1418835463_AppendixA.qxd 5/22/06 02:31 PM Page 879

APPENDIX A

A

DEBUGGING

After completing this appendix, you will be able to:

- Describe the types of programming errors
- Trace statement execution
- Set breakpoints in code
- Use the Immediate window to evaluate expressions
- Add watch expressions
- Trace cascading events with the Call Stack window
- Use the Locals window

1418835463_AppendixA.qxd 5/22/06 02:31 PM Page 880

DEBUGGING APPLICATIONS

This appendix describes how to use the Visual Studio debugging tools. In any application that you create, you will likely make mistakes. Finding those mistakes and being able to effectively correct them is an important part of the development process. This appendix discusses numerous techniques to help you locate and fix common programming errors.

TYPES OF PROGRAMMING ERRORS

Any error that causes an application to end abnormally or to produce unexpected results is considered a programming error. The process of locating and fixing programming errors in an application is called **debugging**. Visual Studio supports numerous built-in tools that help you to debug the applications that you create.

Many developers feel that debugging an application is part science and part art. The science involves being able to use the Visual Studio debugging tools effectively. These tools include windows that allow you to watch statements as they execute, to examine the current value of variables, and to use the commands that execute statements one at a time. The art involves deciding where to search for the exact cause of an error and choosing which Visual Studio tools to use to find the error.

»NOTE

A common term used to describe any kind of programming error is "a bug."

Programming errors can be categorized into three different types: syntax errors, run-time errors, and logic errors, which are discussed in the following sections.

SYNTAX ERRORS

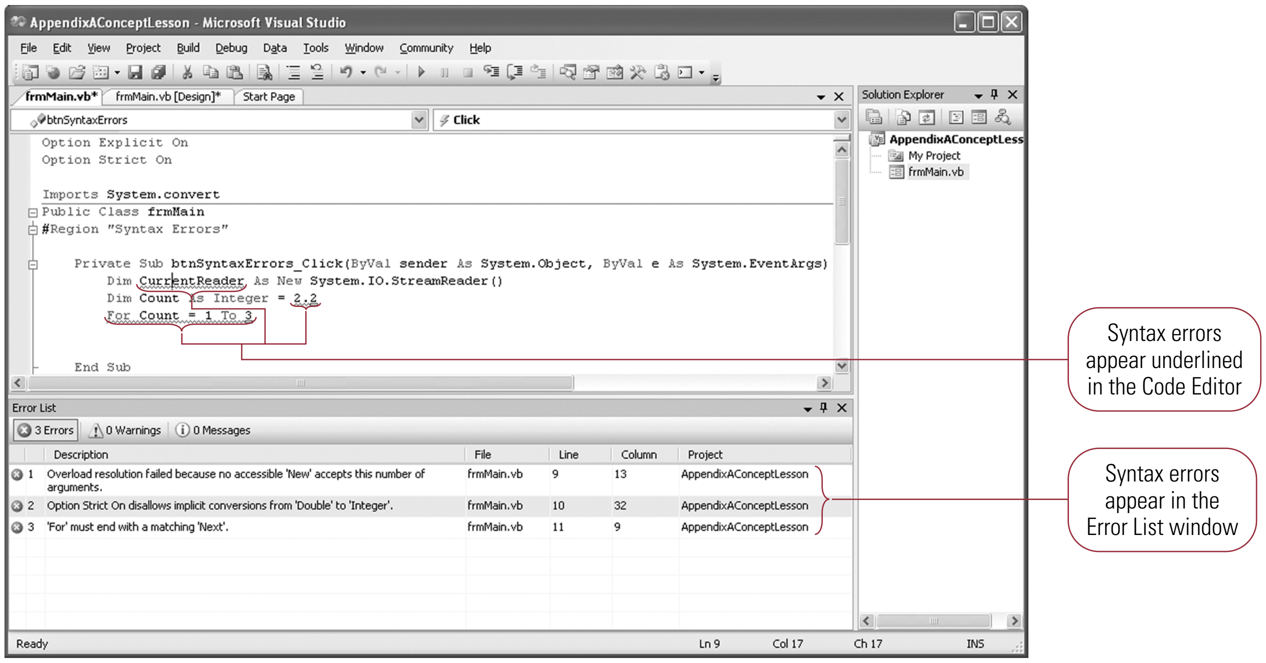
As each statement of an application is entered into the Code Editor, the Visual Basic compiler examines the statement for syntax errors (violations of the rules of a programming language) as the insertion point is moved from one statement to the next. A syntax error occurs when the Visual Basic compiler cannot understand a statement that you have written. If an application has syntax errors, the application cannot be compiled or executed.

You can find syntax errors in two ways:

- The Visual Basic compiler underlines syntax errors with a ragged blue line in the Code Editor.
- Syntax errors are listed in the Error List window.

1418835463_AppendixA.qxd 5/22/06 02:31 PM Page 881

Figure A-1 shows the Code Editor and Error List windows displaying various syntax errors.


AppendixAConceptLesson - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
frmMain.vb* frmMain.vb [Design]* Start Page
btnSyntaxErrors
Click
Solution Explorer
AppendixAConceptLess
My Project
frmMain.vb
Syntax errors appear underlined in the Code Editor
Syntax errors appear in the Error List window
Ready Ln 9 Col 17 Ch 17 INS

```
Option Explicit On
Option Strict On

Imports System.convert
Public Class frmMain
#Region "Syntax Errors"

    Private Sub btnSyntaxErrors_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
        Dim CurrentReader As New System.IO.StreamReader()
        Dim Count As Integer = 2.2
        For Count = 1 To 3


    End Sub
```

Error List — 3 Errors | 0 Warnings | 0 Messages

		Description	File	Line	Column	Project
⊗	1	Overload resolution failed because no accessible 'New' accepts this number of arguments.	frmMain.vb	9	13	AppendixAConceptLesson
⊗	2	Option Strict On disallows implicit conversions from 'Double' to 'Integer'.	frmMain.vb	10	32	AppendixAConceptLesson
⊗	3	'For' must end with a matching 'Next'.	frmMain.vb	11	9	AppendixAConceptLesson

Figure A-1: Syntax errors appearing in the Code Editor and Error List windows

As shown in Figure A-1, three syntax errors appear in the Code Editor with ragged underlines. The same syntax errors appear in the Error List window. The columns appearing in the Error List window provide the following information:

» The first column displays an icon showing the severity of the error. In Figure A-1, all three errors are critical and prevent the application from compiling.

» The second column shows the error number. Errors are numbered sequentially.

» The third column contains a description of the syntax error.

» The fourth column displays the name of the file that contains the syntax error.

» The fifth and sixth columns display the line and column position where the error was detected.

» The final column displays the project where the error appeared.

»NOTE

The Error List window also displays warning errors. For example, if you declare a variable but do not use that variable, a warning error appears in the Error List window. Warning errors do not prevent an application from compiling.

1418835463_AppendixA.qxd 5/22/06 02:31 PM Page 882

In this exploration exercise, you will use the Code Editor and Error List windows to locate and correct syntax errors.

