712ch15 9/25/01 10:05 AM Page 393

CHAPTER 15

The .NET Compact Framework

Bringing the Power of .NET to Handhelds

WHILE FIRST RESEARCHING THE series of seminars upon which this book is based, we fell in love with .NET. For sheer ease of use and limitlessness of potential applications, we feel that there is no other programming platform in the world today that even comes close to equaling .NET. As we have become even more familiar with .NET, we have been finding programming with other technologies increasingly frustrating.

"Why can't Microsoft just port .NET to small devices," we wondered, "so we could use all of the same, great .NET tools to develop for the Pocket PC and Palm OS?" Then yesterday, we received a CD in the mail labeled "Microsoft .NET Compact Framework." Ever get the feeling that the folks up in Redmond developed some kind of secret, mind-reading technology without telling us?

Introduction

So what exactly was on the CD, and how can you use it to create Mobile .NET applications?

Seeing the Big Mobile .NET Picture

In order to fully understand what the .NET Compact Frameworks are, you must first understand why they exist, and specifically what void they fill within Microsoft's overall Mobile .NET strategy.

Reviewing Your Progress

So far in this book, we have examined two kinds of Mobile .NET technologies: Mobile Web Services and Mobile Web Applications.

Mobile Web Services

You may recall from Chapter 10 that Web Services provide you with a platform-independent way to access code libraries running on remote computers. Individual methods within these libraries may be exposed for remote invocation via industry-standard XML messages. These messages, and the responses that they generate, are typically structured according to the SOAP protocol for XML Remote Procedure Calls (RPCs).

In Chapter 11, we showed you how to use .NET Web Services from virtually any device. Table 15-1 below summarizes the devices we discussed and their level of support for .NET Web Services.

Table 15-1. Devices and .NET Web Services

DEVICE NAME	WSDL?	SOAP?	XML?	HTTP?	MARKUP?
Internet Explorer	Yes	No	Yes	Yes	Yes
Pocket PC	No	Yes	Yes	Yes	Yes
J2ME	No	No	Yes	Yes	No
Palm OS	No	No	No	Yes	Yes
WAP/i-Mode	No	No	No	No	Yes

Table 15-1 is organized from easiest to hardest in terms of getting each device to work with .NET Web Services.

The row to which particular attention should be paid is the one for Pocket PC. Were it not for Simon Fell's excellent PocketSOAP libraries, our answer for SOAP support on this device would be "No." It would then be easier to invoke a .NET Web Service from a Java Micro Edition device than from a Pocket PC. How would you feel about *that* if you were Microsoft? (Hint: this is one of the shortcomings that the .NET Compact Frameworks is meant to address.)

Mobile Web Applications

In Chapters 12 through 14, we examined Microsoft's brilliant Mobile Internet Toolkit technology. As you might have guessed from the previous sentence, this is a particular favorite of ours. Furthermore, it should be noted that there is very

little that one might want to do from a mobile device that could not be done by using the Mobile Internet Toolkit.

Still, like all markup-centric technologies, there are certain resource limitations that can have a limiting effect on their usefulness. One of the most obvious of these limited resources is bandwidth. It can be prohibitively expensive to have to send data back to the server just to respond to every minor UI event.

NOTE *Moreover, consider the case of the technologist in a remote part of the world. Wireless Internet service is not available everywhere on this planet at the time of writing, so it is entirely possible to travel to areas where server-based applications using the Mobile Internet Toolkit (or any other server-based technology, for that matter) would be completely inaccessible. In these situations, the only .NET option left available would be the Mobile Device Applications you can create using the .NET Compact Frameworks.*

Another potentially even more limited resource is the number of developer hours that can be devoted to producing new device adapter code. As mentioned in the previous chapter, we would like to assist in this respect by leading some open-source device adapter initiatives at `http://www.mobiledotnet.com`. It would seem likely, however, that the evolution of devices will always exceed the pace at which adapters for them can be developed.

Mobile Device Applications

There is a third kind of Mobile .NET application that we have not yet discussed. The reason is that until the arrival of the .NET Compact Frameworks, they didn't really exist as anything other than an idea on paper, the idea being that developers should be able to write .NET code for execution on devices as well as on servers.

