msdn® training

Module 3: Language and Syntax Enhancements

Contents

Overview	1
Data Types	2
Using Variables	9
Demonstration: Using Variables and Data Structures	20
Functions, Subroutines, and Properties	21
Lab 3.1: Working with Variables and Procedures	29
Exception Handling	36
Demonstration: Structured Exception Handling	48
Lab 3.2: Implementing Structured Exception Handling	49
Review	52

This course is based on the prerelease version (Beta 2) of Microsoft® Visual Studio® .NET Enterprise Edition. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 2 version of Visual Studio .NET Enterprise Edition.

Microsoft®

Instructor Notes

Presentation:
90 Minutes

Labs:
75 Minutes

This module provides students with the knowledge needed to use many of the new language and syntax enhancements to the Microsoft® Visual Basic® language, including variable declaration and initialization, procedure syntax, structured exception handling, and assignment operator changes.

In the labs, students will create a simple application that uses variables and procedures to add customers to a customer array. The students will then add structured exception handling to the application.

After completing this module, students will be able to:

- Describe the changes to data types in Visual Basic .NET.
- Declare and initialize variables and arrays.
- Use shorthand syntax to assign values to variables.
- Implement functions and subroutines.
- Call the default properties of an object.
- Use the new **Try…Catch…Finally** statement to implement structured exception handling.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the following materials:

- Microsoft PowerPoint® file 2373A_03.ppt
- Module 3, "Language and Syntax Enhancements"
- Lab 3.1, Working with Variables and Procedures
- Lab 3.2, Implementing Structured Exception Handling

Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Read the instructor notes and the margin notes for the module.
- Practice the demonstrations.
- Complete the labs.

Demonstrations

This section provides demonstration procedures that will not fit in the margin notes or are not appropriate for the student notes.

Using Variables and Data Structures

↙ To test the application

1. Open the DataTypes.sln file in the *install folder*\DemoCode\Mod03\DataTypes folder.
2. In the code for frmData.vb, set breakpoints on the first lines of the **TestVariables** and **TestStructures** subroutines.
3. Run the project.

↙ To debug the Variables button

1. Click the **Variables** button on the form. The Visual Basic process will halt at the first breakpoint.
2. Explain the code, pointing out that both *intA* and *intB* are created as **Integer** data types.
3. Step through the code by using F11, and explain the new assignment operators and the **CType** function.
4. Allow the program to continue by pressing F5.
5. Stop the project and uncomment the **Option Strict** code at the top of the form code. Attempt to run the project. A compilation error will occur. Explain why this occurs. Re-comment the **Option Strict** code, and run the project to test the **Structures** code.

↙ To debug the Structures button

1. Click the **Structures** button on the form, and step through the **TestStructures** subroutine. Point out the **Employee** structure definition at the top of the form code before continuing to debug. Note that the array is initialized to a size of two employees. Continue debugging the code by using F11. Explain each line when required.
2. Point out the block-level variable *iCounter*. Allow the remaining code to execute before closing the form and stopping the debugger.

↙ To create an out-of-scope variable exception

1. Return to the form code and uncomment the final line in the **TestStructures** routine **MsgBox(***iCounter***)**. Attempt to run the project again, and observe the error message that is generated because of the attempt to access the block-level variable outside of its scope.
2. Close the Microsoft Visual Studio® .NET integrated development environment (IDE).

Structured Exception Handling

↙ To test the application

1. Open the Exceptions.sln solution in the *install folder*\DemoCode\Mod03\Exceptions folder.
2. Open the code window for the Errors.vb form and set breakpoints on the **Try** statement, each **Catch** statement, and the **Finally** statement in the **RunExceptions** routine.
3. Run the Exceptions project.
4. Click the **Overflow** button on the form. The Visual Basic process will halt at the first breakpoint.
5. Use F11 to step through the code in the **Try** block, and explain the overflow details. Explain that the **OverflowException** class is filtering the exception. Step through the remaining code, and allow execution to continue.
6. Click the remaining buttons on the test form (**Divide**, **Err.Raise**, and **Throw**) to demonstrate how the exceptions are handled in each case.
7. End the debugging session by closing the form.
8. Close the Visual Studio .NET IDE.

Module Strategy

Use the following strategy to present this module:

- Data Types

  This lesson shows some of the new data types and the changes to existing data types. This lesson also introduces the common type system, the differences between value and reference types, and the **CType** keyword that is essential for converting variables of one data type to another. Creating objects is not covered in this module, but it may be necessary to point out some simple syntax in the examples in which the **New** keyword is used. This topic will be covered in Module 4, "Object-Oriented Design for Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

  It may be worth pointing out that the new Microsoft .NET Framework classes provide alternative ways to perform many similar tasks that can be performed in the Visual Basic language. The **System.Collections** namespace contains several good examples of these classes.

- Using Variables

  This lesson shows how to declare and initialize variables in Visual Basic .NET and introduces changes to variable scope. The topic of data structures is introduced. This topic will be covered further in Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*. Point out that **Option Explicit** is the default compiler option in Visual Basic .NET. Finally, introduce students to the new shorthand syntax for assignment operators.

- Functions, Subroutines, and Properties

  This lesson shows the changes to functions and subroutines, in addition to the changes to the calling syntax for object properties. Remind students that object property creation will be covered in Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

  In the default property examples that use ADO Recordset objects, focus the discussion on the errors generated by incorrect use of default properties. Other errors may also be generated by assigning incorrect data types to other variables, but these errors should be ignored.

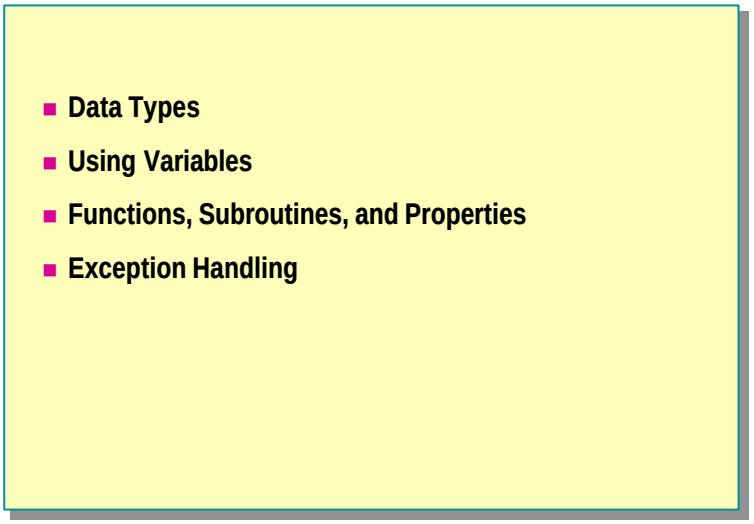
- Exception Handling

  This lesson shows the new structured exception handling syntax and explains why it is preferable to unstructured exception handling. To create exceptions by using the **Throw** statement, as used in the notes and labs, requires a basic understanding of object constructors that is not covered until Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*. Point this out to students and provide a short explanation of this statement.

Overview

Topic Objective
To provide an overview of the module topics and objectives.

Lead-in
In this module, you will learn about the enhancements to the Visual Basic language and syntax in Visual Basic .NET.



Microsoft® Visual Basic® .NET version 7.0 introduces many language and syntax enhancements that help make it an excellent development tool for the Microsoft .NET platform. Some of these enhancements include:

- Incorporation of the .NET Framework type system, making Visual Basic .NET compatible with other languages in the .NET Framework.
- Enhancements to syntax for working with variables, thereby increasing the clarity and performance of code.
- Changes to functions, subroutines, and properties, making code easier to read and maintain.
- Structured exception handling, making Visual Basic .NET a more robust development language.

After completing this module, you will be able to:

- Describe the changes to data types in Visual Basic .NET.
- Declare and initialize variables and arrays.
- Use shorthand syntax to assign values to variables.
- Implement functions and subroutines.
- Call the default properties of an object.
- Use the new **Try...Catch...Finally** statement to implement structured exception handling.

Data Types

Topic Objective
To explain the changes to data types in Visual Basic .NET.

Lead-in
This lesson discusses changes to data types and how they are used in Visual Basic .NET code.

- Common Type System
- Comparing Value-Type and Reference-Type Variables
- New Data Types
- Changes to Existing Data Types
- Using CType to Convert Data Types

In this lesson, you will learn about the data types available in Visual Basic .NET. After you complete this lesson, you will be able to:

- Explain the .NET Framework common type system and how it affects Visual Basic .NET development.
- Explain the difference between value-type variables to reference-type variables.
- Describe and use the data types available in Visual Basic .NET.
- Use the **CType** function to convert values from one data type to another.

Common Type System

Topic Objective
To introduce the common type system.

Lead-in
The common type system provides predefined data types for the .NET Framework.

- **Integrated in the Common Language Runtime**
- **Shared by the Runtime, Compilers, and Tools**
- **Controls How the Runtime Declares, Uses, and Manages Types**
- **Includes a Set of Predefined Data Types**
- **Common Type System Objects Are Based on the System.Object Class**

The .NET Framework is based on a new common language runtime. The runtime provides a common set of services for projects built in Visual Studio® .NET, regardless of the language. The common type system is an integral part of the runtime. The compilers, tools, and the runtime itself share the common type system. It is the model that defines the rules that the runtime follows when declaring, using, and managing types. The common type system establishes a framework that enables cross-language integration, type safety, and high-performance code execution.