1. Start Visual Studio, if necessary, and open the solution appearing in the folder named **Appendix.A\AppendixAConceptLesson**.
2. Activate the Code Editor for the form named **frmMain**.
3. Activate the Error List window by clicking **View**, **Error List** on the menu bar. By default, the Error List window is docked along the bottom of the Visual Studio IDE.
4. Expand the Syntax Errors region, if necessary. At the beginning of the module, modify the statement that reads `Private Count1 As Integer` so that it reads **Pirvite Count1 As Integer** (misspelling the keyword Private).
5. Move the insertion point to the next line so that Visual Studio checks the syntax of the statement you just modified. In the Code Editor, the statement with the incorrect syntax appears underlined. In addition, a message appears in the Error List window with the description "Declaration expected." indicating that only declaration statements can appear at the module level.
6. Correct the error by spelling the keyword **Private** correctly, and then move the insertion point to the next line. The syntax of the statement is checked. Because the syntax of the statement is correct, the error is removed from both the Code Editor and the Error List window.

»TIP

Syntax errors can cascade. That is, one syntax error can cause Visual Studio to identify additional syntax errors. Thus, when an application contains multiple syntax errors, fix the first syntax error in the list first, as it might be the cause of subsequent syntax errors.

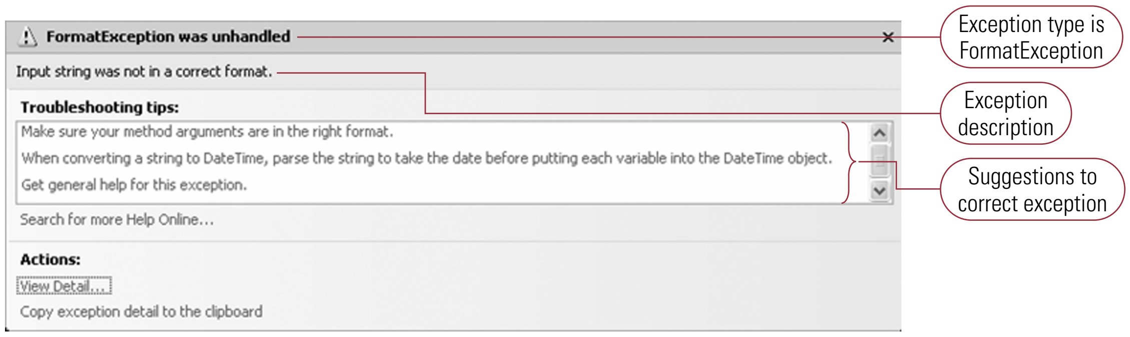
RUN-TIME ERRORS

Unlike syntax errors, run-time errors are not detected by the Visual Basic compiler when the application is compiled. Rather, these errors occur as an application executes and an exception is thrown.

Chapter 7 describes exceptions and how to handle them. Exceptions can be thrown for many reasons, such as the following:

- Trying to store data of an incompatible data type in a property or variable
- Calling functions with arguments having invalid data
- Trying to store too large or too small a value into a variable (numeric overflow)
- Trying to reference an object before creating an instance of the class with the `New` keyword

If a statement that is not enclosed in an exception handler executes and causes an exception to be thrown, a run-time error occurs. When an exception is thrown, Visual Studio displays a message box to help explain the cause of the exception and to help you locate the statement that caused the exception. Figure A-2 shows an unhandled exception appearing in a dialog box.


FormatException was unhandled

Input string was not in a correct format.

Troubleshooting tips:

Make sure your method arguments are in the right format.

When converting a string to DateTime, parse the string to take the date before putting each variable into the DateTime object.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

Exception type is FormatException
Exception description
Suggestions to correct exception

Figure A-2: Unhandled exception message

As shown in Figure A-2, the dialog box shows the exception type, which, in this case, is a `System.FormatException`. It means that Visual Studio executed a statement but could not convert the textual input data to a numeric value. The dialog box displays a descriptive message, along with tips to solve the problem.

NOTE In some cases, the statement that caused an exception might be only a symptom of the underlying problem. Instead, the exception might be thrown at a later point during the execution of the program. In such cases, you might want to examine the statements that lead up to an exception, even though these statements themselves are not directly responsible for the exception.

Exceptions can be thrown when an application is running and are often caused by type mismatch errors and numeric overflow and underflow errors. These errors are discussed in the following sections.

TYPE MISMATCH AND FORMATEXCEPTION ERRORS

Calling a method of the `System.Convert` class with incorrect data causes an exception to be thrown. For example, a `FormatException` is thrown if a string cannot be converted to a numeric value, as shown in the following code segment:

```
Dim StringDemo As String = "$1234.56"
Dim DoubleDemo As Double = _
    System.Convert.ToDouble(StringDemo)
```

The second of the preceding statements causes a `FormatException` to be thrown because the value stored in the variable StringDemo cannot be converted to the `Double` data type.

This type of error can be corrected in two ways. First, you can write code to validate the contents of a string variable or text box using the `IsNumeric` and `IsDate` functions. Second, any statements that might cause a type mismatch error or format error can be enclosed in a structured exception handler.

»NOTE

When performing arithmetic operations on floating-point numbers, Visual Studio does not raise an exception in the case of numeric overflow. Rather, Visual Studio stores a special value (not a number) in the variable.

NUMERIC OVERFLOW ERRORS

Numeric overflow exceptions are caused by executing statements that attempt to store a value that is too large or too small into a variable having an integral data type. The `Long`, `Integer`, and `Short` data types are all subject to numeric overflow errors. For example, the largest value that can be stored in the `Short` data type is 32,767. Trying to store a value larger than that causes an exception to be thrown. The same problem occurs when an attempt is made to store too large a negative value in a variable. For example, the `Short` data type cannot store a negative value beyond –32,768.

Numeric overflow and underflow errors are resolved by creating structured exception handlers. In this exploration exercise, you will examine how execution of a statement can cause an exception to be thrown.

1. Run the solution for this appendix. By default, the Run-Time Errors tab should be active. Figure A-3 shows the Run-Time Errors tab.

