block

**Try**

```
fs = New FileStream("data.txt", _  
    FileMode.Open)
```

Program Logic

**Catch** ex As FileNotFoundException

```
    MessageBox.Show("File not found")
```

Exception Handling

**Catch** ex As Exception

```
    MessageBox.Show(ex.Message)
```

**End Try**

# How to Use the Finally Block

- Optional section; if included, it always executes
- Place cleanup code, such as that for closing files, in the Finally block

## Try

```
fs = New FileStream("data.txt", FileMode.Open)
```

**Catch** ex As FileNotFoundException

```
MessageBox.Show("Data File Missing")
```

**Catch** ex As Exception

```
MessageBox.Show(ex.Message)
```

## Finally

```
If Not (fs Is Nothing) Then fs.Close( )
```

## End Try

# How to Throw Exceptions

- Use the Throw statement to create an exception that you can handle with structured exception handling code

```
If (day < 1) Or (day > 365) Then
    Throw New ArgumentOutOfRangeException( )
Else
    ...
End If
```

# Guidelines for Using Structured Exception Handling

Do not use structured exception handling for errors that are likely to occur routinely. Use other code blocks to address these errors.

- **If...End If**, and so on.

Return a value for common error cases.

- Example: File I/O read methods do not throw end-of-file exception.

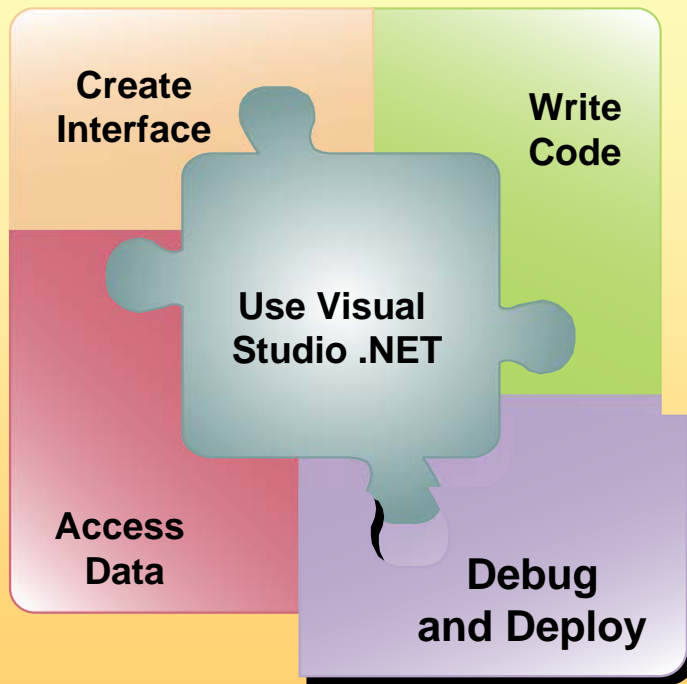
Arrange **Catch** blocks from specific to general.

# Demonstration: Using Structured Exception Handling



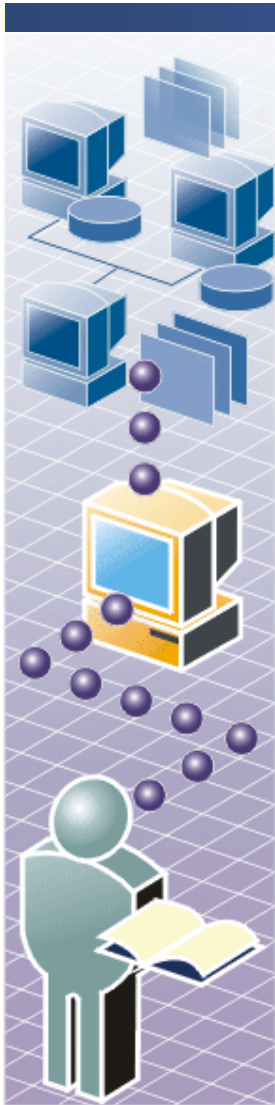
- In this demonstration, you will learn how to implement structured exception handling in your code

# Review



- Types of Errors
- Using the Debugger
- Handling Exceptions

# Lab 8.1: Implementing Structured Exception Handling



- Exercise 1: Using Try...Catch Blocks
- Exercise 2: Using Try...Catch...Finally Blocks