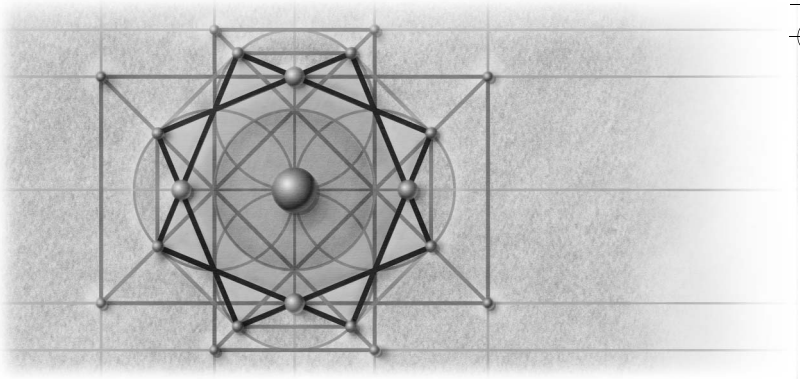

6

Common Tasks in Visual Basic .NET

In the previous chapter, we walked through how to upgrade an application. If you're new to Visual Basic .NET, some parts of the integrated development environment (IDE) may seem a little foreign. The purpose of this chapter is to familiarize you with some of the basics of working with applications in Visual Basic .NET. We'll start by building a simple Visual Basic .NET application from scratch. We'll use this application to introduce the new IDE and explain how to create new Visual Basic projects, and then we'll follow with a discussion of troubleshooting and debugging techniques and tactics. If you're already familiar with Visual Studio .NET, you might want to skip to the section on problem solving later in this chapter.

A Guide to Working in Visual Basic .NET

Microsoft Visual Studio .NET has a somewhat different development experience than Visual Basic 6 developers are used to. In fact, the new IDE combines features from Visual Basic 6, Microsoft Visual C++, and Microsoft Visual InterDev. The single most powerful aspect of the new IDE is that it is shared across all development languages. The Visual Basic editing experience is not all that different from the C# or C++ editing experience. The menus change slightly between project types, but the core interface, illustrated in Figure 6-1, remains the same.

In this section, you'll develop a simple Windows application to get a feel for the new IDE. The application will contain two buttons, a TextBox control and a ListBox control. The first button will add text from the TextBox control to the ListBox control. The second button will remove the selected item from the ListBox control. Nice and simple. Let's rock.

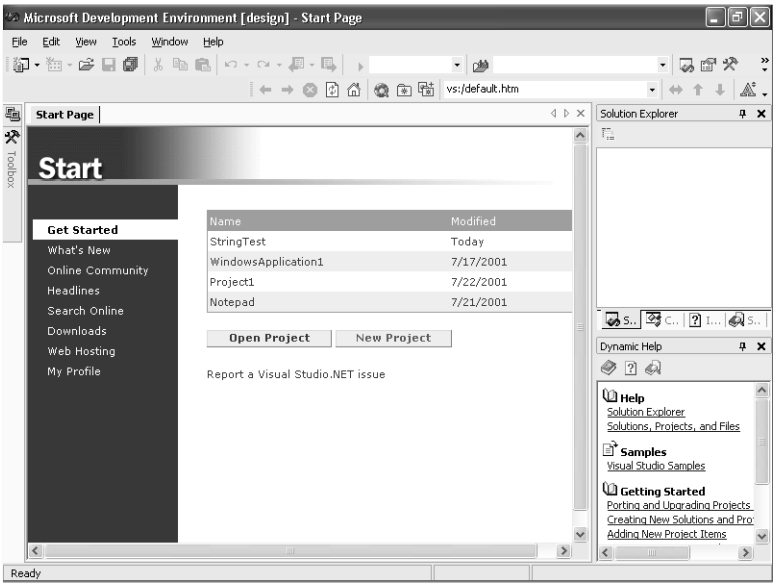

Figure 6-1 New Visual Basic .NET IDE.

