2

Using Visual Studio .NET: IntelliSense and Debugging

Since you're going to be stuck using Visual Studio .NET anyway, at least for this edition of the .NET Compact Framework, you might as well get familiar with some of the features it provides. A lot of the things in Visual Studio .NET are targeted at database development, Web development, and other projects very different from .NET Compact Framework programming, but there are at least two features beyond the basic code entry and compilation which are well worth learning: IntelliSense, which helps you type correct code, and the debugging features, which can help you sort out what's going on when your program logic doesn't behave as expected.

> If you've used Visual Studio .NET before, this chapter may not be very exciting to you. Feel free to skip around.

IntelliSense

IntelliSense is a feature Microsoft provides to help you navigate through the ever-growing number of methods and parameters in their libraries and the code you write yourself. Thanks to IntelliSense, whenever you type an object or variable name it recognizes and type a period, you'll get a menu of choices that Visual Studio .NET thinks might be appropriate there. Type the first few letters, and IntelliSense will show you the possibilities corresponding to what you typed. Hit the space bar (or any other key that isn't allowed in a variable, property, or method name, and it will fill in the rest for you. You can also summon IntelliSense with keyboard combinations, or use the Edit > IntelliSense menu to request particular assistance.

> The IntelliSense menu only shows when the cursor is in the code window, so if you can't find it, that's why!

Those options are worth considering, so here's a list:

Licensed under Creative Commons Attribution-NonCommercial-NoDerivs 2.0 license.
http://creativecommons.org/licenses/by-nc-nd/2.0/

List Members (Ctrl+J)

The List Members option is what you get in the traditional "type a period and see the list" option. You can summon the list any time, though, and it will show you everything in scope at the cursor position. This means that you can type part of a variable name and find it in the list.

Parameter Information (Ctrl+Shift+Space)

Parameter information is automatically displayed when you type the open parenthesis of a method call, but you may want to see the full list of parameters when you've clicked into an existing method call. This will show you the list, provided that you're in a method call. (If you're not, it does nothing.)

Quick Info (Ctrl+K, then Ctrl+I)

If there's documentation available for the identifier the cursor is in when you choose this, Visual Studio .NET will show it as a tooltop. At minimum, you'll at least get its declaration.

Complete Word (Alt+Right Arrow, or Ctrl+Space)

If you've typed enough of a word for IntelliSense to guess how to finish it, this will complete the word for you. If you haven't, you'll get a list like that provided by List Members, which will be set at the closest point IntelliSense can guess.

All of this sounds pretty good, and is generally helpful, but there's one catch: Visual Studio .NET's IntelliSense isn't always perfect when it comes to the .NET Compact Framework. It's not precisely a problem with IntelliSense, but rather a design issue in the .NET Compact Framework: some methods are implemented by various controls, but are implemented so that they do nothing. IntelliSense can tell that the method is there, but not that it's empty. If you find yourself facing mysterious behavior when IntelliSense insists a method exists, this is the likely problem.

Debugging

The other major feature which Visual Studio .NET provides is a debugging environment that will let you set breakpoints and inspect values in running code. You can perform debugging on both the emulator and actual devices. We'll start with the emulator, and then discuss how this works (pretty much the same, just slower) on actual devices. Emulator testing is fine for basic applications, but if you need to do anything with I/O, you'll definitely want to test on an actual device.

To get started, we'll return to the very simple application from the previous chapter which asked the user to enter some information into a text field and then changed values or showed a message box depending on what button they pushed. A simple breakpoint is the right place to start. You can add a breakpoint to any line in your program by putting the cursor in it and pressing F9. To remove it, just press F9 again - it toggles the breakpoint on and off. Figure 2-1 shows the code window for the MySecondFormVB project created in the previous chapter, with a breakpoint added on the line which displays a message box.