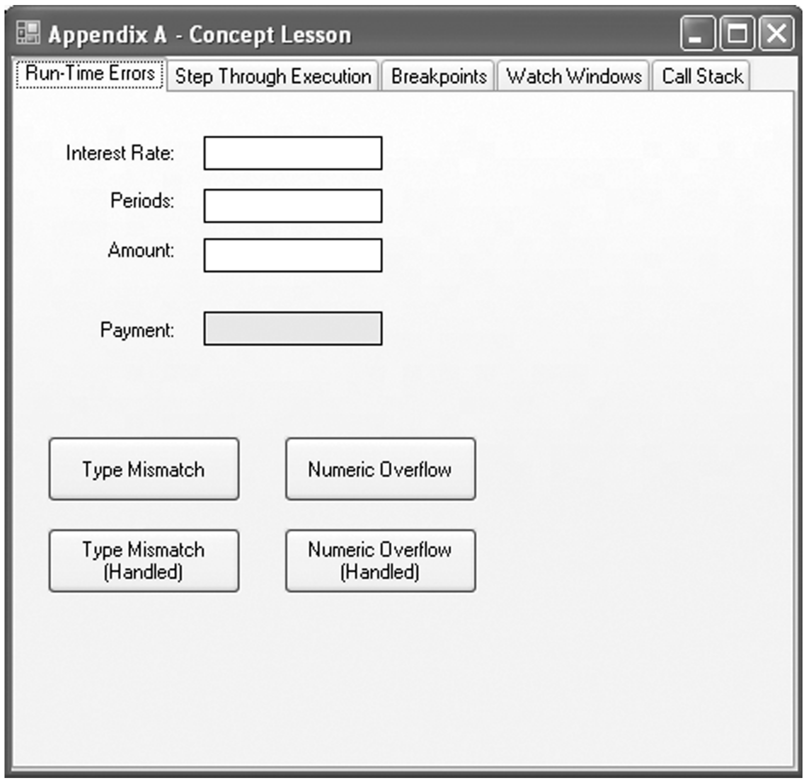


Figure A-3: Concept lesson—Run-Time Errors tab

1418835463_AppendixA.qxd 5/22/06 02:32 PM Page 885

2. Enter the value **.25A** for the interest rate, **15000A** for the period, and **10000A** for the amount. Each of these values is invalid and causes an exception to be thrown when the executing statement tries to convert the invalid value to a numeric data type.
3. Click the **Type Mismatch** button. A dialog box opens describing the error. The error is categorized as a `System.FormatException` error because the value of an argument is not valid. The statement causing the exception is also highlighted in the Code Editor.
4. End the program and run it again. Enter the same invalid input into the text boxes. Click the **Type Mismatch (Handled)** button. This time, an exception handler appears in the event handler. This exception handler displays a message box describing the nature of the error. Click **OK** to close the message box.
5. Click the **Numeric Overflow** button. A multiplication operation is performed causing a `System.OverflowException` to be thrown.
6. End the program and run it again. Reenter the invalid values from Step 2. Click the **Numeric Overflow (Handled)** button. The same multiplication operation is performed, but this time, the statement that causes the exception is enclosed in an exception handler. The statement in the exception handler displays a message box to the end user.
7. End the application.

LOGIC ERRORS

A **logic error** occurs when an application does not perform as it is expected to perform, but instead produces incorrect results. In such a situation, logic errors will surface while an application is running. For example, if you intended to compute the area of a rectangle, you would multiply the length of the rectangle by its width. If you added the numbers instead of multiplying them, the application would produce an incorrect answer, and a logic error would have been created. The Visual Studio debugging tools cannot solve logic errors directly. That is, the debugging tools cannot tell you that a statement contains a logic error. Rather, you must use the debugging tools to locate and correct any logic errors that you yourself create.

NOTE The distinction between logic and run-time errors is sometimes unclear to those new to programming. For example, a logic error would occur if you added two values together instead of multiplying them. If this error causes a numeric overflow exception to be thrown, the logic error would, in turn, cause a run-time error.

For example, if you called the `PMT` method with incorrect argument values, the method call might cause an exception to be thrown. You would first need to determine which argument was incorrect, and then locate the statement that set the value of the erroneous argument.

1418835463_AppendixA.qxd 5/22/06 02:32 PM Page 886

INTRODUCING THE VISUAL STUDIO DEBUGGING TOOLS

Visual Studio's debugging tools consist of commands that allow you to temporarily suspend the execution of an application by entering *break mode*. You can then trace the execution of statements and procedures as they are called by Visual Studio. Statements can be executed line-by-line with Visual Studio suspending execution as each statement executes. It is also possible to suspend execution just before executing a specific statement or when the value of a variable or a property's value changes.

"Stepping through the statements" in an application means that the Visual Studio run-time system executes each statement in a procedure line-by-line, highlighting each statement just before executing it. As you step through the statements in an application, you typically examine the variables and object properties to locate the cause of logic or run-time errors.

The Visual Studio debugging tools are made up of several windows that are used together to find errors in an application and to subsequently correct them. These windows are collectively referred to as **debugging windows**. The following list identifies selected debugging windows that are discussed in this appendix. These debugging windows are all tool windows and can be docked or Auto Hidden along an edge of the IDE. The debugging tool windows can also appear as floating windows.

» The *Breakpoints window* is used to define the locations (executable statements) in an application where Visual Studio will suspend execution. After execution is suspended, it is common to examine the values of variables so as to determine the cause of a particular error. In addition, the Code Editor and the buttons on the Debug toolbar can be used to execute statements one at a time.

» In the *Immediate window*, expressions can be entered that display the values of variables and object properties. It is also possible to call procedures using the Immediate window.

» *Watch windows* are used to examine the values of expressions.

» The *Call Stack* window is used to examine the procedures that have been called and the order in which those procedures have been called.

» The *Locals window* is used to examine the values of local variables, or for example, the properties of a form, its control instances, and the variables declared in the form.

»TIP When debugging an application, you will likely have several windows open simultaneously, which might cause windows to obscure one another. Consider Auto Hiding the debugging windows along the bottom of the IDE. To Auto Hide a window, dock the window along an edge of the IDE. Right-click the window's title bar, and then click Auto Hide.

TRACING EXECUTION AND SETTING BREAKPOINTS WITH THE BREAKPOINTS WINDOW

Often, the applications you create contain logic errors. That is, the application produces incorrect results but does not cause any exceptions to be thrown. When an application produces incorrect results, but the reason is not clear, it might prove helpful to step through the statements; that is, it might be helpful to follow the execution of each statement line-by-line until the problem is found. It might also be helpful to execute statements line-by-line to help determine why an exception is thrown.

The following list describes the toolbar buttons used to step through statements as Visual Studio executes them.