All objects in the common type system are based on the **System.Object** class, and all data types declared in Visual Basic .NET code correspond directly to a common type system data-type. For example, when you declare a variable of type **Integer** in Visual Basic .NET, it is the same as declaring a **System.Int32** common type system data type. The keyword **Integer** is an alias for the Int32 data type, and it provides familiar syntax to Visual Basic developers.

Comparing Value-Type and Reference-Type Variables

Topic Objective
To describe differences between value-type and reference-type variables.

Lead-in
Value-type and reference-type variables have significant differences that developers need to understand.

- **Value-Type Variables**
  - Directly contain their data
  - Each has its own copy of data
  - Operations on one cannot affect another
  - Assignment creates a copy of the data
- **Reference-Type Variables**
  - Store references to their data (known as objects)
  - Two reference variables can reference the same object
  - Operations on one can affect another

When you define a variable, you need to choose the right data type for your variable. The data type determines the allowable values for that variable, which, in turn, determine the operations that can be performed on that variable. The common type system supports both value-type and reference-type variables.

Value-Type Variables

Value-type variables directly contain their data. Each value-type variable has its own copy of data, so operations on one value-type variable cannot affect another variable.

Examples of value-type variables include integers, doubles, floats and structures.

Reference-Type Variables

Reference-type variables contain references to their data. The data is stored in an instance. Two reference-type variables can reference the same object, so operations on one reference-type variable can affect the object referenced by another reference-type variable.

Examples of reference-type variables include strings, arrays, and classes.

New Data Types

Topic Objective
To describe the new data types introduced in Visual Basic .NET.

Lead-in
Three new data types are available in Visual Basic .NET: **Char**, **Short**, and **Decimal**.

Visual Basic .NET data type	Storage size	Value range
Char	2 bytes	0 to 65535 (unsigned)
Short	2 bytes	-32,768 to 32,767
Decimal	12 bytes	Up to 28 digits on either side of decimal (signed)

Delivery Tip
The changes to **Short** and the other **Integer** data types are covered in more detail on the next slide.

Point out that **Decimal** is a fixed-point type integer as opposed to **Double** and **Single**, which are floating point. Therefore, **Decimal** is more accurate for precise calculations.

There are three new data types available in Visual Basic .NET: **Char**, **Short,** and **Decimal**.

Char

This data type stores a single Unicode character in a two-byte variable.

Short

In Visual Basic 6.0, a 16-bit integer is an **Integer** data type. In Visual Basic .NET, a 16-bit integer is designated as a **Short**.

Decimal

A **Decimal** data type is stored as a 96-bit (12-byte) fixed-point signed integer, scaled by a variable power of 10. The power of 10 specifies the precision of the digits to the right of the decimal point, and ranges from 0 to 28. This data type should be used when calculations are required that cannot tolerate rounding errors; for example, in financial applications.

If no decimal places are required, the **Decimal** data type can store up to positive or negative 79,228,162,514,264,337,593,543,950,335.

Using the full 28 places for precision, the largest value that can be stored is 7.9228162514264337593543950335 and the smallest non-zero value is positive or negative 0.0000000000000000000000000001.

Changes to Existing Data Types

Topic Objective
To describe the changes to existing data types in Visual Basic .NET.

Lead-in
Several data types from Visual Basic 6.0 have changed or are no longer supported.

Visual Basic 6.0	Visual Basic .NET
Integer	**Short**
Long (32 bits, signed)	**Integer**
(none)	**Long** (64 bits, signed)
Variant	Not supported: use **Object**
Currency	Not supported: use **Decimal**
Date	No longer stored as a **Double**
String (fixed length)	Not supported

Several data types from Visual Basic 6.0 have changed or are no longer supported in Visual Basic .NET. These changes make data types in Visual Basic .NET more consistent with data types used by other programming languages in the .NET Framework and in the runtime.

Integer

The **Integer** and **Long** data types in Visual Basic 6.0 have a different meaning in Visual Basic .NET, as described in the following table.

Integer size	Visual Basic 6.0 data type	Visual Basic .NET data type	.NET Framework and runtime type
16 bits, signed	Integer	Short	System.Int16
32 bits, signed	Long	Integer	System.Int32
64 bits, signed	(None)	Long	System.Int64

Variant

Visual Basic .NET updates the universal data type to **Object** for compatibility with the common language runtime.

Visual Basic 6.0

You can assign to the **Variant** data type any primitive type (except fixed-length strings) and **Empty**, **Error**, **Nothing**, and **Null**.

Visual Basic .NET

The **Variant** type is not supported, but the **Object** data type supplies its functionality. **Object** can be assigned to primitive data types, **Nothing**, and as a pointer to an object.

Currency

The **Currency** data type is not supported in Visual Basic .NET. You can use the **Decimal** data type as a replacement. The **Decimal** data type uses 12 bytes of memory, and allows more digits on both sides of the decimal point.

Date

The **Date** data type is available in Visual Basic .NET but is not stored in the same format as it was in Visual Basic 6.0.

Visual Basic 6.0

The **Date** data type is stored in a **Double** format.

Visual Basic .NET

Date variables are stored internally as 64-bit integer. Because of this change, there is no implicit conversion between **Date** and **Double** as there is in previous versions of Visual Basic. Representing dates as integers simplifies and speeds up the manipulation of dates.

String

Fixed-length strings are no longer supported, but you can simulate this behavior by padding a string to the desired length with spaces, as shown in the following example:

```
'Create a string containing spaces
Dim s As String = Space(10)
```


The type name **String** is an alias for the **System.String** class. Therefore, **String** and **System.String** can be used interchangeably. The **String** class represents a string of characters that cannot be modified after the text has been created. Methods that appear to modify a string value actually return a new instance of the string containing the modification.

This can impact performance in applications performing a large number of repeated modifications to a string, so the **System.Text.StringBuilder** object is provided. This object allows you to modify a string without creating a new object, and is therefore a better choice if you are performing a large number of string manipulations. The following example shows how to create a **StringBuilder** variable and how to append values to it:

Delivery Tip
Point out the **StringBuilder** class and the example shown in the notes.

```
Dim s As New system.Text.StringBuilder()
s.Append("This")
s.Append(" is")
s.Append(" my")
s.Append(" text!")
MsgBox(s.ToString)   'generates "This is my text!"
```

Visual Basic 6.0 provides many string manipulation methods that are still available in Visual Basic .NET. The **System.String** class also has many predefined properties and methods that simulate this behavior by using an object-oriented approach. These properties and methods include **Insert**, **Length**, **Copy**, **Concat**, **Replace**, **Trim**, **ToLower**, and **ToUpper**. For more information, search for “string methods” in the Visual Studio .NET documentation.

Using CType to Convert Data Types

Topic Objective:
To explain how to use the new **CType** function.

Lead-in:
In Visual Basic .NET, you can convert any data type to any other data type by using the **CType** function.

- **Use CType to Convert Values from One Data Type to Another Data Type**
- **Similar to CStr and CInt in Visual Basic 6.0**
- **Syntax:**
  - `CType (expression, typename)`

You can use the **CType** function to convert any value from one data type to another data type. If the value is outside the range allowed by the type, an error will occur. The **CType** function is similar to the **CStr** and **CInt** conversion functions in Visual Basic 6.0, but it can be used for composite data type conversion in addition to elementary types.

Syntax

Use the following syntax to convert data types:

```
CType(expression, typename)
```

- *expression*

  The expression argument can be any valid expression, such as a variable, a result of a function, or a constant value.

- *typename*

  The typename argument can be any expression that is valid within an **As** clause in a **Dim** statement, such as the name of any data type, object, structure, class, or interface.

Example

The following examples show how to convert a **String** value to an **Integer**, and how to convert to a data structure type:

```
Dim x As String, y As Integer
x = "34"
y = CType(x, Integer)

Dim custNew as Customer 'Predefined structure type
custNew = CType(data, Customer)
```

◆ Using Variables

Topic Objective
To explain how to declare, initialize, and use variables and arrays in Visual Basic .NET.

Lead-in
This lesson explains the differences between declaring, initializing, and using variables in Visual Basic 6.0 and Visual Basic .NET.

- **Declaring and Initializing Variables and Arrays**
- **Declaring Multiple Variables**
- **Variable Scope**
- **Creating Data Structures**
- **Compiler Options**
- **Assignment Operators**

After you complete this lesson, you will be able to:

- Declare and initialize variables.
- Explain changes to variable scope in Visual Basic .NET.
- Create data structures.
- Use compiler options effectively.
- Use a new shorthand syntax for assignment operators.

Declaring and Initializing Variables and Arrays

Topic Objective
To explain how variables and arrays can be declared and initialized.

Lead-in
How do you declare variables and arrays in Visual Basic .NET?

- **You Can Initialize Variables When You Declare Them**
- **You Can Initialize Arrays with a Size, But They Are No Longer Fixed**
  - You must dimension arrays in declaration before **ReDim**

```
Dim i As Integer = 21
Dim dToday As Date = Today( )

'Array declarations
Dim Month(12) As Integer  'Creates array with 13 elements
'Initialize the array with 12 elements
Dim aMonth( ) As Integer = {1,2,3,4,5,6,7,8,9,10,11,12}
```

In Visual Basic .NET, you can use a different process to declare some types of variables, including arrays and strings. For example, you can declare and initialize variables in a single statement.