Creating a Visual Basic .NET Project

Creating a Visual Basic .NET Windows Forms project is simple: under the File menu, select New, and then select Project. In the New Project dialog box, shown in Figure 6-2, select Visual Basic Projects as the project type and Windows Application as the project template. Leave all settings at their defaults and click OK to finish creating your project. Figure 6-3 shows the new project.

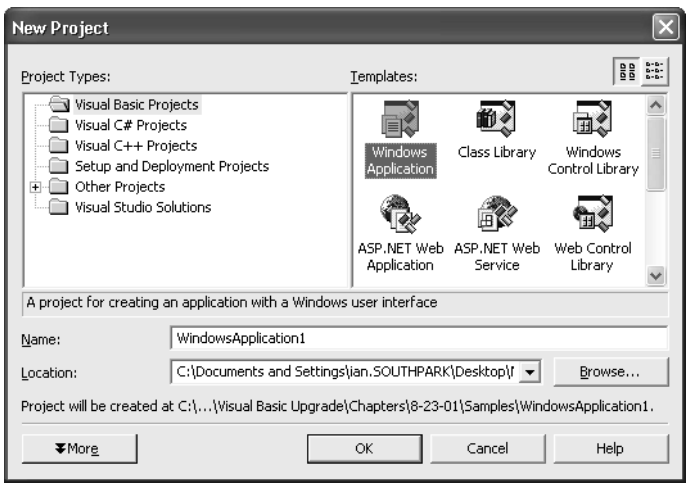

Figure 6-2 New Project dialog box.

Visual Basic .NET Project Types

Visual Basic .NET offers a sizable list of projects that you can create by using the New Project dialog box. The notion of a multilanguage "solution" is evident to anyone who examines this list. The dialog box does not change, whether you're creating a new project or adding a new project to an existing solution. Here are the Visual Basic .NET project types that are supported out of the box:

- Windows application
- Class library
- Windows control library
- ASP.NET Web application
- ASP.NET Web service
- Web control library
- Console application
- Windows service
- Empty project
- Empty Web project
- New project in existing folder

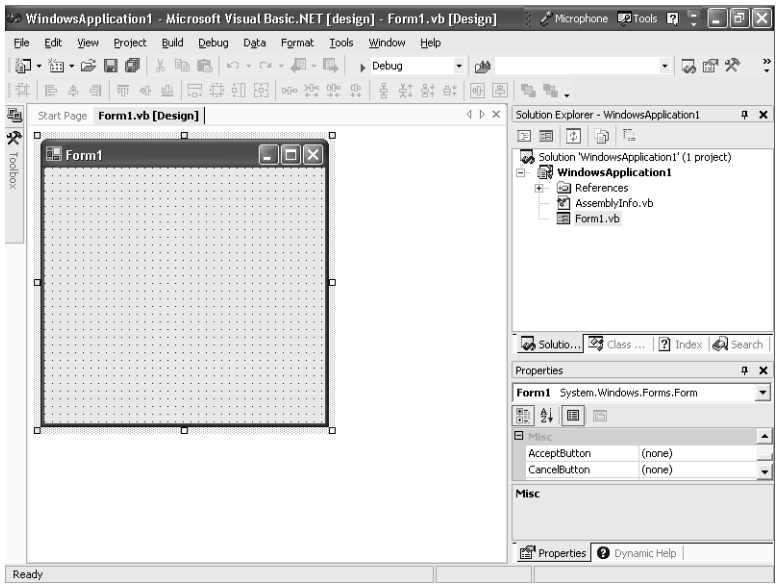

Figure 6-3 Your new project.

Getting to Know the Visual Studio .NET IDE

In Visual Basic .NET you still have projects, but instead of project groups (as you had in Visual Basic 6) you now have solutions. Each solution can contain multiple projects, even projects created in different programming languages (for example, Visual C# and C++). Visual Basic .NET now supports additional project types, including console applications, Web applications, Web services, and Windows services, all of which can be included in your solutions. You use the Solution Explorer to organize and manipulate your projects and build settings. By default, you will find the Solution Explorer in the upper right corner of the screen.