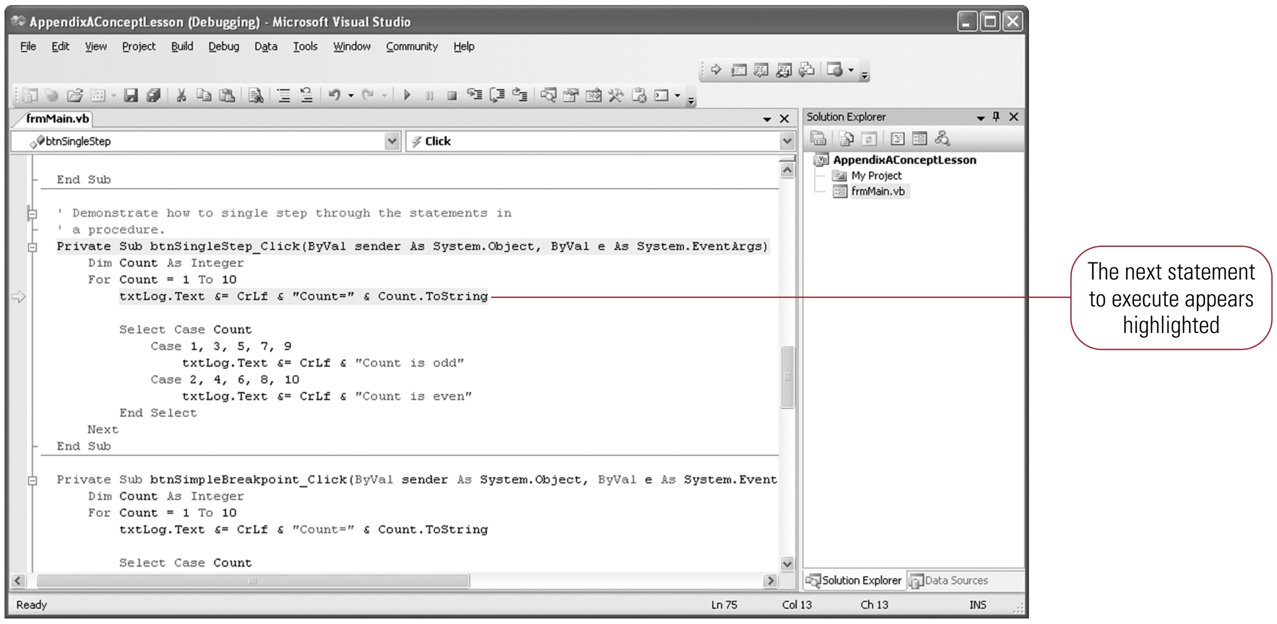
- Visual Studio executes one statement, and then enters break mode when the Step Into button is clicked. The *Step Into* button traces execution as procedures are called and as they complete. If a statement is a procedure call, the statements in the called procedure are executed line-by-line.
- The *Step Over* button works similarly to the Step Into button. If the statement is a procedure call, however, Visual Studio executes all of the statements in the procedure, and then enters break mode just before executing the statement following the procedure call.
- The *Step Out* button works similarly to the Step Over button. When clicked, Visual Studio executes all of the remaining statements in the current procedure, and then enters break mode at the statement following the statement that called the procedure.

In this exploration exercise, you will trace the execution of statements line-by-line using the Step Into button. The `Click` event handler for this button contains the following code to determine whether a number is even or odd:

```
Dim Count As Integer
For Count = 1 To 10
    txtLog.Text &= CrLf & "Count=" & Count.ToString()
    Select Case Count
        Case 1, 3, 5, 7, 9
            txtLog.Text &= CrLf & "Count is odd"
        Case 2, 4, 6, 8, 10
            txtLog.Text &= CrLf & "Count is even"
    End Select
Next
```

1. Click **View**, **Toolbars**, **Debug**, if necessary, to display the Debug toolbar. Click the **Step Into** button or press **F11** to begin executing the application.
2. Click the **Step Through Execution** tab. Click the **Single Step** button on the form. The Code Editor appears and the procedure declaration appears highlighted in yellow, indicating that the procedure is about to execute.

3. Press **F11** several times to execute each statement line-by-line. Pay particular attention to the fact that the statement that will execute next is highlighted in the Code Editor. Watch the statements in the loop as they execute. Furthermore, watch when the different `Case` blocks execute depending on whether the value of the variable Count is even or odd. Figure A-4 shows the Code Editor while single stepping through an application. Note that the Output Log text box on the main form shows the results of statements as they execute.


AppendixAConceptLesson (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
frmMain.vb
btnSingleStep
Click
Solution Explorer
AppendixAConceptLesson
My Project
frmMain.vb
Solution Explorer
Data Sources
Ready
Ln 75
Col 13
Ch 13
INS
The next statement to execute appears highlighted

```
End Sub

' Demonstrate how to single step through the statements in
' a procedure.
Private Sub btnSingleStep_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Dim Count As Integer
    For Count = 1 To 10
        txtLog.Text &= CrLf & "Count=" & Count.ToString

        Select Case Count
            Case 1, 3, 5, 7, 9
                txtLog.Text &= CrLf & "Count is odd"
            Case 2, 4, 6, 8, 10
                txtLog.Text &= CrLf & "Count is even"
        End Select
    Next
End Sub

Private Sub btnSimpleBreakpoint_Click(ByVal sender As System.Object, ByVal e As System.Event
    Dim Count As Integer
    For Count = 1 To 10
        txtLog.Text &= CrLf & "Count=" & Count.ToString

        Select Case Count
```

Figure A-4: Line-by-line statement execution in the Code Editor

4. As you watch the statements execute, highlight the variable **Count** by moving the cursor over the variable. Note that the value of the variable appears in a ToolTip.
5. Press **F5** or click the **Step Out** button. The process of single stepping ends and execution continues as normal.
6. End the application.

Clicking the Step Into button causes Visual Studio to step through the statements in a procedure. If a statement is a procedure call, Visual Studio single steps into the procedure, and then continues executing the statements in that procedure one statement at a time.

In addition to stepping through every statement in every procedure in an application, it is possible to step through parts of an application or pause the application (enter break

1418835463_AppendixA.qxd 5/22/06 02:32 PM Page 889

mode), and then continue executing statements one at a time. When debugging a procedure that calls other procedures, for instance, it is not necessary to trace through the statements in a called procedure after it is known to work correctly. Instead, the Step Over button can be used to execute all of the statements in a procedure, and then suspend execution at the statement following the one that called the procedure. Furthermore, execution can be suspended at any time by clicking the Break All button while the application is running.

»TIP

While debugging an application, it is often useful to enter break mode, and then step through each statement that you suspect is in error. When a procedure appears to work correctly, you can step over it and move on to the next statement. This is where setting breakpoints comes into play.