For Your Information
Object declaration and initialization are covered in Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

Declaring and Initializing Variables

In Visual Basic .NET, you can initialize a variable when you declare it by using the following syntax:

```
Dim [WithEvents] varname[([subscripts])] [As [New] type]
[= initexpr]
```

Most of this syntax is familiar to Visual Basic developers. However, there is a new optional **initexpr** argument that allows you to assign an initial value to a variable as long as the argument is not used in conjunction with the **New** keyword.

Examples

The following code shows how to declare and initialize variables in a single statement:

Delivery Tip
Explain the examples on the slide.

```
Dim i As Integer = 21
Dim dToday As Date = Today( )
Dim dblFloat As Double = 1232.23312
Dim dBirthday As Date = #1/1/1995#
Dim iCalculate As Integer = i * 5
```

Declaring and Initializing Arrays

You use a slightly different syntax to declare and initialize arrays. This syntax allows you to specify not only the size of the array but also the initial values for it.

In Visual Basic .NET, all arrays must have a lower bound value of zero. You cannot declare an array by using the *lower bound* `To` *upper bound* syntax as you do in Visual Basic 6.0. In the following example, the *Month* variable is created with 13 elements, as it is in previous versions of Visual Basic. The *aMonth* variable, however, creates and initializes an array of precisely 12 elements.

```
Dim Month(12) As Integer
Dim aMonth( ) As Integer = {1,2,3,4,5,6,7,8,9,10,11,12}
```

Redimensioning Arrays

In Visual Basic 6.0, you can only redimension an array if it is not dimensioned when it is declared. In Visual Basic .NET, you can redimension an array if it is dimensioned when it is declared.

Visual Basic 6.0

The following code shows how to redimension an array in Visual Basic 6.0:

```
Dim x( ) As String
ReDim x(5) As String    'Correct in Visual Basic 6.0

Dim y(2) As String
ReDim y(5) As String    'Error in Visual Basic 6.0 because you
                        'cannot redim a dimensioned array
```


Visual Basic .NET

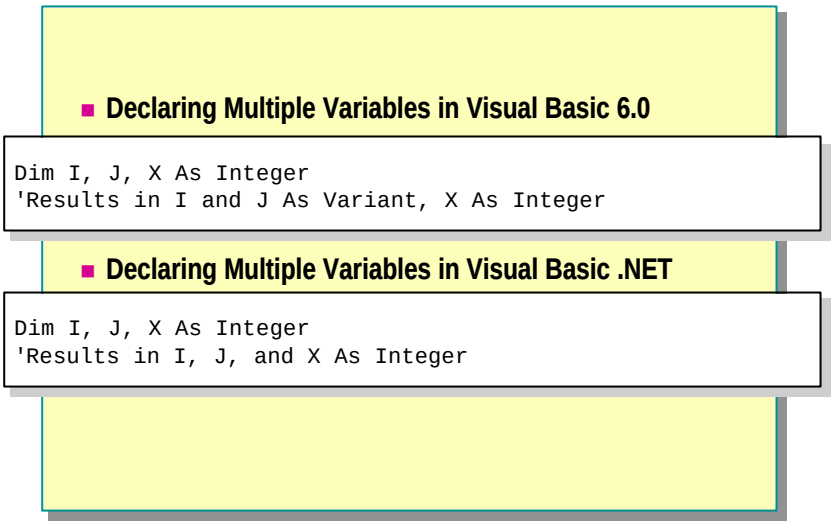
The following code shows how to redimension an array in Visual Basic .NET:

```
Dim x( ) As String
ReDim x(5)          'Correct in Visual Basic .NET
Dim y(2) As String
ReDim Preserve y(5)         'Allowed in Visual Basic .NET
```

Declaring Multiple Variables

Topic Objective
To explain how to declare multiple variables in a single statement.

Lead-in
When you declare multiple variables in one line of code, Visual Basic .NET interprets the code differently than previous versions of Visual Basic.



In Visual Basic 6.0, you can use a single line of code to declare multiple variables, but you may get unexpected results. Consider the following example:

```
Dim I, J, X As Integer
```


Visual Basic 6.0

In Visual Basic 6.0, I and J are created as **Variants**, and X is created as an **Integer** data type.

Visual Basic .NET

In Visual Basic .NET, all three variables are created as **Integers**. This is consistent with how many other programming languages create multiple variables and is more intuitive.

Variable Scope

Topic Objective
To explain the concept of block scope and how it is implemented in Visual Basic .NET.

Lead-in
Visual Basic .NET introduces a new level of variable scope into the Visual Basic language: block scope.

- **Procedure Scope**
  - Variables accessible to entire procedure
- **Block Scope**
  - Variables only accessible within that block
  - Lifetime of block variable is entire procedure

```
Dim iLooper As Integer    'Procedure level variable

For iLooper = 1 to 10
    Dim iMax As Integer  'Block level variable
    iMax = iLooper
Next
MsgBox (iMax)     'This line generates a compiler error
```

In Visual Basic 6.0, if you declare variables inside a block of code, they are accessible to the entire procedure that contains the block. This level of accessibility is referred to as *procedure scope*. In Visual Basic .NET, variables inside a block of code are only accessible to that block of code. This level of accessibility is referred to as *block scope*.

Example

Consider an example in which you need procedure scope for your variables.

Visual Basic 6.0

The following code executes successfully in Visual Basic 6.0, because the *iMax* variable has procedure scope. In Visual Basic .NET, the last line of code generates a compile error because the *iMax* variable has block scope and is only accessible in the **For …Next** loop.

Delivery Tip
Explain how the sample code would work in Visual Basic 6.0, and then explain how it would work in Visual Basic .NET.

```
Dim iLooper As Integer
For iLooper = 1 to 10
    Dim iMax As Integer
    iMax = iLooper
NextMsgBox (iMax)

'The last line generates a compiler error in Visual Basic .NET
```

Visual Basic .NET

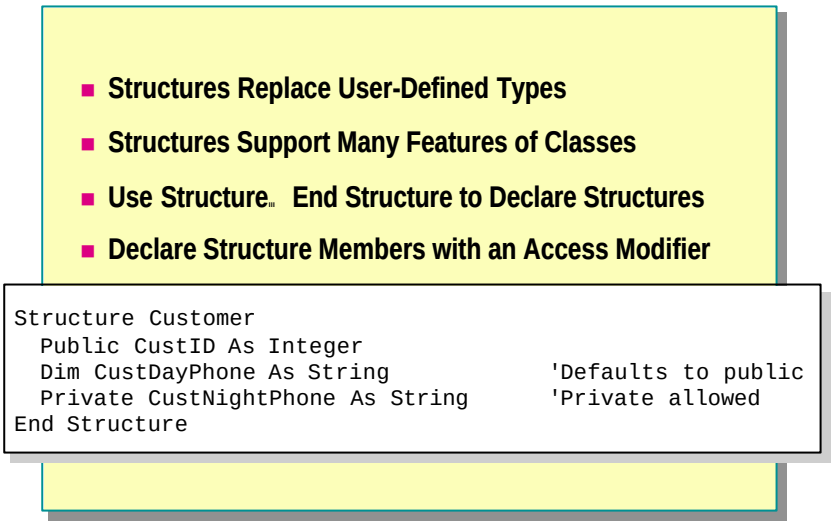
If you rewrite the code as follows, it will execute successfully in Visual Basic .NET.

```
Dim iLooper As Integer
Dim iMax As Integer
For iLooper = 1 to 10
    iMax = iLooper
NextMsgBox (iMax)
```

Creating Data Structures

Topic Objective
To explain how to create your own data types by using data structures.

Lead-in
Data structures replace user-defined types in Visual Basic .NET.



In Visual Basic 6.0, you create user-defined types (UDTs) by using **Type… End Type** syntax. In Visual Basic .NET, you create your own data types by creating data structures. To create a data structure, you use the **Structure… End Structure** syntax.

Delivery Tip
Point out that structures and access modifiers will be covered in more detail in Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

The members of UDTs can only contain **Public** data types. Internal members of a data structure can contain **Public**, **Friend**, or **Private** data types. Therefore, you must declare internal members of a structure with one of these access modifiers, as shown in the following code:

```
Structure Customer
  Public CustID As Integer
  Dim CustDayPhone As String          'Defaults to public
  Private CustNightPhone As String    'Private allowed
End Structure
```

The syntax for using structures and classes in Visual Basic .NET is very similar. In fact, structures support most features of classes, including methods.

Note For more information about data structures and access modifiers, see Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

Compiler Options

Topic Objective
To explain how setting compiler options affects data types.

Lead-in
Setting various compiler options can alter the way your data types behave.

- **Option Explicit**
  - Default option
- **Option Strict**
  - Enforces strict type semantics and restricts implicit type conversion
  - Late binding by means of the **Object** data type is not allowed
- **Option Base 1 Not Supported**
  - Arrays must start at zero

The compiler options that you select affect many parts of your application. Two options directly influence how your data types will behave and how you should use them: **Option Explicit** and **Option Strict**. You set these options as **On** or **Off** at the beginning of a module by using the following code:

```
Option Explicit On
Option Strict Off
```


Delivery Tip
Point out that if **Option Explicit** is left off in previous versions of Visual Basic, spelling mistakes can create variant variables instead of causing a compiler error.

Option Explicit

This option is on by default in Visual Basic .NET. When **Option Explicit** is enabled, you must explicitly declare all variables before using them. Undeclared variables generate a compiler error.

Without this option, you may accidentally create unwanted variables as a result of spelling mistakes or other errors.

Option Strict

Option Strict is a new compiler option in Visual Basic .NET that controls whether variable type conversions are implicit or explicit. This option prevents the data inaccuracies that may result from implicit narrowing conversions.