Figure 6-3 demonstrates several differences between Visual Basic .NET and Visual Basic 6. In the Solution Explorer you can see that references are now accessible from the project tree, and Visual Basic no longer uses file extensions to identify forms, classes, and code modules. All Visual Basic files now have the same extension, .vb.

On the left side of the IDE is the Toolbox. Moving your mouse over the Toolbox tab causes the drawer to slide out from the side of the screen. The Toolbox view defaults to the Windows Forms tab, which lists all the controls available to your application.

Now we are going to walk through creating a simple Windows application. To create a Windows Forms project, do the following:

1. Drag a TextBox control from the Toolbox and position it in the upper left corner of the form.
2. Drag a Button control from the Toolbox and position it to the right of the TextBox.
3. Drag another Button from the Toolbox and position it to the right of the first button.
4. Drag a ListBox control to the form and position it below the TextBox and buttons.
5. Position the controls so that the form looks roughly like Figure 6-4.

Now let's set some properties for the controls we just added to the form.

1. Click the Button1 control. The property page should be visible on the right. If it's not, right-click the control and select Properties from the shortcut menu.

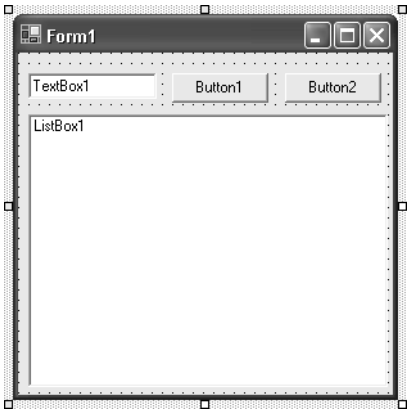

Figure 6-4 Newly added controls.

2. In the property page set the Text property to &Add and the (Name) property to AddButton.
3. Click the second button and set the Text property to &Remove and the Name property to RemoveButton.
4. Click the ListBox1 control and find the Anchor property. (It is in the group of properties that specify layout, toward the bottom of the properties list.) Click the down arrow and make sure the top, left, right, and bottom sides of the control are selected. Setting these options will ensure the correct resizing behavior of the control.

You now have a Microsoft .NET application that doesn't do much but look pretty and resize nicely. Let's add some functionality. Double-click the AddButton control. This takes you to the Code Editor in the AddButton_Click method. Add the following line of code:

```
ListBox1.Items.Add(TextBox1.Text)
```

From the left drop-down menu directly above the editor, select RemoveButton. In the drop-down menu on the right, select the *Click* event. This creates a *RemoveButton_Click* event and positions the cursor inside the event. Now you should add the following line of code:

```
ListBox1.Items.RemoveAt(ListBox1.SelectedIndex)
```

Not only have we implemented the desired functionality, but we've also explored two ways of associating event handlers with controls in Visual Basic .NET. Now that we have the features we want, let's run the application.

Running Your Project

Compiling in Visual Basic .NET is different from compiling in Visual Basic 6. In Visual Basic 6, you could run your application with coding errors. The IDE would break in and notify you of problems before it would execute problematic code. In Visual Basic .NET, your entire application has to be compiled before you can test anything at run time.

To run your application, press F5. This step will cause Visual Basic .NET to compile the application before launching it under the debugger. At this point we shouldn't have any errors. Everything should work just fine and look something like Figure 6-5 (after the user has added a few listbox entries).

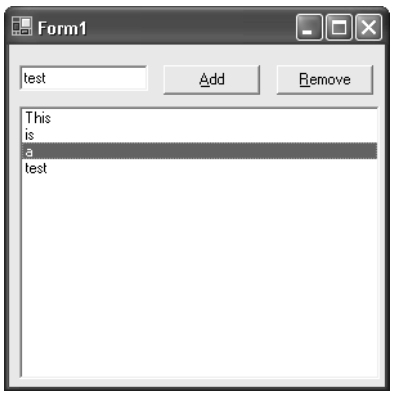

Figure 6-5 Your new Visual Basic .NET application.