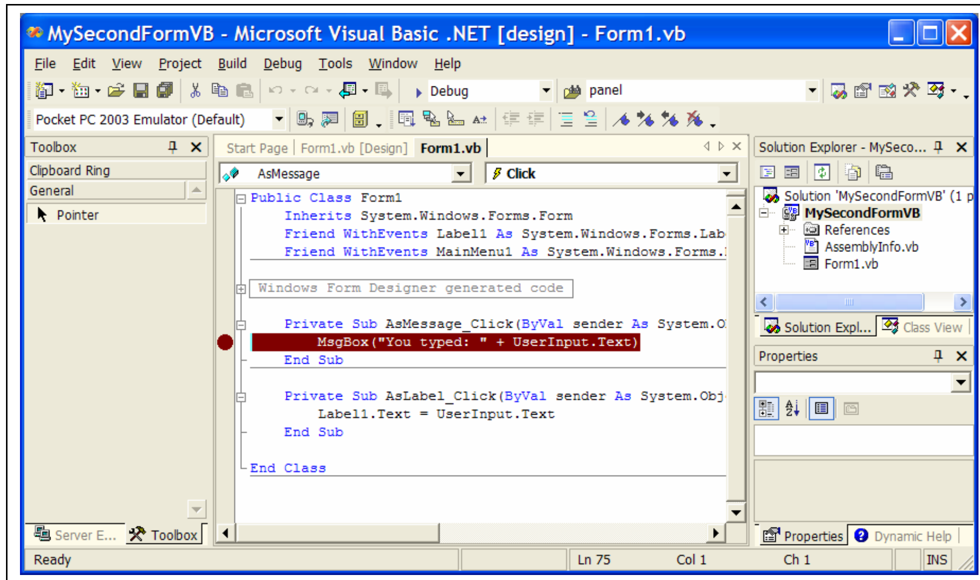

Figure 2-1. Code window, with a breakpoint added.

If you then run the program, using the F5 key (which starts it with debugging), everything will run normally until you click the "Display As Message" button. When you do that, the Visual Studio .NET window will reappear, looking like Figure 2-2.

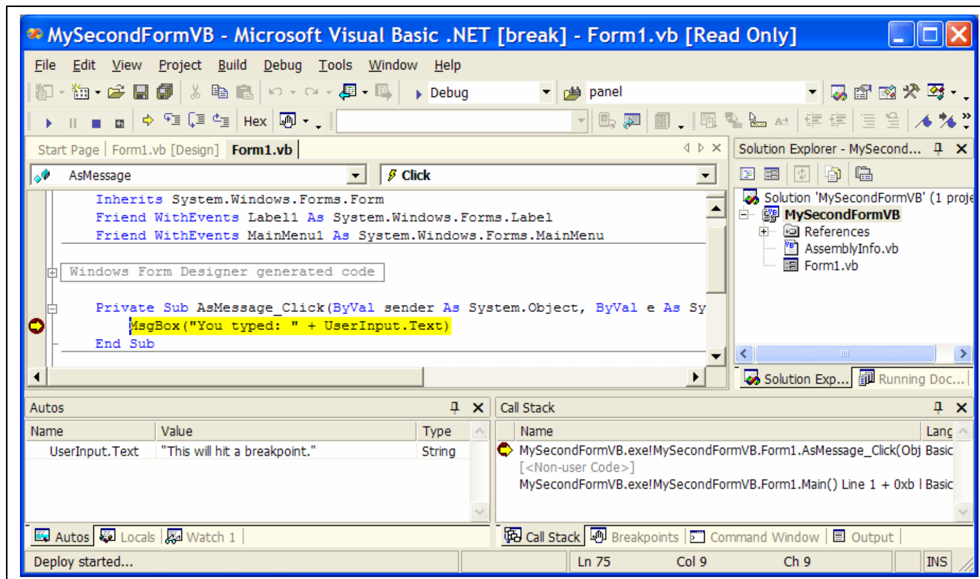

Figure 2-2. Visual Studio .NET responding to a breakpoint.

Visual Studio .NET both highlights the code where the breakpoint occurred and shows some key information at the bottom of the screen. The Autos area lists variables in use in the current call (and its neighbors), their value, and their type. Since the MsgBox routine is using UserInput.Text as part of its text, that name, value, and type all appear. If you click on the Locals tab, you'll see a similar window listing variables currently in scope. You can change the values of these variables as well as watch them go by. If you change

a variable, its new value will appear in red. For fun, I changed the value to "This will hit breakpoint."