There are a couple of wonderful possibilities that become entirely feasible once you can make this happen.

Leveraging Existing Skills

The first great thing that happens when you can execute .NET code on devices is that everyone who has previously been a .NET developer suddenly becomes a device developer.

712ch15 9/25/01 10:05 AM Page 396

At first this might appear to be a bit of an oversimplification and/or an exaggeration. After all, devices still have peculiar limitations that need to be learned and worked around. Furthermore, the tools that support .NET on devices might require an additional degree of learning effort that not every .NET developer would feel inclined to invest.

Unfortunately for those who would like to see device development remain a black and mysterious art, Microsoft has addressed both of these issues with .NETcf. In the case of device limitations, the very nature of .NET's managed execution environment allows developers to largely (though not entirely) ignore such tricky issues as memory allocation and reclamation. After all, sorting out details like this is why we have a CLR in the first place!

Microsoft has also integrated the .NET Compact Frameworks completely and seamlessly into the existing Visual Studio architecture. This has ensured that little if any additional learning is required in order to begin immediately creating .NET applications for devices. Heck, we had our first .NETcf application running on our Pocket PC within an hour of taking the CD out of the mailbox—and we didn't even read any of the instructions until something broke!

Supporting Multiple Platforms

The part of Microsoft's .NET Compact Frameworks initiative that you may really find surprising is its stated intention to be platform independent. Specifically, Microsoft is attempting to attract OEM licensees for this technology who manufacture non–Windows CE devices. These licensees would be encouraged to provide a special subset of the .NETcf class libraries as a part of the internal flash ROMs of their devices.

For more information on this, see the section later in this chapter entitled "The Cross-Platform Profile."

Getting Started

To get started creating applications for devices with .NET, you need only do two things: install .NETcf and then take a quick tour of the new features it has added to Visual Studio .NET.

Installing .NETcf

We found installing .NETcf to be extremely easy. It worked for us the first time, right out of the box. We just put the CD in the computer and ran the setup program.

712ch15 9/25/01 10:05 AM Page 397

NOTE *The CD we received in the mail was a special Alpha-software version of .NETcf. By the time this is published, the technology will be in early Beta. By the time you read this, it may even be in its final release.*

The current plan is for the .NETcf bits to remain a freely downloadable add-on to Microsoft's Visual Studio .NET product. This means that anyone can get their hands on the bits without paying Microsoft one additional penny. On the other hand, you will have had to have already purchased Visual Studio .NET in order to derive any benefit from these bits whatsoever.

Learning the Interface

Once the installer has finished, you should be able to launch Visual Studio .NET and take a look at the features that have been added. If you are particularly paranoid, you may reboot first—though we didn't find it to be necessary.

Choosing Your Project Type

Immediately after starting Visual Studio .NET, you should choose to create a new project. Immediately, you will notice that options have been added to the New Project dialog box.

If you elect to create a Visual Basic project type at this point, you will see that you are able to create projects for devices now in addition to the desktop. Choose to create a Windows application for the Pocket PC platform. We will explain the significance of this choice later, in the section called "Profiles."

Name your new project Pocket Chatter and click OK.

Selecting Your Emulator

The first thing you should notice when your new project has opened in the IDE is a new drop-down list in the upper-left corner of the screen, as shown in Figure 15-1. From this drop-down list, you can choose which device emulator you want to run your code under. For our purposes here, you should choose the Pocket PC Emulator option.

712ch15 9/25/01 10:05 AM Page 398

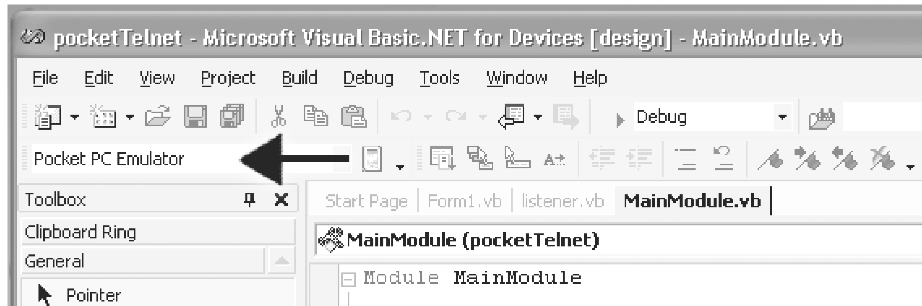

Figure 15-1. The new emulator choices

TIP *Depending upon how far the code has progressed by the time you get your copy of .NETcf, you may notice that some of the fonts look a little off whenever you run your code in the emulator. This is not your fault—it is a minor issue with .NETcf's floating-point calculation that is currently being fixed.*