If you select this option, implicit widening type conversion, such as converting an **Integer** to a **Long,** is allowed. However, implicit narrowing type conversions, such as converting a numeric **String** to an **Integer**, or a **Long** to an **Integer**, cause a compiler error.

The following example shows that assigning a **Double** value to an **Integer** variable causes a compiler error with **Option Strict** enabled, because of implicit narrowing. However, assigning an **Integer** value to a **Long** variable will not cause an error because this is implicit widening.

```
Dim i As Integer, i1 As Double, lng As Long
i1 = 12.3122
i = i1       'Causes a compiler error
i = 256
lng = i      'No error because widening is acceptable
```

The following example shows a subroutine that takes an **Integer** argument but is passed a **String** value, resulting in a compiler error:

```
Sub TestLong(ByRef lng As Long)
    ...
End Sub
TestLong("1234")
'Causes a compiler error because narrowing is unacceptable
```

Late binding is not allowed under **Option Strict**. This means that any variable declared `As Object` can only use the methods provided by the **Object** class. Any attempt to use methods or properties belonging to the data type stored in the variable will result in a compiler error.

The following example shows what will happen if you use late binding when **Option Strict** is enabled. A **String** value in an **Object** variable is allowed, but calling a method from the **String** class is not allowed.

```
Dim x As Object
x = "MyStringData"

'Attempt to retrieve a character fails
MsgBox(x.Chars(4))
```

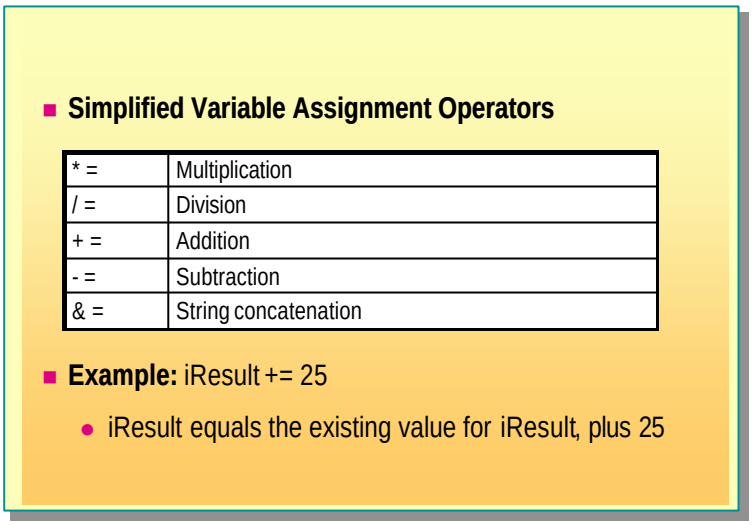

Option Base 1

In Visual Basic .NET, all arrays must start with a lower bound of 0. Therefore, **Option Base 0|1** is not a compiler option in Visual Basic .NET. This is consistent with all programming languages using the .NET Framework.

Assignment Operators

Topic Objective
To explain the new assignment operators.

Lead-in
Visual Basic .NET supports a simplified syntax for assignment operations.



Visual Basic .NET provides a shorthand syntax that you can use to assign values to variables. The standard assignment operators are still valid; the new syntax is optional.

Syntax

The original syntax and the shorthand version are shown below:

Original: `{variable} = {variable} {operator} {expression}`
Shorthand: `{variable} {operator} = {expression}`

For example:

Original: `iResult = iResult + 25`
Shorthand: `iResult += 25`

Shorthand Operators

The following table shows how the compiler will interpret the new shorthand operators.

Assignment operator	Purpose
*=	Multiplies the value of a variable by the value of an expression and assigns the result to the variable.
/=	Divides the value of a variable by the value of an expression and assigns the result to the variable.
+=	Adds the value of a variable to the value of an expression and assigns the result to the variable. Can also be used for string concatenation.
-=	Subtracts the value of a variable from the value of an expression and assigns the result to the variable.
&=	Concatenates a string variable with the value of an expression and assigns the result to the variable.
^=	Raises the value of a variable to the power of an exponent and assigns the result to the variable.
\=	Divides the value of a variable by the value of an expression and assigns the integer result to the variable.

Example

The following example shows how to use the new assignment operators to concatenate character strings and provides the resulting string:

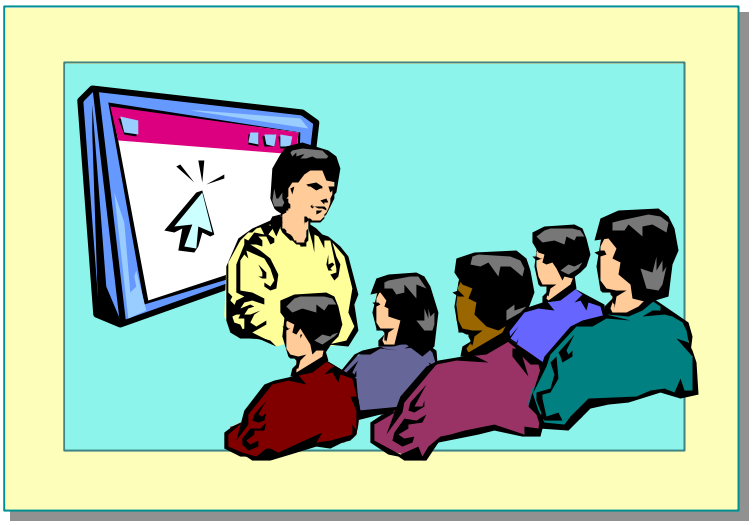
```
Dim myString As String = "First part of string; "
myString &= "Second part of string"

MsgBox (myString)
'Displays "First part of string; Second part of string"
```

Demonstration: Using Variables and Data Structures

Topic Objective
To demonstrate how to declare and initialize variables and data structures.

Lead-in
In this demonstration, you will learn how to declare and initialize variables, including basic data types, arrays, and data structures.



Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to declare and initialize different data types, including some basic data types, arrays, and data structures. You will also learn how to use block-scoped variables.

◆ Functions, Subroutines, and Properties

Topic Objective
To describe the changes relating to using functions, subroutines, and default object properties.

Lead-in
The way you create and use functions and subroutines and the way you use default object properties have changed in Visual Basic .NET.

- Calling Functions and Subroutines
- Passing Arguments ByRef and ByVal
- Optional Arguments
- Static Function and Static Sub
- Returning Values from Functions
- Using Default Properties

Delivery Tip
This lesson covers creating and using procedures, but it only covers using default object properties. For more information about creating properties, see Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

After you complete this lesson, you will be able to work with functions, subroutines, and default properties in Visual Basic .NET.

Calling Functions and Subroutines

Topic Objective
To explain the new syntax for calling functions and subroutines.

Lead-in
In Visual Basic .NET, the rules for when to use parentheses when calling functions and subroutines are much simpler than in previous versions of Visual Basic.

- **Visual Basic 6.0**
  - You must follow complex rules regarding use of parentheses
  - You must use parentheses when using a return value from a function
- **Visual Basic .NET**
  - You must use parentheses to enclose the parameters of any function or subroutine
  - You must include empty parentheses for procedures without parameters

---

Delivery Tip
Point out that the inconsistent requirements for parentheses in previous versions of Visual Basic are often difficult for inexperienced developers.

In Visual Basic .NET, the syntax that you use to call a procedure is different from the syntax used in Visual Basic 6.0.

Visual Basic 6.0

When calling a procedure, you must follow a complex set of rules regarding the use of parentheses. You must use them when you are using a return value from a function. In other circumstances, use of parentheses will change the passing mechanism being used.

Visual Basic .NET

You must use parentheses to enclose the parameters of any function or subroutine. If you are calling a procedure without supplying any parameters, you must include empty parentheses. The following statements show how to call a subroutine that has parameters:

```
DisplayData(1, 21)   'Subroutine
Call DisplayData(1, 21)
```

Passing Arguments ByRef and ByVal

Topic Objective
To explain the default mechanism for passing arguments in Visual Basic .NET.

Lead-in
There are some important differences between the Visual Basic 6.0 and Visual Basic .NET mechanisms for passing parameters.

- **Visual Basic 6.0**
  - **ByRef** is the default passing mechanism
- **Visual Basic .NET**
  - **ByVal** is the default passing mechanism

Delivery Tip
Check whether the students understand the difference between **ByRef** and **ByVal.** If necessary, explain the differences on a whiteboard or flip chart.

When you define a procedure, you can choose to pass arguments to it either by reference (**ByRef**) or by value (**ByVal**).

If you choose **ByRef**, Visual Basic passes the variable's address in memory to the procedure, and the procedure can modify the variable directly. When execution returns to the calling procedure, the variable contains the modified value.

If you choose **ByVal,** Visual Basic passes a copy of the variable to the procedure. If the procedure modifies the copy, the original value of the variable remains intact. When execution returns to the calling procedure, the variable contains the same value that it had before it was passed.

There are some important differences between the Visual Basic 6.0 and Visual Basic .NET mechanisms for passing parameters.

Visual Basic 6.0

- **ByRef** is the default passing mechanism.