It turns out that everything is not working just fine. Try selecting an item in the listbox (add one if necessary), and then click the Remove button twice. You will get the error shown in Figure 6-6.

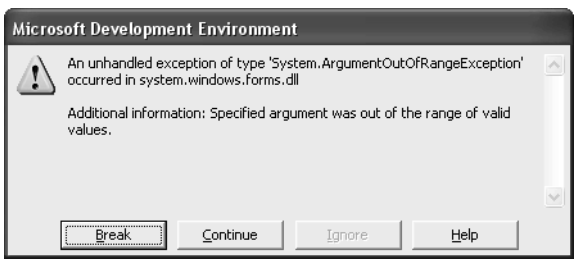

Figure 6-6 The application is broken.

So it's broken. Click Break and let's dive into the debugger.

A Quick Introduction to Debugging

Debugging in Visual Basic .NET is not exactly what traditional Visual Basic developers are used to. Probably the most surprising missing feature is Edit And Continue. However, Visual Basic .NET offers a host of new features that greatly enhance the debugging experience, including service debugging, cross language debugging, XCopy deployment, structured exception handling, and a more sophisticated debugging API.

Where Is Edit And Continue?

The developers of the initial version of Visual Basic .NET decided not to include Edit And Continue. Dropping a feature that Visual Basic developers have come to rely on was not done lightly, but it had to be done. Visual Basic .NET and the .NET Framework are totally new platforms developed from the ground up. The challenge was to provide all the features of previous platforms that had come about during a decade of refinements. Unfortunately, including Edit And Continue would have meant delaying the initial release of Visual Basic .NET. Although it is a sadly missed feature, don't let this omission detract from the other powerful debugging features available in Visual Basic .NET.

In the preceding section, we learned that the application is throwing an exception on the following line of code, and we need to find out why. The exception dialog box has already told us that the error was an ArgumentOutOfRange exception. Looking at the code, we can see that the problem is located in the *SelectedIndex* property of the ListBox1 control:

```
ListBox1.Items.RemoveAt(ListBox1.SelectedIndex)
```

The Autos window in the lower left side of the IDE displays the value for *SelectedIndex*. When no item is selected, the *SelectedIndex* property is –1. You can also use the Command window to inspect the *SelectedIndex* property. Just type **?ListBox1.SelectedIndex** to print out the value in the Command window.

One possible solution is to add code that will check for this condition before trying to remove an item from the ListBox control. You must end the debugging session before you can make any changes to the source file. The current version of Visual Studio .NET locks the source files during debug sessions. Moving to the Code Editor, replace the body of the *RemoveButton_Click* event with the following code:

```
If ListBox1.SelectedIndex <> -1 Then
   ListBox1.Items.RemoveAt(ListBox1.SelectedIndex)
End If
```

This code prevents the application from attempting to remove any items from the ListBox unless an item is selected. It will avoid the exception previously encountered at this point. There is, however, another way to do this—possibly a more proper way from a user-interface perspective. It makes sense that the user should not be able to click the Remove button unless it will actually perform a valid action. To specify this behavior, do the following:

1. In the Form Designer, select the Remove button and change its Enabled property to False.
2. In the Code Editor, select the ListBox1 control from the left drop-down menu. Select the *SelectedIndexChanged* event from the right drop-down menu. Add the following code to the new event handler:

```
If ListBox1.SelectedIndex = -1 Then
   RemoveButton.Enabled = False
Else
   RemoveButton.Enabled = True
End If
```

The end result is that by default, the Remove button is disabled at startup. As soon as an item is selected in the listbox, the Remove button is enabled. If the user clicks the button, the selected item is removed, and the button is again disabled.

Miscellaneous Items

The Visual Basic .NET IDE has a whole host of new features. It also introduces some new behaviors that will take time for Visual Basic developers to get used to. This section walks you through some of those behaviors.

Handling Build Errors

If your project contains compile errors, when you build it Visual Basic .NET shows the dialog box seen in Figure 6-7.