If you have a real Pocket PC device attached to your computer, it should also show up in this list. By connecting a real device to your computer and choosing it from this list, you can automatically have your code downloaded onto your device for testing whenever you run your application under the Visual Studio .NET IDE.

The most exciting thing that you should know about emulation for .NETcf is that Microsoft has completely redone its emulator for the Pocket PC. In addition, emulation for other kinds of Windows CE devices (such as the Stinger smart phones) will be made available for use with Visual Studio .NET and .NETcf.

The benefit to you, besides being able to emulate a wider range of devices, is that the new breed of emulators will run a truer implementation of the real Windows CE operating system than the current Pocket PC emulator. This means you will be able to be much more confident that your applications will work on real devices after testing them only on the emulator than you can be at present.

Examining Your Properties

Once you have chosen to use the Pocket PC emulator, move your mouse over to the Solution Explorer window. Here, you should right-click the Pocket Chatter icon and choose Properties from the pop-up context menu. In the Properties window that pops up, notice the presence of a new option, Device Tools, under Common Properties.

If you click the Device Tools option, you should see the options shown in Figure 15-2.

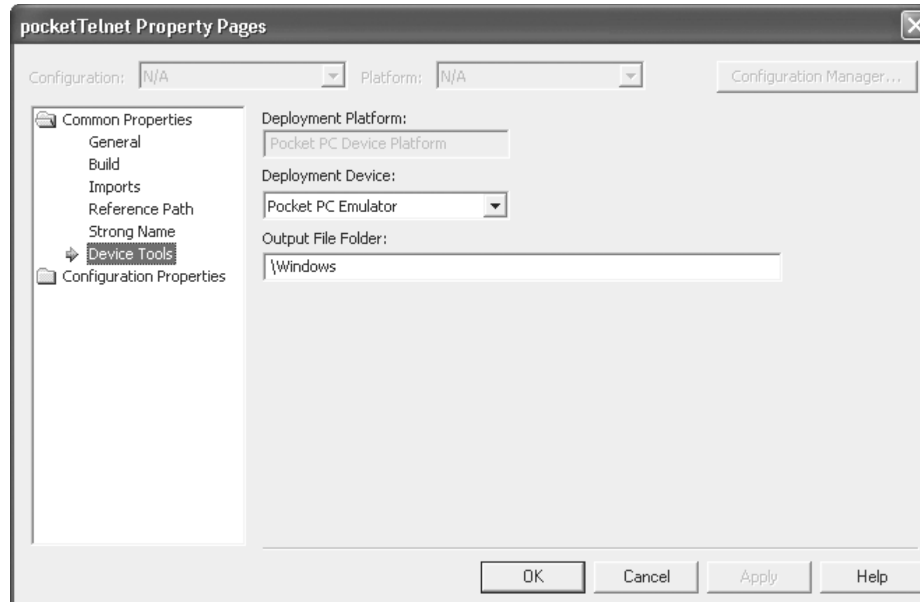

Figure 15-2. Device tools

The Deployment Device drop-down list is just another way to set your choice of emulator or device, as described in the previous section. The Output File Folder drop-down list is more interesting. This is the actual location on the device or emulator's file system to which the executables and other files for application will be transferred. In this case, the location will be \Windows.

If this were inappropriate for some reason, you could change it here. You should not, however, do this now—as \Windows is exactly where you want your code to go if you are following along with this example. So, just click Cancel to leave this dialog box.