TYPES OF BREAKPOINTS

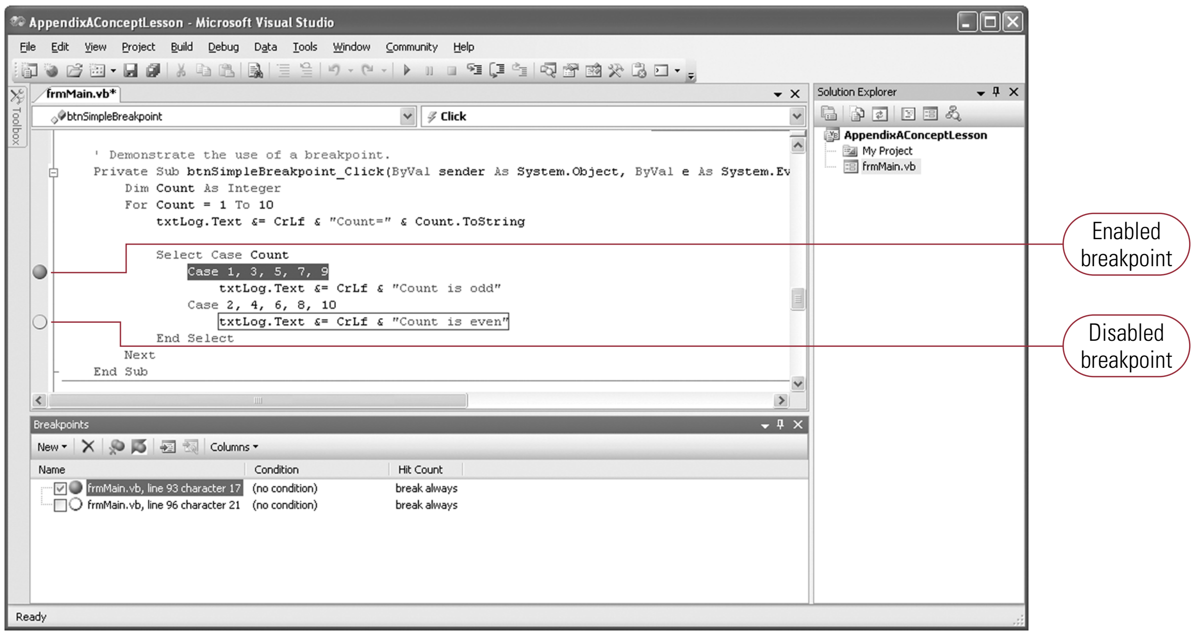
When you suspect that a problem is occurring in a particular procedure or that a particular statement is incorrect, it is possible to suspend execution at any executable statement by setting a breakpoint. A **breakpoint** is an executable statement where Visual Studio suspends execution and enters break mode, just before executing the marked statement.

Breakpoints are one of the most common and easy to use of the Visual Studio debugging tools. Breakpoints can be created and deleted as needed so as to examine the values of variables at a particular point in an application's execution. Visual Studio supports different types of breakpoints. In this appendix, you will examine a type of breakpoint called a file breakpoint. A file breakpoint is an executable statement in an application where execution will be suspended.

»NOTE

Breakpoints cannot be set on declaration statements because they are not executable statements.

Visual Studio lists the current breakpoints in a project in different ways. First, an icon appears in the left margin of the Code Editor to indicate that a breakpoint is set. In addition, a list of breakpoints appears in another window called the Breakpoints window. Figure A-5 shows the Code Editor and Breakpoints window with two breakpoints set.



```
    ' Demonstrate the use of a breakpoint.
    Private Sub btnSimpleBreakpoint_Click(ByVal sender As System.Object, ByVal e As System.Ev
        Dim Count As Integer
        For Count = 1 To 10
            txtLog.Text &= CrLf & "Count=" & Count.ToString

            Select Case Count
                Case 1, 3, 5, 7, 9
                    txtLog.Text &= CrLf & "Count is odd"
                Case 2, 4, 6, 8, 10
                    txtLog.Text &= CrLf & "Count is even"
            End Select
        Next
    End Sub
```

	Name	Condition	Hit Count
✓ ●	frmMain.vb, line 93 character 17	(no condition)	break always
☐ ○	frmMain.vb, line 96 character 21	(no condition)	break always

Figure A-5: Breakpoints appearing in the Code Editor and the Breakpoints window

1418835463_AppendixA.qxd 5/22/06 02:32 PM Page 890

»NOTE
It's possible to create as many breakpoints as you need. In addition, breakpoints are persistent from one invocation of Visual Studio to the next. That is, if breakpoints exist and you exit Visual Studio, those breakpoints will continue to exist the next time the application is loaded into Visual Studio.

Note the following about the breakpoints appearing in Figure A-5:

» A filled circle denotes an enabled breakpoint. Visual Studio suspends execution just before executing the statement containing the enabled breakpoint. If the breakpoint is set on a procedure declaration, execution is suspended just before executing the first statement in the procedure. Note that the breakpoints appear in both the Breakpoints window and the Code Editor.

» An outlined circle indicates that a breakpoint is disabled. Breakpoints are enabled and disabled by right-clicking the breakpoint in the Code Editor, and then clicking Disable Breakpoint or Enable Breakpoint from the pop-up menu. Visual Studio does not suspend execution on breakpoints that are disabled.

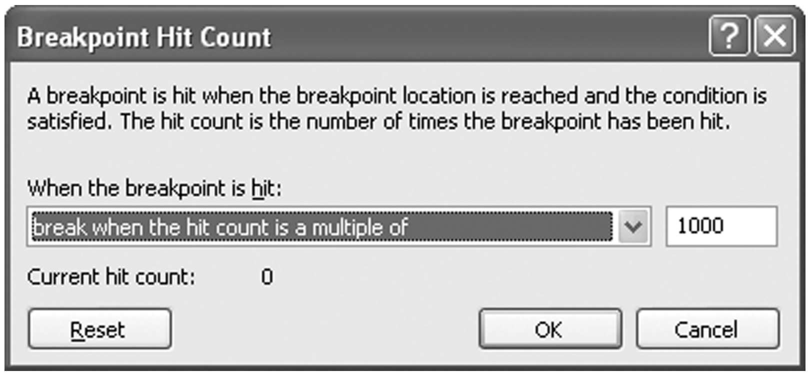
CREATING A BREAKPOINT

A breakpoint can be created in one of two ways. First, you can locate an executable statement in the Code Editor, and then click in the left margin. A filled circle appears in the margin indicating that a breakpoint is set and enabled. Using the second technique, right-click an executable statement in the Code Editor, and then click Breakpoint, Insert Breakpoint from the pop-up menu.

To remove a breakpoint, right-click the statement in the Code Editor containing the breakpoint, and then click Breakpoint, Delete Breakpoint from the pop-up menu. It is also possible to remove a breakpoint by selecting the breakpoint in the Breakpoints window, and then clicking the Delete button. Clicking Debug, Delete All Breakpoints removes all of the breakpoints in an application. A breakpoint can also be removed by clicking the filled circle in the Code Editor.

In addition to setting simple breakpoints that suspend execution just before a statement executes, it is possible to set hit count breakpoints and conditional breakpoints, as follows:

» A *hit count* breakpoint causes execution to be suspended after the statement containing the breakpoint has executed a certain number of times. For example, a hit count breakpoint is often useful with a `For` loop. After the statement(s) in the loop has executed some number of times, Visual Studio suspends execution. The dialog box shown in Figure A-6 is used to set a hit count breakpoint.



Breakpoint Hit Count

A breakpoint is hit when the breakpoint location is reached and the condition is satisfied. The hit count is the number of times the breakpoint has been hit.