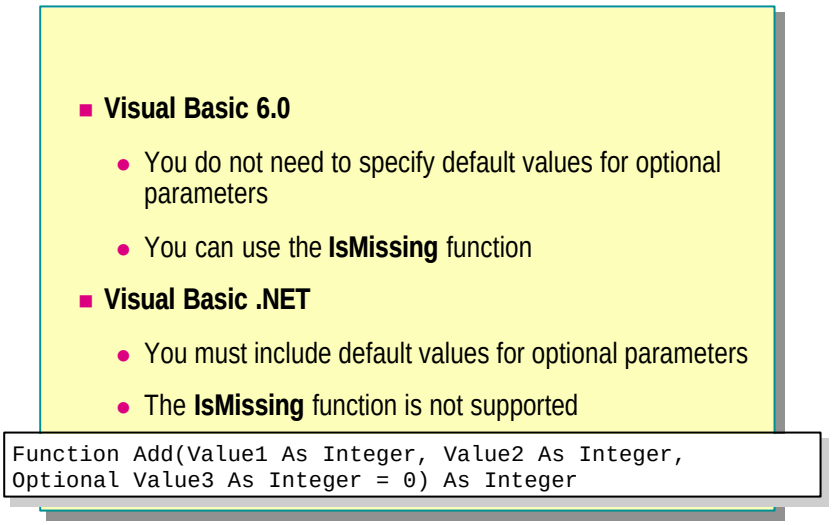
Visual Basic .NET

- **ByVal** is the default passing mechanism, and is automatically added to parameter definitions if you do not specify either ByVal or ByRef.

Optional Arguments

Topic Objective
To explain changes in how you use optional arguments.

Lead-in
There are some changes to how you use optional arguments in Visual Basic .NET.



Optional arguments allow you to choose whether or not to pass all parameters to a function or subroutine. There are some changes to how you use optional arguments in Visual Basic .NET.

Visual Basic 6.0

- You do not need to specify default values for optional parameters.
- You can use the **IsMissing** function to verify that the parameters have been passed to the procedure, if arguments are declared as **Variant**.

Visual Basic .NET

- You must include default values for optional parameters.
- The **IsMissing** function is not supported.

The following example shows how to declare an argument as optional in Visual Basic .NET.

```
Function Add(Value1 As Integer, Value2 As Integer, Optional
Value3 As Integer = 0) As Integer
```

Note You can use overloaded functions to provide the same functionality as optional arguments. For more information about overloading, see Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

Static Function and Static Sub

Topic Objective
To explain how to declare static variables in Visual Basic .NET.

Lead-in
Static variables must be explicitly declared in Visual Basic .NET.

- **Visual Basic 6.0**
  - You can place **Static** in front of any **Function** or **Sub** procedure heading
  - Local variables in a static function or static subroutine retain their values between multiple calls
- **Visual Basic .NET**
  - Static functions and static subroutines are not supported
  - You must explicitly declare all static variables

Delivery Tip
Check whether students understand static variables, and provide an example of a static counter variable if required.

Static variables are declared differently in Visual Basic .NET.

Visual Basic 6.0

- You can place **Static** before any **Sub** or **Function** procedure heading. This makes all the local variables in the procedure static, regardless of whether they are declared with **Static**, **Dim**, or **Private**, or are declared implicitly.
- Local variables in a static function or static subroutine retain their values between multiple calls to the function or subroutine.

Visual Basic .NET

- Static functions and static subroutines are not supported.
- You must explicitly declare all static variables.

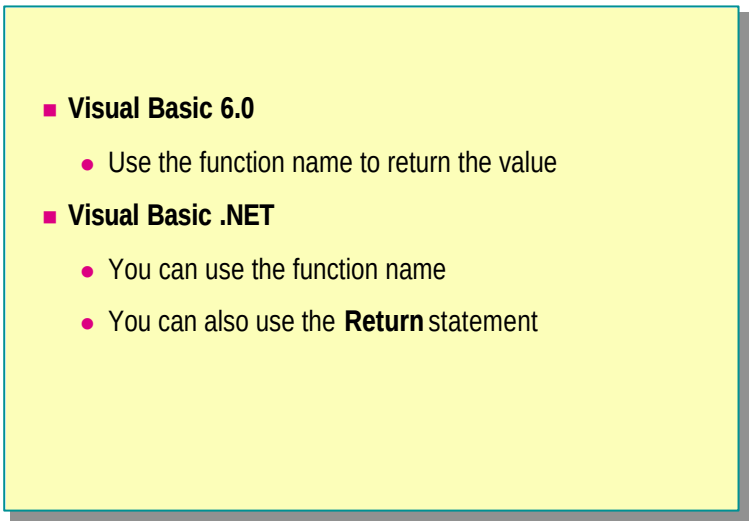
The following example shows how to use a static variable:

```
Dim iLooper As Integer
Static iMax As Integer
For iLooper = 1 To 10
iMax += 1
Next
MsgBox(iMax)
```

Returning Values from Functions

Topic Objective
To explain how values are returned from functions in Visual Basic .NET.

Lead-in
Visual Basic .NET provides flexibility in how you can return values from functions.



Visual Basic .NET provides flexibility in how you can return values from functions.

Visual Basic 6.0

Use the function name to return the value.

Visual Basic .NET

You can use the function name to return the value. The following example shows how to use the function name to return the value:

```
Function GetData( ) As String
    ...
    GetData = "My data"
End Function
```

You can also use the **Return** statement to return the value. This avoids linking the return of the function to the function name, allowing for easier renaming of functions. The following example shows how to use the **Return** statement to return the value:

```
Function GetData( ) As String
    ...
    Return "My data"
End Function
```

Note The **Return** statement exits the function immediately and returns the value to the calling procedure.

Using Default Properties

Topic Objective
To explain how you can use default properties in Visual Basic .NET.

Lead-in
The syntax for calling the default properties of an object has been updated in Visual Basic .NET.

- **Visual Basic 6.0**
  - Supports default properties on most objects
  - Use **Set** to determine whether assignment is referring to the object or the default property
- **Visual Basic .NET**
  - Supports default properties only for parameterized properties
  - Do not need to differentiate between object and default property assignments
  - Default properties are commonly used to index into collections

Delivery Tip
The syntax for calling properties other than the default property has not changed. It is only the default properties that have been altered.

Explain that **Set** is provided in Visual Basic 6.0 to determine whether code is referring to the object or the default property; therefore, it has been removed in Visual Basic .NET.

Also note that property creation is covered in Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

Visual Basic .NET updates default property support for simplification and improved readability.

Visual Basic 6.0

- Default properties are supported on most objects. For example, the **Text** property of a **TextBox** control is defined as the default property, meaning you can call the property without having to specify the property name.
- To allow this feature, the **Set** keyword is provided to distinguish between using the object itself for assignment and using the object's default property for assignment.

Visual Basic .NET

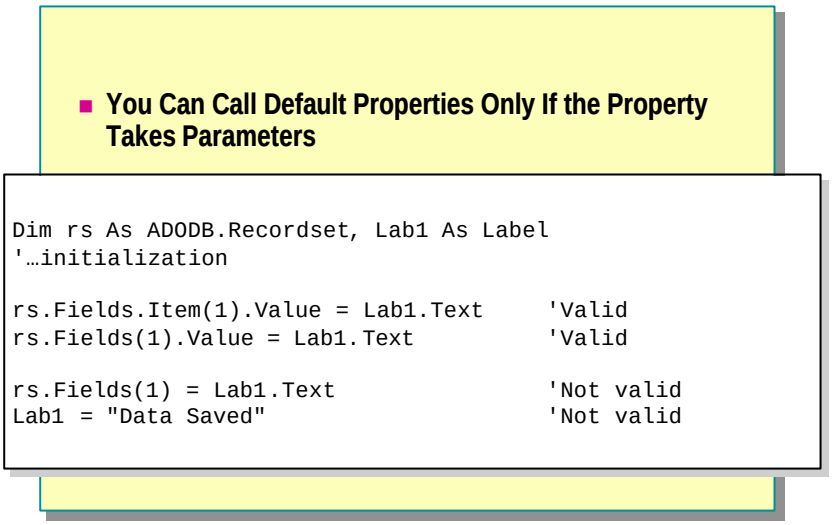
- You can only mark a property as default if it takes parameters.
- You specify a property as the default property by starting its declaration with the **Default** keyword.
- Default properties are commonly used to index into collections, such as the ADO Recordset's **Fields.Item** collection.

Note **Let** is still a reserved word in Visual Basic .NET, even though it has no syntactical use. This helps avoid confusion with its former meanings. **Set** is used in Visual Basic .NET for property procedures that set the value of a property.

Using Default Properties (*continued*)

Topic Objective
To examine correct and incorrect syntax for using default properties.

Lead-in
Here are some examples showing correct and incorrect syntax for using default properties.



The following examples show valid and invalid syntax for using default properties:

Delivery Tip
Explain every step of the example and ask why each line is valid or invalid.

```
Dim rs As ADODB.Recordset, Lab1 As Label
'…initialization

rs.Fields.Item(1).Value = Lab1.Text
'Valid because no defaults used
rs.Fields(1).Value = Lab1.Text
'Valid because Item is parameterized

rs.Fields(1) = Lab1.Text
'Not valid because Value is not parameterized
Lab1 = "Data Saved"
'Not valid because Text is not parameterized
```

Lab 3.1: Working with Variables and Procedures

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will declare and initialize variables, and create subroutines and functions.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Declare and initialize variables.
- Create and call functions and subroutines.

Prerequisites

Before working on this lab, you must be familiar with using variables, arrays, and procedures.

Scenario

In this lab, you will create a simple application that is based on Microsoft Windows® in which you can enter customer information into an array and then retrieve it. The application will consist of a single form that you use to input this information.

Solution Files

There are solution files associated with this lab. The solution files are in the *install folder*\Labs\Lab031\Solution folder.