Creating Your Application

Up to this point in the book, we have explained the creation of our applications in a two-part method:

- First, draw the GUI
- Second, fill in the code

In this case, however, we would like to illustrate an important point about Visual Studio .NET that differentiates it from previous versions of Visual Studio. VS .NET creates its forms using code that is actually stored in the same file(s) as the rest of your code. This form creation code is usually stored in a collapsed portion of your code so that you don't notice it. If you observe the section of code in Listing 15-1 between the `#Region` and `#End Region` lines, however, you will see it quite clearly.

Listing 15-1. Pocket Chatter GUI

```
Imports System.Threading

Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub
    Friend WithEvents btnStart As System.Windows.Forms.Button
    Friend WithEvents txtMessages As System.Windows.Forms.TextBox

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.Container

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
```

```
    'Do not modify it using the code editor.
    Private Sub InitializeComponent()
        Me.btnStart = New System.Windows.Forms.Button()
        Me.txtMessages = New System.Windows.Forms.TextBox()
        Me.SuspendLayout()
        '
        'btnStart
        '
        Me.btnStart.Location = New System.Drawing.Point(8, 232)
        Me.btnStart.Name = "btnStart"
        Me.btnStart.Size = New System.Drawing.Size(216, 48)
        Me.btnStart.TabIndex = 2
        Me.btnStart.Text = "Start Listening"
        '
        'txtMessages
        '
        Me.txtMessages.Location = New System.Drawing.Point(8, 8)
        Me.txtMessages.Multiline = True
        Me.txtMessages.Name = "txtMessages"
        Me.txtMessages.ReadOnly = True
        Me.txtMessages.Size = New System.Drawing.Size(216, 216)
        Me.txtMessages.TabIndex = 0
        Me.txtMessages.Text = ""
        '
        'Form1
        '
        Me.ClientSize = New System.Drawing.Size(232, 286)
        Me.Controls.AddRange(New System.Windows.Forms.Control() {Me.btnStart, _
Me.txtMessages})
        Me.Name = "Form1"
        Me.Text = "Form1"
        Me.ResumeLayout(False)

    End Sub

#End Region

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles MyBase.Load

    End Sub

    Private Sub btnStart_Click(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles btnStart.Click
```

```
        Dim lsnr As Listener = New Listener()
        lsnr.setMessageControl(txtMessages)
        Dim tsLsnr As ThreadStart = New ThreadStart(AddressOf lsnr.run)
        Dim thrLsnr As Thread = New Thread(tsLsnr)
        Dim app As Application
        thrLsnr.Start()

        btnStart.Enabled = False
    End Sub
End Class
```

To create your application, simply follow these steps:

1. In Solution Explorer, right-click the Form1.vb icon.
2. Choose View Code from the pop-up context menu.
3. In the code window, press Ctrl-A to select everything.
4. Hit the Backspace key to clear all existing code.
5. Enter the code from Listing 15-1 to replace it.
6. In Solution Explorer, right-click the Pocket Chatter icon.
7. Choose Add and then Add Class from the pop-up context menus.
8. Name your new class listener.vb and click OK.
9. Replace all of the code in this file with the code in Listing 15-2.

Listing 15-2. Pocket Chatter Listener

```
Imports System.Net.Sockets
Imports System.Text

Public Class Listener

    Dim txtMessages As System.Windows.Forms.TextBox

    Public Sub run()
```

```
        Dim tcpl As TcpListener = New TcpListener(1993)
        Dim enc As ASCIIEncoding = New ASCIIEncoding()
        Dim app As Application

        tcpl.Start()

        Do Until False

            Dim sckt As Socket = tcpl.AcceptSocket()
            Dim btChar(0) As Byte

            sckt.Receive(btChar, 1, 0)

            Try
                txtMessages.Text = txtMessages.Text & Now & " - "
                Do Until btChar(0) = 13
                    app.DoEvents()
                    txtMessages.Text = txtMessages.Text & _
enc.GetString(btChar, 0, _btChar.Length)
                    sckt.Receive(btChar, 1, 0)
                Loop
                txtMessages.Text = txtMessages.Text & vbCrLf & vbCrLf
            Catch Problem As Exception
            End Try

            sckt.Close()

        Loop

    End Sub

    Public Sub setMessageControl(ByVal c As System.Windows.Forms.TextBox)
        txtMessages = c
    End Sub

End Class
```

At this point, you should have your first .NETcf application!