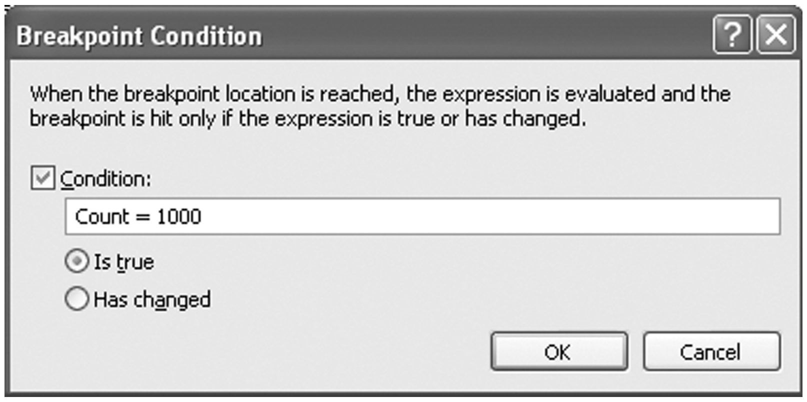
When the breakpoint is hit: break when the hit count is a multiple of 1000

Current hit count: 0

Figure A-6: Breakpoint Hit Count dialog box

As shown in Figure A-6, a list box is used to define how the hit count breakpoint will operate. In Figure A-6, execution will be suspended every 1000 times the statement executes.

» A *conditional breakpoint*, as its name implies, suspends execution when some condition is met. The condition in a conditional breakpoint has the same syntax as the condition in an `If` statement. The dialog box shown in Figure A-7 is used to set a conditional breakpoint.



Breakpoint Condition

When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is true or has changed.

[✓] Condition:

Count = 1000

(●) Is true

( ) Has changed

Figure A-7: Breakpoint Condition dialog box

As shown in Figure A-7, the Breakpoint Condition dialog box contains a text box in which a condition is entered. The Is true radio button, when selected, causes execution to be suspended when the condition is true. The Has changed radio button,

1418835463_AppendixA.qxd 5/22/06 02:32 PM Page 892

when selected, causes execution to be suspended when the value of the condition changes. Using the breakpoint shown in Figure A-7, execution will be suspended when the variable Count is equal to 1000.

In this exploration exercise, you will see how to set breakpoints and use breakpoints to trace the execution of program statements.

1. Activate the Code Editor for the form named **frmMain**. Locate the procedure named **btnSimpleBreakpoint_Click**. Set breakpoints on the following statements (shown in bold):

```
Case 1, 3, 5, 7, 9
    txtLog.Text &= CrLf & "Count is odd"
Case 2, 4, 6, 8, 10
    txtLog.Text &= CrLf & "Count is even"
```

2. Run the application. Click the **Breakpoints** tab. Click the **Simple Breakpoint** button. When the breakpoint is reached, the Code Editor appears and the statement that will be executed next is highlighted. While in break mode, highlight the variable named **Count** and the **Text** property of the text box named **txtLog**. As you do, the value of the variable and object property appears in a ToolTip.
3. Press **F5** to continue execution. Execution continues until the next breakpoint is hit.
4. Remove the two breakpoints and end the application.
5. Next, activate the Code Editor, and locate the **Click** event handler for the button named **btnHitCountBreakPoint**. Set a breakpoint on the following line:

```
Do Until Count > 10000000
```

6. In the Code Editor, right-click the breakpoint, and click **Hit Count** from the shortcut menu.
7. In the Breakpoint Hit Count dialog box that opens, select **Break when the hit count is a multiple of** from the drop-down list, and then enter **100** in the text box. Click **OK** to close the dialog box.
8. Run the application. Click the **Breakpoints** tab. Click the **Hit Count Breakpoint** button. The breakpoint is activated every 100 iterations. To see this, move the cursor over the variable Count in the Code Editor. Note that its value is 99. The value of the breakpoint will be 199, 299, and so on when the breakpoint is subsequently hit. Thus, the breakpoint is hit every 100 iterations.
9. Remove the breakpoint and end the application.

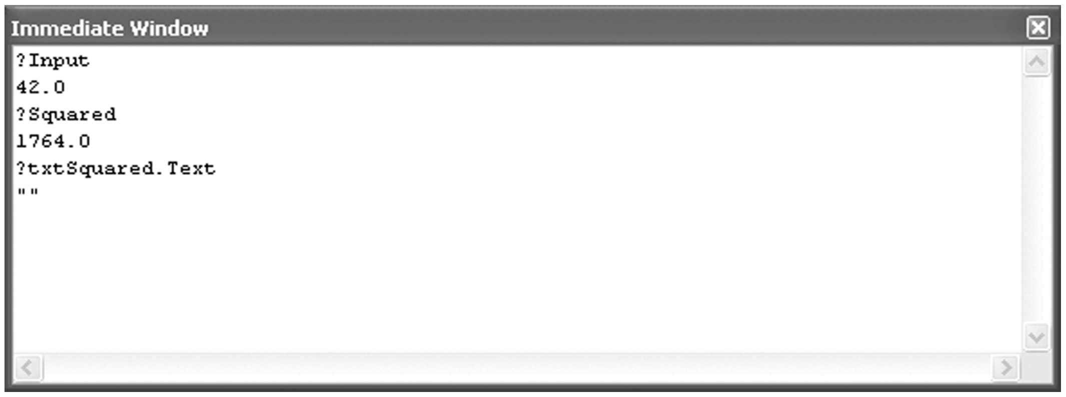
»NOTE

Hit count breakpoints are not available in the Visual Basic Express edition.

1418835463_AppendixA.qxd 5/22/06 02:32 PM Page 893

USING THE IMMEDIATE WINDOW

The Immediate window is used to examine the values of variables, change those values, and call procedures. Typing a question mark (?) followed by an expression causes Visual Studio to evaluate that expression and display the result on the line following the expression. The expression following the question mark can be a variable, object property, or any other expression that is valid on the right side of an assignment statement. Expressions containing variables or object properties are valid only in break mode because variables only have values while a Visual Studio program is running. It is also possible to set the value of a variable or object property using an assignment statement. Figure A-8 shows the Immediate window with various expressions.



```
?Input
42.0
?Squared
1764.0
?txtSquared.Text
""
```

Figure A-8: Immediate window

As shown in Figure A-8, expressions are entered in the Immediate window by entering a question mark (?) followed by the expression name. The expression is evaluated, and the result is then displayed.

In this exploration exercise, you will see how to use the Immediate window to examine the value of variables and object properties.

1. Activate the Code Editor, and locate the **Click** event handler for the button named **btnImmediateWindow**. Set a breakpoint on the following line:

   ```
   txtSquared.Text = Squared.ToString
   ```

2. Run the application, and click the **Breakpoints** tab. Enter the value **42** in the Enter a Number text box. Click the **Immediate Window** button. The breakpoint you specified in the previous step is hit and execution is suspended. Again, the statement is highlighted in the Code Editor.

3. View the Immediate window by clicking **Debug**, **Windows**, **Immediate**.
4. Enter the following statement in the Immediate window, and then press **Enter**: **`?Input`**

   The value of the variable 42.0 appears in the Immediate window. This is the value stored in the variable Input.
5. Enter the following statement in the Immediate window, and then press **Enter**: **`?Squared`**

   The value of the variable (1764.0) appears in the Immediate window.
6. Enter the following statement in the Immediate window, and then press **Enter**: **`?txtSquared.Text`**

   An empty string is displayed because a value has yet to be stored in this control instance.
7. Clear the breakpoint and end the application.