Figure 6-7 Your reward for trying to run a project containing errors.

The only purpose of this dialog box is to let you choose between running a stale build of your solution and fixing the problems and trying again. We can't really see why you'd want to make changes and then run a previous build. The vast majority of the time you are going to try to fix the problems first. And that's why you have the Task List.

Using the Task List

The Task List has several different uses. First and foremost, it reports compilation errors. Unlike compiling in Visual Basic 6, you can now view a list of all the errors and fix them in whichever order you prefer, instead of having the order dictated by the compiler, and you can do so without having an annoying dialog box pop up for every error that the compiler comes across. Figure 6-8 displays the Task List for a sabotaged sample application. Double-clicking any of these items brings you to the related line of code. When the issue is fixed, the item in the Task List automatically goes away.

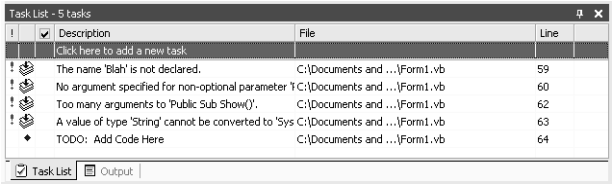

Figure 6-8 Task List.

Of course, the Task List has many other purposes. In the Options dialog box, displayed by choosing Options from the Tools menu, you can select Environment and then select Task List. This dialog box, shown in Figure 6-9, allows you to customize what is displayed in the Task List and gives you the option of adding custom tokens. This feature can be a useful way of marking parts

of your application and prioritizing feature areas. For example, you can create tokens that indicate the phase in which features will be added. You could create tokens like Beta1, Beta2, and Beta3 and filter based on the phase of the project you are currently in. You could also create tokens like Bill, Bob, or Janice that indicate areas that individuals on your team need to address. This approach is often useful when implementing large applications.

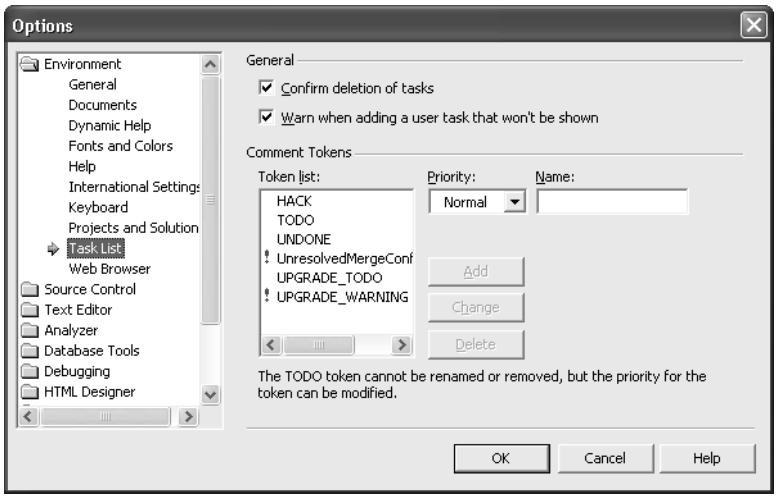

Figure 6-9 Visual Studio .NET Options dialog box.

These comments would look like this in your code:

```
'TODO: Do this
'Beta2: This feature needs to be implemented later
'Janice: Can you deal with this?
```

Using Breakpoints

Breakpoints work the same way they did in Visual Basic 6. Find a line of code you want to break into the debugger on, and either click to the far left of the line in the editing window or right-click the line and select Insert Breakpoint from the shortcut menu. You can remove breakpoints by clicking the red circle or right-clicking the line of code and selecting Remove Breakpoint from the shortcut menu.

References

Visual Basic .NET has two types of references that you can add to your project: a standard reference and a Web reference. A standard reference is used to import the namespace of either a COM type library or a .NET reference. Web references are used exclusively for importing a Web service.