Testing Your Application

This application was inspired by my wife's reluctance to call me on my cell phone for fear of running up the bill. My wireless Internet Access plan, unlike

my wireless phone plan, has an unlimited access option. I can therefore afford to leave my Pocket PC and wireless modem constantly running and connected to the Internet. (Let's imagine that battery life isn't an issue.)

So, anyhow, if I left the Pocket Chatter application running on my Pocket PC all the time, my wife could send me messages anywhere that I happened to be right from our home computer. To see this in action, start the application now by pressing F5 or choosing Start from the Debug menu.

At this point, the Pocket PC emulator should automatically launch. When you are ready to start listening to messages from my wife, click Start Listening. The button will go dark, and your Pocket PC will now be listening for connections from other computers on the Internet.

In order to simulate a message from my wife, run the following command from your Windows Run menu:

```
telnet localhost 1993
```

Shortly, you should get a completely blank Telnet window on your display. Type something, anything—but don't expect to be able to see what you are typing. There is no echo on this primitive application. When you are done, press Enter. The Telnet session will be closed by the host (the Pocket Chatter application).

If you go back to the Pocket Chatter application now, you will see that your message has been recorded, as in Figure 15-3.

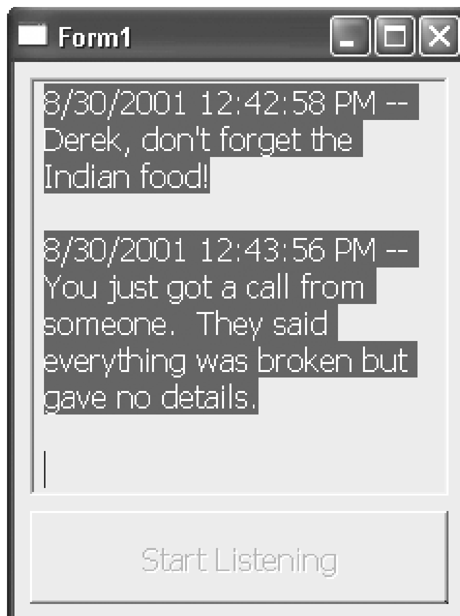

Figure 15-3. Pocket Chatter in action

712ch15 9/25/01 10:05 AM Page 405

To stop the application, first close the emulator, and then choose Stop Debugging from the Debug menu. You have now seen an Internet server running on a Pocket PC. Previously, this would have required C++ and several hundred lines of code. Why were neither of these things required in this case? It was the miracle of the .NETcf class library.

The .NETcf Class Library

Like any .NET application, a great deal of the platform's value is conveyed to you as an application by way of the runtime's extensive class library. .NETcf's class libraries are organized around a series of profiles and vendor extensions.

The Three Kinds of Code

There are three kinds of code involved in creating .NETcf systems: application code, device extensions, and .NETcf Profiles.

Application Code

The code that gets created when you compile and deploy your device applications from Visual Studio .NET is the same kind of bytecode that gets generated when you are creating "regular" desktop .NET applications. This bytecode is known as MSIL, and is Microsoft's standard instruction set for programming the Common Language Runtime (CLR) virtual machine.

The general benefits of bytecode were discussed thoroughly in Chapter 8. However, there is one particular benefit that bears repeating at this point, given the peculiar nature of mobile devices. This is the fact that bytecode applications tend to be much smaller than their native counterparts.

This is extremely beneficial considering the limited storage resources of most mobile devices. .NETcf's ability to execute bytecode on devices, therefore, allows Microsoft to compete fiercely with the J2ME technologies that we have learned about previously in this book.

Device Extensions

Device extensions are bits of .NET code that are *not* an official part of the .NETcf distribution, but are instead bits of .NET code that vendors include with the distributions to add value to their particular devices.

Microsoft's Extensions

Microsoft is perhaps the most obvious source for device-specific extensions. When you choose to create a Pocket PC project in Visual Studio .NET, for example, you are using code contained in Microsoft's Pocket PC extensions. Similar extensions are expected from Microsoft for the Stinger phone.