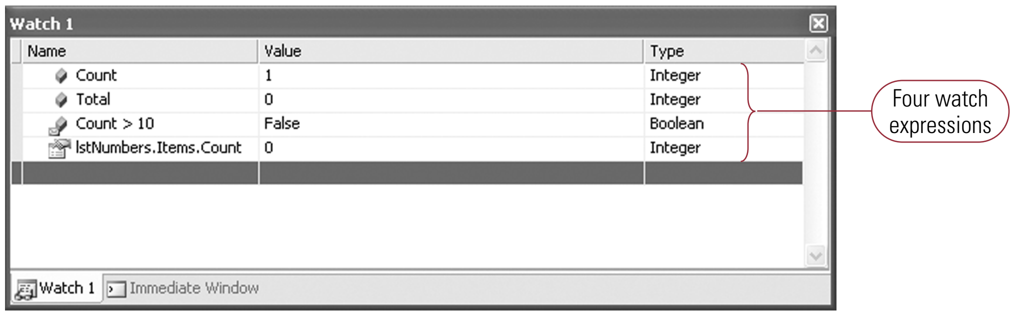
NOTE

Equivalent results can be obtained by highlighting a variable in the Code Editor or entering a comparable expression in the Immediate window. Both display the value of a variable. Which technique to use is a matter of personal preference.

SETTING WATCH EXPRESSIONS WITH THE WATCH WINDOWS

To examine the value of the same variable repeatedly, you can use watch expressions. Watch expressions provide a useful alternative to entering the same expression again and again in the Immediate window. Each time Visual Studio enters break mode, the values of watch expressions appear in one of four Watch windows. Watch expressions can be created, changed, or deleted while a project is in break mode. Like breakpoints, watch expressions are persistent from one invocation of Visual Studio to the next.

To add a watch expression to a project, one of four Watch windows is used. Each Watch window works exactly the same way. Visual Studio supplies four Watch windows so that you can organize the variables or expressions that you want to watch into functional or logical groupings. The contents of Watch windows can only be edited while Visual Studio is in break mode. Figure A-9 shows the first Watch window with four watch expressions.



Name	Value	Type
Count	1	Integer
Total	0	Integer
Count > 10	False	Boolean
lstNumbers.Items.Count	0	Integer

Figure A-9: Watch window

As shown in Figure A-9, you can watch a variable, or create more complex expressions such as `Count > 10`. Furthermore, it is possible to examine the properties of control instances such as a list box.

Each of the four Watch windows contains the following three columns:

» In the *Name* column, you enter the expression that you want Visual Studio to evaluate. To avoid typographical errors, you can copy the expression or variable from the Code Editor to a Watch window by means of the Windows Clipboard. The watch expression can consist of a variable, object, property, or more complex expressions.

» The *Value* column contains the current value of the watch expression.

» The *Type* column contains the data type of the watch expression.

A watch expression can be created, edited, or deleted only when Visual Studio is in break mode. To edit a watch expression, click on the expression to be edited in the desired Watch window, and then change the expression, as necessary.

In this exploration exercise, you will see how to work with a Watch window to examine expressions as an application executes.

1. Activate the Code Editor and select the **Click** event handler for the button named **btnWatch**, which is located in the Watch Windows region. Set a breakpoint on the following line:

```
lstNumbers.Items.Add(Count.ToString)
```

2. Run the application. Click the **Watch Windows** tab. Click the **Watch** button to execute the event handler for which you set a breakpoint in the previous step. Execution is suspended at the breakpoint you just created.
3. Click **Debug** on the menu bar, point to **Windows**, point to **Watch**, and then click **Watch 1** to display the first Watch window. Note that the Visual Basic Express edition has only one Watch window named Watch.
4. Type **Count** in the first row of the Name column.
5. Type **Total** in the second row of the Name column.
6. Type **Count** > **10** in the third row of the Name column.
7. Press **F11** several times. Each time the value of one of the watched variables changes, the value is updated in the Watch window. Clear the breakpoint and end the application.

»TIP

Use care when creating watch expressions. Visual Studio must check and evaluate each watch expression every time a statement executes. Thus, if you create too many watch expressions at once, program execution becomes very slow.

»NOTE

Although not a Visual Studio debugging tool, a message box can be used to display the value of a variable or object property. It's also possible to call the `Console.WriteLine` and `Debug.WriteLine` methods. Of course, when the application is known to be working correctly, this code must be removed.

THE CALL STACK WINDOW

The Call Stack window allows you to view which event handlers or other procedures have been called and the order in which those procedures were called. One use of the Call Stack window is to detect a phenomenon called cascading events. Cascading events and the Call Stack window are discussed in the following sections.

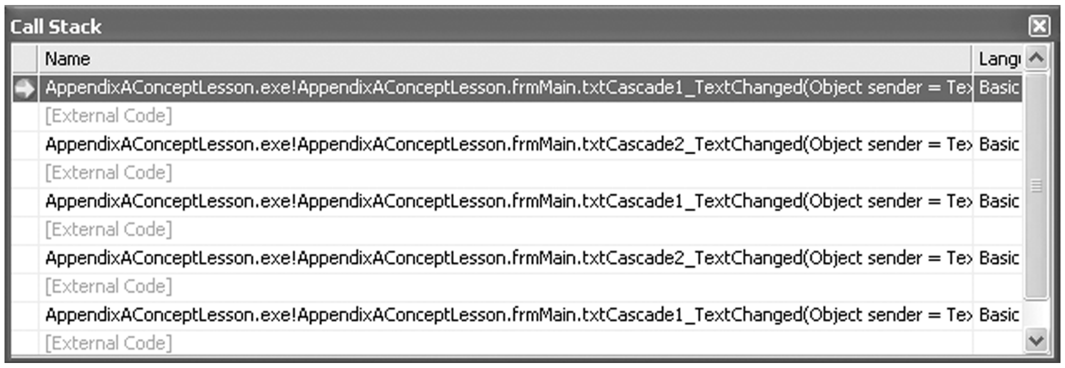
CASCADING EVENTS

In event-driven applications, improper logic can cause one event to raise another event indefinitely. Such a problem can arise when a statement fires a `TextChanged` event for one control instance. That control instance, in turn, fires a `TextChanged` event in a second control instance, and so on. If a control instance down the line fires a `TextChanged` event for the first control instance, the execution path becomes circular; that is, the events continue firing each other indefinitely. This phenomenon is called **cascading events**.

Consider a simple example involving two text boxes. After each text box gets input focus, it sets the focus to the other text box, and focus switches back and forth between the two text boxes indefinitely. The application seems to "lock up," and you cannot click any other control instance on the form while the text boxes continue updating one another. Whenever an application seems to lock up because of cascading events, it's possible to press the Break All button on the Debug toolbar, and then check the relationship between the events in the program.

VIEWING THE CALL STACK WINDOW

The Call Stack window lists both the procedures you create and those procedures that Visual Studio executes internally to create objects and handle events. Figure A-10 shows the Call Stack window demonstrating the cascading event problem discussed in the preceding section.