Standard References

As Figure 6-10 shows, three kinds of standard references are available to you through the Add Reference dialog box: COM references, .NET references, and project references. Use .NET references and project references for referencing managed components in your project. Adding a COM reference causes the COM type library to generate a managed type library, thereby enabling the use of COM objects within your Visual Basic .NET project as if they were managed classes.

To open the Add Reference dialog box, right-click the References node in the Solution Explorer, and choose the Add Reference menu item. This opens the Add Reference dialog box as seen in Figure 6-10.

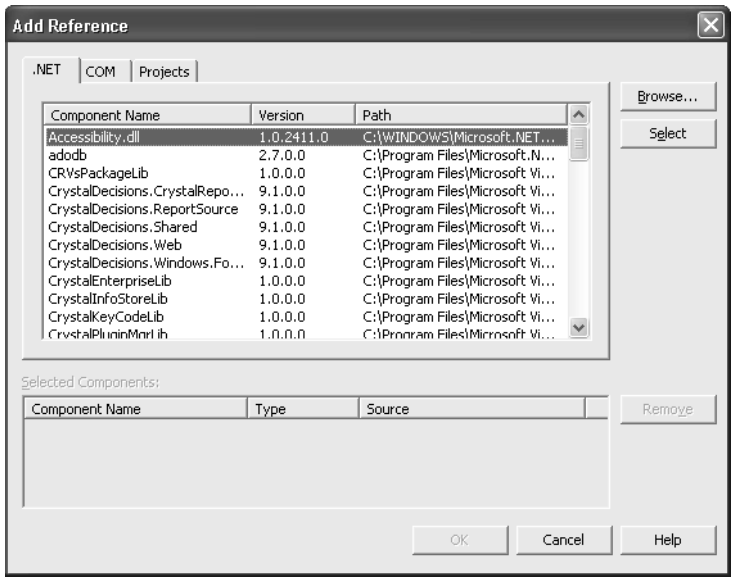

Figure 6-10 Add Reference dialog box.

To remove a reference, open the References node in the Solution Explorer, right-click the reference you want to remove, and choose the Remove menu item.

Web References

To enable your application to consume Web services, you add a Web reference. Adding a Web reference generates a local proxy class that enables you to call methods on the Web service. You add Web references by right-clicking the Reference node in the Solution Explorer and choosing the Add Web Reference menu item. This opens the Add Web Reference dialog box, as seen in Figure 6-11.

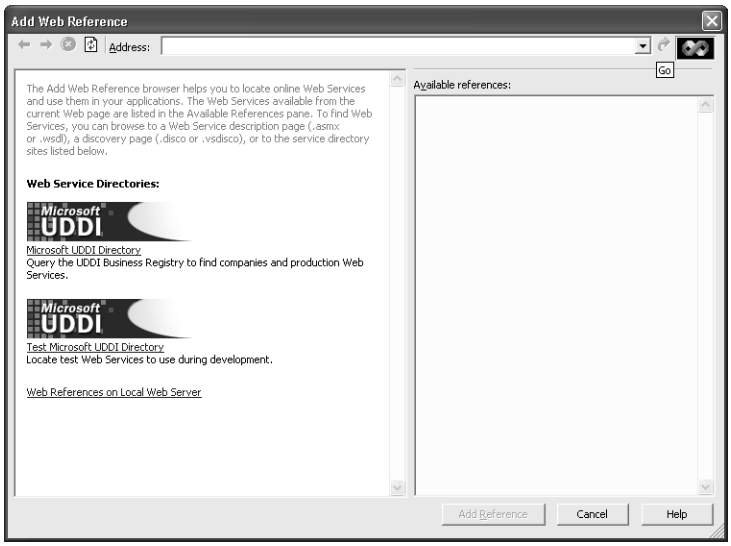

Figure 6-11 Add Web Reference dialog box.

Problem-Solving Techniques

You need to master a number of skills to become effective at debugging. This section explores some of those skills.