Estimated time to complete this lab: 45 minutes

Exercise 1
Creating the Customer Form

In this exercise, you will create the customer entry form.

↙ To create a new project

1. Open Visual Studio .NET.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Project Types** box, click the **Visual Basic Projects** folder.
4. In the **Templates** box, click **Windows Application**.
5. Save the project as Lab031 in the *install folder*\Labs\Lab031 folder, and then click **OK**.

↙ To create the customer form

1. In Solution Explorer, open the design window for Form1.vb.
2. In the Properties window, set the **Text** property of the form to **Customer**.
3. Add controls to the form, as shown in the following screen shot:

4. Set the properties of the controls as shown in the following table.

Control	Property name	Property value
Label1	**Text**	**First Name:**
	Name	**lblFirstName**
Label2	**Text**	**Last Name:**
	Name	**lblLastName**
Label3	**Text**	**Date of Birth:**
	Name	**lblDOB**
TextBox1	**Text**	**<empty>**
	Name	**txtFirstName**
TextBox2	**Text**	**<empty>**
	Name	**txtLastName**
TextBox3	**Text**	**<empty>**
	Name	**txtDOB**
Button1	**Text**	**Add Customer**
	Name	**btnAddCustomer**
Button2	**Text**	**Retrieve**
	Name	**btnRetrieve**

5. Save the project.

Exercise 2
Adding a Customer

In this exercise, you will write code to add a new customer to an array of customers when the user clicks **Add Customer**.

↙ To create the module-level variables

1. Create a private structure called **Customer** after the **Inherits System.Windows.Forms.Form** statement within the **Public Class** code block by using the information in the following table.

cCustomer member	Data type
Id	**Integer**
FirstName	**String**
LastName	**String**
DateOfBirth	**Date**

2. Declare a private array called **aCustomers** to hold **Customer** elements with an initial size of one.

↙ To add a customer

1. Create the **btnAddCustomer_Click** event handler.
2. In the **btnAddCustomer_Click** event handler, create a local variable named *cCustomer* based on the information in the following table.

Variable name	Data type
cCustomer	**Customer**

3. Assign the upper bound limit of the **aCustomers** array to the Id member of the **cCustomer** object.
4. Assign the **Text** properties of the text boxes to the corresponding members of the *cCustomer* variable as defined in the following table.

   Use the **CDate** function to convert the text property of **txtDOB** to the **Date** data type for use by the **cCustomer.DateOfBirth** member.

cCustomer member	Text box
FirstName	txtFirstName
LastName	txtLastName
DateOfBirth	txtDOB

5. Using the **UBound** function for the array index, add the *cCustomer* variable to the **aCustomers** array.

6. Use the **ReDim Preserve** syntax and the **UBound** function to increase the size of the **aCustomers** array by one.

   This creates one more array element than is required. However, this is acceptable for this exercise.

Important When you use the **UBound** function to increase the size of the array, you must add the integer value of 1 to the result of the **UBound** function.

---

7. Use the **MsgBox** function to display a message box that confirms that the customer has been added.
8. Clear the **txtFirstName**, **txtLastName**, and **txtDOB** text boxes.
9. Save the project.

↙ To test your application

1. On the first line of the **btnAddCustomer_Click** event handler, set a breakpoint.
2. On the **Debug** menu, click **Start**.
3. Enter customer details into the text boxes.
4. To add a customer, click **Add Customer**.
5. To step through the code, on the **Debug** menu, click **Step Into**.
6. To open the Locals window, on the **Debug** menu, point to **Windows**, and click **Locals**. View the values of the variables as you step through the code.

Exercise 3
Retrieving a Customer

In this exercise, you will write code to retrieve a customer from an array when the user clicks **Retrieve**.

↙ To create the RetrieveCustomer function

1. At the end of the form definition, add a new private function named **RetrieveCustomer**.

   This function takes one argument by value, as defined in the following table. It returns a **Customer** structure.

Argument name	Data type
iIndex	**Integer**

2. Return the **Customer** object stored in the **iIndex** position of the **aCustomers** array as the result of the function.

↙ To call the RetrieveCustomer function

1. In the **btnRetrieve_Click** event handler, declare three local variables as defined in the following table.

Variable name	Data type
aCustomer	**Customer**
sInput	**String**
sMessage	**String**

2. Use the **InputBox** function to ask the user to enter a customer identification number, and then store the response in the *sInput* variable.
3. Use an **If** statement and the **IsNumeric** function to test whether the entered data is numeric.
4. If the data is numeric, call the **RetrieveCustomer** function, and then pass it the value of the *sInput* variable converted to an integer.
5. Store the return value of the **RetrieveCustomer** function in the *aCustomer* variable.
6. To create a message to be displayed to the user, concatenate the values of each of the *aCustomer* elements into the *sMessage* variable, and then display the string in a message box.
7. Save the project.

↙ To test your application

1. On the **Debug** menu, click **Clear All Breakpoints**.
2. On the first line of the **btnRetrieve_Click** event handler, set a breakpoint.
3. Start the application, and then add three customers of your choice.
4. To step through the code, click **Retrieve**.
5. In the InputBox, type **1** and then confirm that the correct customer information is displayed.

   You should see details for the second customer that you entered.
6. Quit the application.
7. On the **Debug** menu, click **Clear All Breakpoints**, and then save your project.
8. Quit Visual Studio .NET.

◆ Exception Handling

Topic Objective
To explain the extensions to exception handling in Visual Basic. NET.

Lead-in
Exception handling is an important topic in any application.
Visual Basic .NET introduces a powerful new form of handling: structured exception handling.

- **Structured Exception Handling**
- **Try... Catch... Finally**
- **Using Try... Catch... Finally**
- **The System.Exception Class**
- **Filtering Exceptions**
- **Throwing Exceptions**

In this lesson, you will learn about the extensions to error handling (or exception handling) in Visual Basic .NET. After completing this lesson, you will be able to:

- Explain the advantages of the new exception handling system by comparing unstructured handling to structured handling.
- Use the **Try…Catch..Finally** statement in conjunction with the **System.Exception** class to implement structured exception handling.
- Create your own exceptions by using the **Throw** statement.

Structured Exception Handling

Topic Objective
To introduce structured exception handling and its advantages.

Lead-in
Structured exception handling is new to Visual Basic .NET and offers many advantages over unstructured exception handling.

- **Disadvantages of Unstructured Error Handling**
  - Code is difficult to read, debug, and maintain
  - Easy to overlook errors
- **Advantages of Structured Exception Handling**
  - Supported by multiple languages
  - Allows you to create protected blocks of code
  - Allows filtering of exceptions similar to **Select Case** statement
  - Allows nested handling
  - Code is easier to read, debug, and maintain

Visual Basic developers are familiar with unstructured exception handling in the form of the **On Error** statement. With **On Error**, developers can check for and handle exceptions in several different ways by using exception labels, **GoTo** statements, and **Resume** statements.

Disadvantages of Unstructured Exception Handling

Delivery Tip
Point out that you can still use **On Error** syntax, but that structured handling will be a better choice, as will be discussed in subsequent topics.

Unstructured exception handling can make your code difficult to read, maintain, and debug, and may lead you to unintentionally ignore an error. For example, when you use the **On Error Resume Next** statement, you must remember to check the **Err** object after each action that can cause an error. If you do not check the value after each action, you may miss an initial error when a subsequent action also fails. This means you may handle an error incorrectly or unintentionally ignore an error.

The Visual Basic language has been criticized because it lacks structured exception handling. Visual Basic .NET addresses this criticism by supporting structured exception handling, using the syntax **Try…Catch…Finally**.

Advantages of Structured Exception Handling

Structured exception handling is used in many programming languages, such as Microsoft Visual C++® and Microsoft Visual C#™, and combines protected blocks of code with a control structure (similar to a **Select Case** statement) to filter exceptions. Structured exception handling allows you to protect certain areas of code. Any exceptions in code that you leave unprotected are raised to the calling procedure, as in Visual Basic 6.0.

Delivery Tip
You will look at the **Try**... **Catch**... **Finally** syntax in detail on the following slides.

Provide an example of when you may want to nest an exception handler within the **Catch** block of another handler. This might happen when an exception has occurred and you need to test some code that may create another exception.

You can filter exceptions by using the **Catch** block, which provides functionality similar to a **Select Case** statement in Visual Basic 6.0. This allows you to filter multiple exceptions in the same way that a **Select Case** can handle outcomes from a comparison.

You can also nest exception handlers within other handlers as needed (in the same procedure or in a calling procedure), and variables declared within each block will have block-level scope.

It is easier to create and maintain programs with structured exception handling. The flow of execution is easy to follow and does not require jumps to non-sequential code.

The old style of error handling is still supported in Visual Basic. NET. The only restriction is that you can't mix both styles of error handling in the same procedure.

Try… Catch… Finally

Topic Objective
To explain the basics of the **Try… Catch… Finally** syntax.

Lead-in
So how do you use the **Try… Catch… Finally** syntax?

```
...
Try
' Include code to be tried here
' Can use Exit Try to exit block and resume after End Try

Catch
' Define exception type and action to be taken
' Can use series of statements (multiple error handling)

Finally
' Optional block
' Define actions that need to take place

End Try
...
```

Delivery Tip
At this stage students might not know anything about class inheritance, so do not attempt to explain too much about how exceptions can inherit from the **System.Exception** class. Simply point out that there are multiple exception classes that are based on the main **System.Exception** class.