Name	Langı
AppendixAConceptLesson.exe!AppendixAConceptLesson.frmMain.txtCascade1_TextChanged(Object sender = Te›	Basic
[External Code]	
AppendixAConceptLesson.exe!AppendixAConceptLesson.frmMain.txtCascade2_TextChanged(Object sender = Te›	Basic
[External Code]	
AppendixAConceptLesson.exe!AppendixAConceptLesson.frmMain.txtCascade1_TextChanged(Object sender = Te›	Basic
[External Code]	
AppendixAConceptLesson.exe!AppendixAConceptLesson.frmMain.txtCascade2_TextChanged(Object sender = Te›	Basic
[External Code]	
AppendixAConceptLesson.exe!AppendixAConceptLesson.frmMain.txtCascade1_TextChanged(Object sender = Te›	Basic
[External Code]	

Figure A-10: Call Stack window

The Call Stack window displays procedures in the order in which they were called, that is, from the most recently called procedure to the least recently called procedure. The lines that appear dimmed contain procedures that Visual Studio executes internally.

1418835463_AppendixA.qxd 5/22/06 02:32 PM Page 897

These procedures set up event handlers and allow Visual Studio to call the event handlers and procedures that you create.

Figure A-10 shows that the event procedure `txtCascade1_TextChanged` was the procedure most recently called. The contents of this line say that the executable file named AppendixAConceptLesson.exe contains an executing event handler named `txtCascade1_TextChanged` in the form module named `frmMain`. Note that any arguments supplied to the event handler also appear in the Call Stack window. The Call Stack window can be helpful to locate cascading events as it lists each event handler that Visual Studio called.

In this exploration exercise, you will use the Call Stack window to examine cascading events.

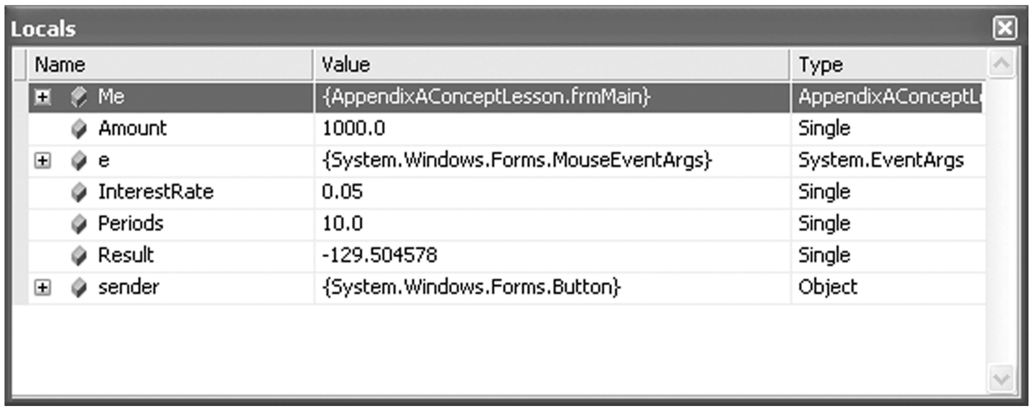
1. Run the solution and click the **Call Stack** tab. Enter the value **5** in the top text box.
2. Press **F11** to fire the cascading `TextChanged` events. The cascading events will continue to fire until an exception is thrown.
3. On the menu bar, click **Debug**, **Windows**, **Call Stack** to view the Call Stack window. The cascading events appear in the Call Stack window.
4. End the application.

In addition to locating cascading events, the Call Stack window can be used to examine the order in which Visual Studio called the various procedures and which procedures are executing.

THE LOCALS WINDOW

The Locals window displays the local variables pertaining to the currently executing event handler or any other executing procedure. As one procedure calls another procedure, Visual Studio updates the contents of the Locals window to display the information pertaining to the currently executing procedure.

The Locals window can be displayed only while Visual Studio is in break mode. In addition, the Locals window is useful only when an event handler or other procedure is executing. Although it is possible to display the Locals window while Visual Studio is waiting for an event handler to execute, the Locals window will display nothing. Figure A-11 shows the Locals window while the event handler named btnTypeMismatch is executing.



Name	Value	Type
Me	{AppendixAConceptLesson.frmMain}	AppendixAConceptL
Amount	1000.0	Single
e	{System.Windows.Forms.MouseEventArgs}	System.EventArgs
InterestRate	0.05	Single
Periods	10.0	Single
Result	-129.504578	Single
sender	{System.Windows.Forms.Button}	Object

Figure A-11: Locals window

As shown in Figure A-11, a reference to the form, its objects, and the form-level variables appears in the Locals window through a reference to the Me keyword. Sender and e provide the reference to the event handler's arguments. As these variables are objects supporting properties, they can be expanded and collapsed. Finally, the variables Amount, InterestRate, Periods, and Result are the local variables declared in the event handler. Note that if the current procedure is not an event handler, Me does not appear, and no reference to the form exists.

Note that the Locals window has a drill-down interface. Clicking the plus sign expands an object and clicking the minus sign collapses the object. The object supplied by the Me keyword is hierarchical. That is, Me (the form) might contain a text box, which, in turn, supports properties.

OTHER DEBUGGING WINDOWS

Visual Studio supplies several other debugging windows. The following list briefly describes the purpose of these windows:

» The Autos window displays the variables and objects in the current statement and the statements surrounding the currently executing statement.

» The Memory window displays blocks of memory in both text and hexadecimal format.

» The Processes window is useful to debug applications that communicate with each other.

- The Disassembly window allows you to view the intermediate language (IL) code for the program and is generated by the Visual Basic compiler.
- The Registers window allows you to view the CPU registers and their contents.

APPENDIX SUMMARY

This appendix presented an overview of the tools that can be used to debug applications. When and how to use these tools is up to the developer.

- To trace the execution of the statements in an application, use the Step Into, Step Over, and Step Out buttons.
- Breakpoints cause execution to be suspended when a particular statement is reached. Breakpoints can be set by clicking the left margin in the Code Editor. It is possible to set several breakpoints at the same time. Just before executing the statement containing the breakpoint, Visual Studio enters break mode, allowing you to examine the values of variables and properties. In addition to simple breakpoints, it is also possible to set hit count breakpoints and conditional breakpoints.
- The Immediate window is used to evaluate expressions while Visual Studio is in break mode. To evaluate an expression, enter a question mark, followed by the expression to evaluate.
- Four Watch windows allow you to examine the values of variables and expressions while Visual Studio is in break mode. Each Watch window works the same way.
- The Call Stack window is used to examine the order in which procedures are called. It can be useful to detect cascading events.

KEY TERMS

breakpoint—An executable statement where execution will be suspended.

cascading events—A phenomenon that occurs when one event causes another event to fire indefinitely.

debugging—The process of locating and fixing programming errors.

debugging windows—A collection of windows supplied by the Visual Studio IDE designed to help you correct errors in the applications you write.

logic error—A type of error that occurs when an application produces unexpected results.

1418835463_AppendixA.qxd 5/22/06 02:32 PM Page 900