Using the System.Diagnostics Library

The System.Diagnostics namespace has three important classes: *Debug*, *Trace*, and *Debugger*. The *Debug* and *Trace* classes are related in that you can use both of them to generate debugging information at run time, which is helpful in diagnosing various application problems. Although their functionality is slightly different, both classes support similar interfaces, making it easy to move back and forth between them. However, you might also be confused about when to use one instead of another. The official word from the Microsoft Developer Network

(MSDN) documentation is that the *Trace* class can be used to instrument release builds. Instrumentation allows you to monitor the health of your application running in real-life settings, isolate problems, and fix them without disturbing a running system. You can use the *Debug* class to print debugging information and check your logic with assertions; you can make your code more robust without affecting the performance and code size of your shipping product.

The *Debugger* class is somewhat different. It enables communication between your application and an attached debugger. This can be useful for inserting code into your application that assists in debugging complex problems or error scenarios. In addition, some application types (such as Windows services) can be notoriously difficult when you're trying to trace the root of failures. You might find these two particular methods of the *Debugger* object useful:

- The *Log* method enables the application to post messages to the attached debugger.
- The *Break* method enables the application to signal the debugger to break into that line of code. Using this method is analogous to using conditional breakpoints in Visual Basic .NET, except that *Break* will work with any attached debugger.

Using CorDbg

CorDbg is a managed command-line debugger. It can be extremely useful for handling problems with deployment machines that don't have the Visual Basic .NET debugger installed. We don't recommend this debugger to the casual developer, but the simple commands explained in this section will provide a useful introduction to command-line debugging.

Where Do I Get CorDbg?

CorDbg is located in the %SYSVOL%\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin directory when the Framework Software Development Kit (SDK) is installed. You can use the Visual Basic .NET installer to install the SDK, or you can download the SDK from the MSDN Web site.

First you need to start the debugger. If you add the path to the directory containing CorDbg.exe to your environment, you can start the debugger by typing **cordbg** at the command prompt. The following sequence of commands is typical:

1. Type **pro** to get a list of managed processes and process IDs.
2. Type **at [pid]** to attach to the process.
3. Type **ca e** to catch all exceptions.
4. Type **g** to continue the program execution.

When an exception occurs, CorDbg breaks to the prompt and reports the exception type. At this stage you can evaluate where the exception occurred, get stack trace information, and inspect variable values.

If command-line debuggers aren't your style, the Framework SDK also provides a GUI debugger (DbgCLR, in the %SYSVOL%\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\GuiDebug directory) that provides a similar debugging experience to the Visual Basic .NET debugger.

Simplifying Complex Expressions

Trying to figure out the cause of an exception in a large compound statement can be a real challenge. When you encounter obtuse errors (either compile or run-time errors), it can be helpful to break down the offending line of code into smaller, isolated statements. There's no penalty for increasing the line count of your application, so why worry? Breaking down complex single-line expressions into several separate expressions can also improve your ability to add run-time error handling to your application.

Let's look at an example of simplifying complex expressions. The following Visual Basic .NET line is a complex expression because there are several statements on one line:

```
MsgBox(Format(CDate(myString & "/2002")))
```

If a line like this causes errors and you can't figure out where the error occurs, try simplifying it by putting each statement on a separate line and storing the results of each statement in a temporary variable. The following sample shows how to do this:

```
Dim tempString, formattedString As String, tempDate As Date
tempString = myString & "/2002"
tempDate = CDate(tempString)
formattedString = Format(tempDate, "d-mmm-yyyy")
MsgBox(formattedString)
```

If an error occurs after you've made the change, it's simple to determine which statement actually caused the error. We are not suggesting you do this with all your code, but it's a useful technique for tracking down the cause of an error when all you know is that it happens somewhere on a particular line.

Conclusion

This chapter covered a lot of information, from the bare debugging essentials to more sophisticated troubleshooting methods. We hope this chapter has provided you with a basic understanding of how to work with the IDE and how to approach debugging applications in Visual Basic .NET. The Visual Basic .NET debugging experience is significantly different from what you're familiar with in Visual Basic 6. This helps emphasize the importance of using the techniques discussed in this chapter to ensure that you can work effectively with your applications.