You can implement structured exception handling in Visual Basic .NET by using the **Try…Catch…Finally** statement.

Syntax

The following code shows the structure of a simple **Try…Catch…Finally** statement:

```
Try
' Include code to be tried here
' You can use Exit Try to exit the code and resume after End
Try
Catch
' Define the exception type and the action to be taken
' You can use a series of statements (multiple error handling)
Finally
' This block is optional
' Define actions that need to take place
End Try
```

Try Block

Note the following as you examine this code:

- The **Try…End Try** block surrounds an area of code that might contain an error.
- Code placed in this block is considered protected.
- If an exception occurs, processing is transferred to the nested **Catch** blocks.
- You can use the **Exit Try** keyword to instantly exit the **Try…End Try** block. Execution will resume immediately after the **End Try** statement.

Catch Block

If an exception occurs in the **Try** block, execution will continue at the beginning of the nested **Catch** block. The **Catch** block is a series of statements beginning with the keyword **Catch** followed by an exception type and an action to be taken. The following are some guidelines for using the **Catch** block:

- You can choose to handle all exceptions in one **Catch** block. You can also declare multiple **Catch** blocks to filter the exception and handle particular errors, similar to how you might use **Select Case** in previous versions of Visual Basic.
- You can filter using the different exception classes defined by the .NET Framework and runtime, or by using your own exception classes.
- You can use a **When** statement to compare the exception to a particular exception number.
- If you use filtering for the exceptions but do not handle the actual exception that occurred, the exception is automatically raised up to the calling procedure (or to the user if no calling procedure exists). However, by using a **Catch** filter with the **Exception** class, you will catch all of the other exceptions that you have not included in your filters. This is the equivalent of a **Case Else** statement in a **Select Case** structure.

Finally Block

The **Finally** block is optional. If you include this block, it is executed after the **Try** block if no errors occurred, or after the appropriate **Catch** block has been processed.

- The **Finally** block is always executed.
- In this block, you can define actions that need to take place regardless of whether an exception occurs. This may include actions such as closing files or releasing objects.
- The **Finally** block is most often used to clean up operations when a method fails.

Using Try… Catch… Finally

Topic Objective
To explain a simple example of structured exception handling.

Lead-in
Let's look at a simple example of **Try… Catch… Finally** syntax.

```
Sub TrySimpleException
  Dim i1, i2, iResult As Decimal
  i1 = 22
  i2 = 0
  Try
    iResult = i1 / i2      ' Cause divide-by-zero error
    MsgBox (iResult)       ' Will not execute

  Catch eException As Exception  ' Catch the exception
    MsgBox (eException.Message)  ' Show message to user

  Finally
    Beep
  End Try
End Sub
```

The following example shows how to implement structured exception handling in Visual Basic .NET by using the **Try…Catch…Finally** syntax:

```
Sub TrySimpleException
  Dim i1, i2, iResult As Decimal
  i1 = 22
  i2 = 0
  Try
    iResult = i1 / i2   ' Cause divide by zero exception
MsgBox (iResult)      ' Will not execute

  Catch eException As Exception  ' Catch the exception

    MsgBox (eException.Message)  ' Show message to user
  Finally
    Beep
  End Try
End Sub
```

The compiler processes this code as follows:

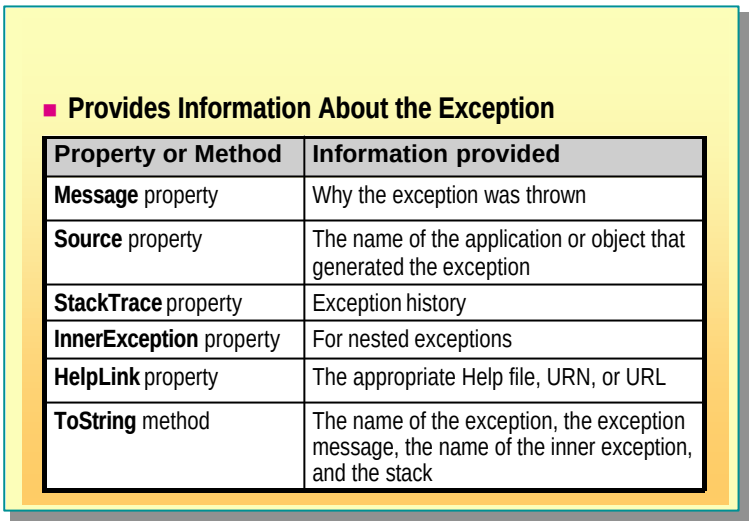
1. Processing begins by attempting the code in the **Try** block.
2. The code creates a divide-by-zero exception.
3. Execution passes to the **Catch** block, where a variable *eException* of type **Exception** class is declared. This variable will display information about the exception to the user.
4. The **Finally** code is executed after all processing in the **Catch** block is complete. The **Finally** code causes a beep to sound, signifying that processing is complete.

Note Any variables declared in any of the three blocks are scoped as block-level variables. They cannot be accessed from outside of the block.

The System.Exception Class

Topic Objective
To examine the various properties and methods of the **System.Exception** class.

Lead-in
The **System.Exception** class provides information about an exception.



The **System.Exception** class in Visual Basic .NET, similar to the **Err** object in Visual Basic 6.0, provides information about a particular exception. When you use this class in your **Catch** blocks, you can determine what the exception is, where it is coming from, and whether there is any help available.

Some of the most useful properties and methods of the **System.Exception** class are described in the following table.

Property or method	Description
Message property	Use the **Message** property to retrieve information about why an exception was thrown. A generic message is returned if the exception was created without a particular message.
Source property	Use the **Source** property to retrieve the name of the application or object that generated the exception.
StackTrace property	Use the **StackTrace** property to retrieve the stack trace of the exception as a string.
InnerException property	Use the **InnerException** property to navigate to multiple nested exceptions. Nesting exceptions may be useful if a more specific (or general) exception needs to be generated while maintaining the information from the original exception. If only the original exception is required, use the **GetBaseException** method.
HelpLink property	Use the **HelpLink** property to retrieve the appropriate Help file, URN, or URL for the exception. (See the note following this table.)
ToString method	Use the **ToString** method to return the fully qualified name of the exception, the exception message (if there is one), the name of the inner exception, and the stack trace.

Note Uniform Resource Locators (URLs) and Uniform Resource Names (URNs) are both examples of Uniform Resource Identifiers (URIs). A URN is a unique identifier that is not necessarily (but can be) in the form of a URL. They can be any combination of characters that is unique. Large organizations are more likely than individuals to use URNs because the guarantee of uniqueness is more difficult to achieve.

Filtering Exceptions

Topic Objective
To explain a more advanced example of **Try...Catch... Finally** syntax that filters exceptions.

Lead-in
Let's look at a more advanced example of **Try...Catch...Finally** syntax that involves filtering.

```
Dim x, y, z As Integer, bSucceeded As Boolean = True
Try
    'Perform various operations on variables
    ...
Catch eException As DivideByZeroException
    MsgBox("You have attempted to divide by zero.")
    bSucceeded = False
Catch eException As OverflowException
    MsgBox("You have encountered an overflow.")
    bSucceeded = False
...
Catch When Err.Number = 11
    MsgBox("Error occurred.")
    bSucceeded = False
Finally
    If bSucceeded Then
        ...
    End If
End Try
```

Delivery Tip
This example is more complex because it uses filtering to catch different exceptions.

Explain that **DivideByZeroException**, **OverflowException**, and **ConstraintException** are all examples of exception classes that are based on (inherit from) the **System.Exception** class.

One of the **Catch** blocks checks the exception number by using a **When** statement.

Also point out that the last **Catch** block uses the **ToString** method of the exception class.

To learn more about structured exception handling in Visual Basic .NET, consider a more advanced example. In this example, errors are filtered based on the class of the exception.

Example

The following example shows how to use filtering to handle several different exceptions in one **Try...Catch...Finally** statement:

```
Sub TryComplexException( )
  Dim x, y, z As Integer, bSucceeded As Boolean = True

  Try
      'Perform various operations on variables
      ...
  Catch eException As DivideByZeroException
      MsgBox("You have attempted to divide by zero!")
      bSucceeded = False
  Catch eException As OverflowException
      MsgBox("You have encountered an overflow.")
      bSucceeded = False
  Catch eException As ConstraintException
      MsgBox(eException.ToString)
      bSucceeded = False
  Catch When Err.Number = 11
      MsgBox("Error occurred")
      bSucceeded = False
  Finally
      If bSucceeded Then
          MsgBox("Success!")
      Else
          MsgBox("Failure")
      End If
  End Try
End Sub
```

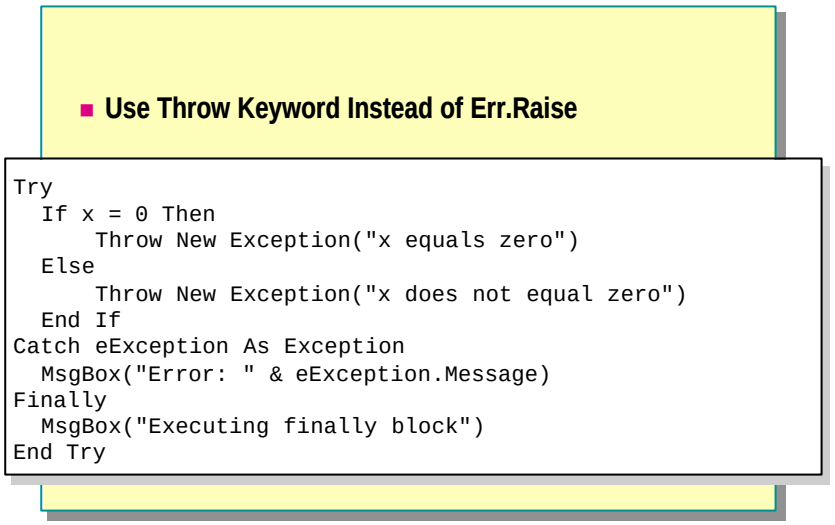

As you examine this code, note the following:

- For demonstration purposes, a test is made against various exception classes such as **DividebyZeroException**, **OverflowException**, and **ConstraintException**. These classes are all derived from the **System.Exception** class.
- One of the **Catch** blocks checks the exception number, **Err.Number**, by using the **When** statement.
- The last **Catch** block uses the **ToString** method of the **Exception** class.
- Note that if the exception does not meet any of the filter expressions, it will be passed up to the calling procedure. Using **System.Exception** as a Catch type would catch other unexpected exceptions.