On the lower right, you'll also see the call stack - which methods or functions were called that brought us here. In this case, we can see that AsMessage_Click was called, then there were a few layers of non-user code, and then the whole thing was started by the Main routine. (As was discussed in the previous chapter, there is no explicit Main routine in Visual Basic.NET, just a property setting that identifies the base form.)

If you want to explore the non-user code, right-click on the call stack and choose "Show Non-User Code".

You can also click on the Breakpoints tab to see currently set breakpoints, or the Command Window tab to be a super-expert and type in Visual Studio .NET commands directly.

Now that the program is stopped, you can control its execution line by line. F11 is "Step Into", which executes the current line and shifts the breakpoint to any functions or subroutines being called so you can watch their internal processing. If you're not interested in those details, F10, "Step Over", executes the current line and goes to the next, letting whatever function calls happen without investigating their contents. If you step into a function and decide that you're not interested in its internal details, you can use Shift+F11 to "Step Out" of the current function. All of these are also available from the Debug menu.

For starters, we'll hit F10, which will let the MsgBox command do its thing with the modified variable. As Figure 2-3 shows, the UserInput.Text property changed, and we have a message box showing a slightly broken message.

Figure 2-3. The emulator proceeding, with the contents of the message modified from Visual Studio .NET.

When you click the ok button on the message box to make it go away, you'll be returned to the debugger, which will now be highlighting the End Sub line, as shown in Figure 2-4. (The MsgBox line will still be maroon.)

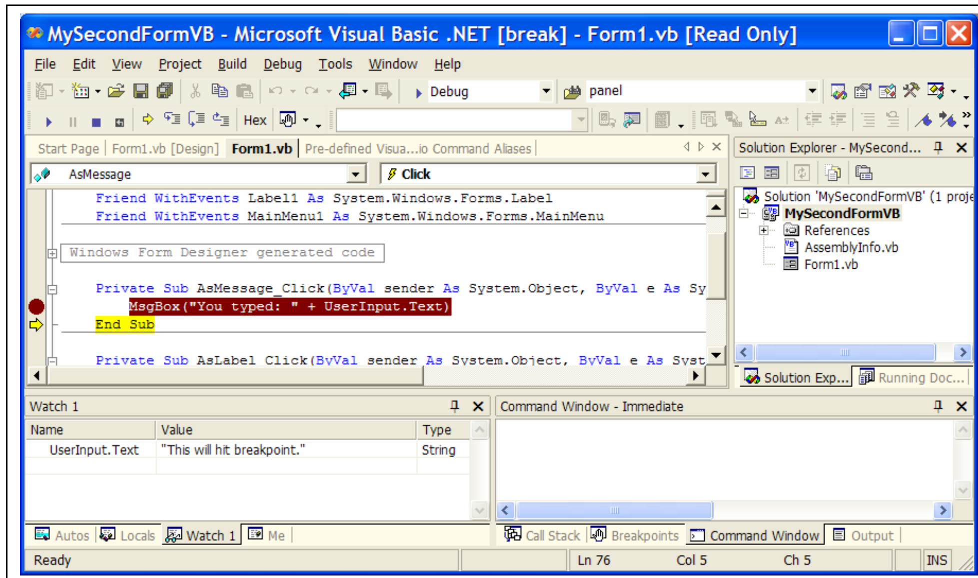

Figure 2-4. Stepping to the next line in the debugger.

The Pocket PC emulator will still be displaying the message box until we get out of this subroutine and the form can redraw itself. To let it do that, you can either single-step again, or you can press F5. Once you're debugging, F5 becomes Continue rather than Start. If you want to halt the program, you can press Shift-F5 instead and go back to editing the code to fix whatever dire problems you've discovered.

Once you've continued, the program will go on running until it hits another breakpoint. In this case, there's only one breakpoint, set explicitly at that line. You can solve a lot of difficult programming problems using just the techniques shown above, but there's one more feature you should know about: setting breakpoints based on rules.

This chapter is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 2.0 license. (http://creativecommons.org/licenses/by-nc-nd/2.0/)