Throwing Exceptions

Topic Objective
To explain how to throw your own exceptions.

Lead-in
You can raise your own exceptions by using the **Throw** statement.



Delivery Tip
You can also create your own exception classes that inherit from the **System.Exception** class. For more information, see Module 4, "Object-Oriented Design for Visual Basic .NET" and Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

In Visual Basic 6.0, you can use the **Raise** method of the **Err** object to raise your own exceptions. You can use this method to create a business logic error or to propagate an error after previously trapping it.

Visual Basic .NET introduces the **Throw** statement, which allows you to create your own exceptions. The **Throw** statement provides similar functionality to the **Err.Raise** method.

Example

The following example shows how to throw an exception in Visual Basic .NET:

```
Try
  If x = 0 Then
     Throw New Exception("x equals zero")
  End If
Catch eException As Exception
  MsgBox("Error: " & eException.Message)

End Try
```

Delivery Tip
This is an example of using object constructors for the **Exception** class. They will be covered in Module 4, "Object-Oriented Design for Visual Basic .NET" and Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

This example will throw an exception if the value of the variable *x* is zero. The **If** statement creates a new **Exception** object and passes a string containing an exception description to the object constructor. This means that the **Catch** block can handle the exception as it would deal with a normal system exception.

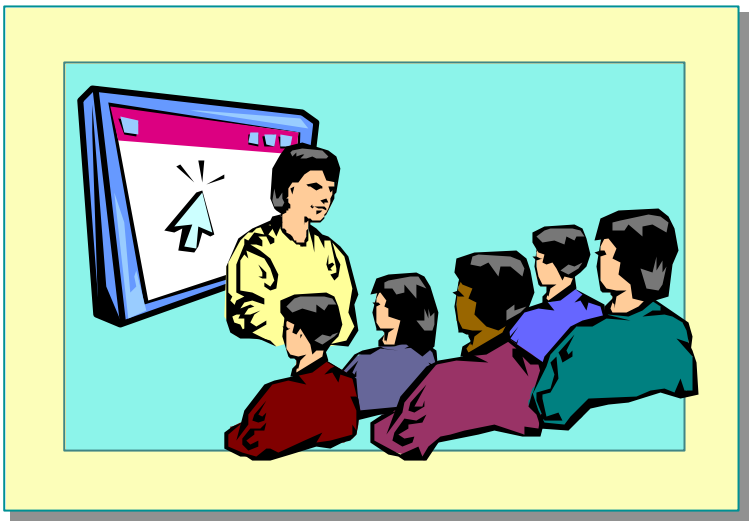
If a **Throw** statement is not executed within a **Try** block, the exception will be raised to the calling procedure.

Note For more information about object constructors, see Module 5, " Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

Demonstration: Structured Exception Handling

Topic Objective
To demonstrate how to use structured exception handling.

Lead-in
In this demonstration, you will learn how to use structured exception handling in Visual Basic .NET.



Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to use the **Try…Catch..Finally** statement to implement structured exception handling. You will also learn how to check values of the **System.Exception** class and how to throw your own exceptions.

Lab 3.2: Implementing Structured Exception Handling

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will add structured exception handling to your procedures from Lab 3.1.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create structured exception handling.
- Throw your own exceptions.

Prerequisites

Before working on this lab, you must:

- Complete Lab 3.1.
- Be familiar with using the **Try…Catch…Finally** statement for structured exception handling.

Scenario

In this lab, you will add structured exception handling to the application that you created in Lab 3.1.

Starter and Solution Files

There are starter and solution files associated with this lab. The starter files are in the *install folder*\Labs\Lab032\Starter folder, and the solution files are in the *install folder*\Labs\Lab032\Solution folder.

Estimated time to complete this lab: 30 minutes

Exercise 1
Adding Exception Handling to Customer Retrieval

In this exercise, you will add structured exception handling to the **RetrieveCustomer** function that you created in Lab 3.1.

↙ Open the previous project

1. Open Visual Studio .NET.
2. Open the project from Lab 3.1. If you did not complete Lab 3.1, use the project in the *install folder*\Labs\Lab032\Starter folder.

↙ To add exception handling to the btnRetrieve_Click event handler

1. In the **btnRetrieve_Click** event handler, add a **Try…Catch…Finally** code block around the code that calls the **RetrieveCustomer** function, excluding the variable declarations.
2. Create a new procedure-level **Boolean** variable named *bSuccess*, and then initialize it to the value of **True** on the same line.
3. In the **Catch** block, create a variable named *eException* of the data type **Exception**.

   This catches an exception when a user tries to access an array element that does not exist.
4. In a message box inside the **Catch** block, display the **Message** property from the *eException* variable, and then set the *bSuccess* variable to **False**.
5. In the **Finally** block, create an **If** statement to test the *bSuccess* variable for a value of **True**.
6. Locate the code that concatenates the *sMessage* variable and the **MsgBox** function, and then perform a cut-and-paste operation to place this code inside the **If** block.
7. Save the project.

↙ To test your application

1. On the first line of the **btnRetrieve_Click** event handler, set a breakpoint.
2. Run the application, and add only one customer.
3. Click **Retrieve**.

   When you enter break mode, step through the code.
4. When asked for the customer identification number, enter the value **20** in the **InputBox**.
5. Confirm that this generates an exception in the **RetrieveCustomer** function that is caught by the exception handling in the **btnRetrieve_Click** event handler.
6. End the application.

↙ To add exception handling to the RetrieveCustomer function

1. In the **RetrieveCustomer** function, add a **Try…Catch…Finally** code block around the existing code.
2. In the **Catch** block, create a variable named *eOutofRange* of type **IndexOutOfRangeException**.

   This catches an exception when a user tries to access an array that does not exist.
3. Add the following line to the **Catch** block:

   ```
   Throw New Exception ("Invalid Customer Id", eOutOfRange)
   ```

   This throws a new exception that includes a specific message, while keeping the original exception as an inner exception. The **Try...Catch...Finally** block in the **btnRetrieve_Click** event handler catches this exception.
4. Delete the **Finally** block.

   It serves no purpose in this procedure.

↙ To display the inner exception in the btnRetrieve_Click event handler

1. In the **btnRetrieve_Click** event handler, modify the **Catch** block to display additional information about the exception, including the **Message**, the **ToString**, and the **GetBaseException.Message** members of the *eException* variable.
2. Save your project.

↙ To test your application

1. Start the application, and add only one customer.
2. Click **Retrieve**.
3. When asked for the **Customer Id**, enter the value **20** in the **InputBox**.
4. Confirm that this generates an exception in the **RetrieveCustomer** function and that it is handled inside the function, but then is raised to the **btnRetrieve_Click** event handler.
5. Close Visual Studio .NET.

Review

Topic Objective
To reinforce module objectives by reviewing key points.

Lead-in
The review questions cover some of the key concepts taught in the module.

- **Data Types**
- **Using Variables**
- **Functions, Subroutines, and Properties**
- **Exception Handling**

1. Declare and initialize an array that contains the following strings: “one”, “two”, “three”, “four”.

```
Dim myArray( ) As String = {"one", "two", "three", "four"}
```

2. What types of variables are created by the following declaration if **Option Strict** is off?

```
Dim a, b As Integer, c
```

The variables a and b are created as Integers; c is created as an Object.

3. What is the value of *c* after the following code executes:

```
Dim c As Integer
c = 1
CheckValue(c)
...
Sub CheckValue(ByVal iValue As Integer)
    ...
    iValue = 13
End Sub
```

The variable c remains equal to 1 because the default passing mechanism is by value.

4. Assuming you have an open **Recordset** called rs and a **TextBox** control called txtData, which of the following statements will create a compiler or run-time error with **Option Strict** off? Why?

```
a. txtData.Text = rs(0)
b. txtData.Text = rs.Fields.Item(0)
c. txtData.Text = rs.Fields(0).Value
```

Statement (a) will fail because `rs(0)` returns a Field object; the Fields collection is the default property of the Recordset object. This cannot be assigned to the Text property of the txtData TextBox because the data types are not compatible.

Statement (b) will fail because Value is the default property of the Field object, and it does not take a parameter. This causes the compiler to attempt to assign the Field object to the txtData.Text property, resulting in an error.

Statement (c) will succeed because Item is the default property of the Fields object, and it does take a parameter.

5. What is the method or property of the **System.Exception** class that retrieves the most information about an exception?

The ToString method provides the fully qualified class name, and the error message (if available), the name of the inner exception, and the stack trace of the exception.