The Stinger phone may present the best opportunity for explaining the existence of extensions, in fact. Imagine if Microsoft had put functionality to dial a phone in the base .NETcf class libraries. These functions would be available on any device where .NETcf was installed. The question then becomes what would happen if you called these functions on a device other than a phone.

At best, nothing would happen. At worst the device might crash or react negatively. One way or another, precious space would be taken up storing code for a part of .NETcf that would never be used. Thus, device-specific features like this are supported in extensions, rather than in the base .NETcf distributions.

Other Extensions

As other vendors begin implementing the .NETcf in the flash ROMs of their new devices, they will also probably begin creating their own extensions. At the time of writing, no announcements about this have yet been made.

Profiles

When we referred to the base .NETcf distributions previously, we were referring to what are known as the .NETcf Profiles. There are two of these: the Cross-Platform Profile and the Windows CE Profile. All that these do is specify particular subsets of .NET functionality that will be available in the base distributions of .NETcf for different kinds of devices.

The Cross-Platform Profile

As we mentioned earlier, one of the most potentially shocking things about .NETcf is that Microsoft truly intends it for cross-platform use. Towards this end, Microsoft is making available a certain base package of classes that it calls the Cross-Platform Profile. The functionality embodied by the Cross-Platform Profile represents the absolute lowest common denominator of classes that need to be provided on a device in order for it to be considered .NETcf compliant.

712ch15 9/25/01 10:05 AM Page 407

What Classes Can You Use?

Having read the preceding section, your next logical question might be, “Well, what classes are in the Cross-Platform Profile?” Delving only down to the namespace level, they are as follows:

- `System`
- `System.Collections`
- `System.Collections.Specialized`
- `System.ComponentModel`
- `System.Configuration.Assemblies`
- `System.Diagnostics`
- `System.Drawing`
- `System.Drawing.Drawing2D`
- `System.Drawing.Imaging`
- `System.Globalization`
- `System.IO`
- `System.Net`
- `System.Net.Sockets`
- `System.Reflection`
- `System.Resources`
- `System.Runtime.CompilerServices`
- `System.Runtime.CompilerServices.CSharp`
- `System.Runtime.InteropServices`
- `System.Security`

- `System.Security.Permissions`
- `System.Security.Policy`
- `System.Text`
- `System.Threading`
- `System.Windows.Forms`
- `System.Xml`
- `System.Xml.Serialization`

It is important to note that not every class within these namespaces is implemented as a part of the profile. Furthermore, on the classes that *are* implemented, not every method, property, and event is implemented. For a complete list, please see the documentation.

The important thing to take away from this is that these classes represent one of two profiles defined by .NETcf. The other profile contains all of these classes, plus a few more. What this means is whenever you are using .NETcf—whether it is this profile or the other—you can be sure that *at least* the classes in this profile are always available.

At the time of writing, Microsoft is currently negotiating with a wide range of hardware vendors to have the Cross-Platform Profile included on their devices. This would potentially allow you to run .NET code on such diverse devices as Nokia cell phones and Palm OS handhelds (if deals with those vendors were ever struck).

What about the GUI?

The GUI capabilities of the Cross-Platform Profile are more limited in comparison to those of the Windows CE Profile.

The Challenge

The problem with doing graphics in this profile is that it is intended for potential use on all sorts of devices: phones, set-top boxes, blenders—you name it! So, the question becomes what kind of user interface is equally well suited to a telephone and a blender.

The Answer: the Portable Graphics Library

The answer is this: an interface that you draw yourself, almost completely from scratch. The .NETcf Portable Graphics Library provides only graphical primitives at the time of writing, such as circles, lines, polygons, and so on. By providing simple drawing capabilities, the application developer is enabled to draw the kind of user interface that is best suited to his or her own application.

NOTE *This doesn't mean that you can't have buttons—it just means that you would have to draw the buttons yourself (as squares), and then specifically test for clicking within their bounds to be notified of button presses.*

The Windows CE Profile

The Windows CE Profile is a superset of the Cross-Platform Profile that is intended for use with all Talisker devices. This version of Windows CE has yet to be released at the time of writing, so .NETcf is a little ahead of the technology curve here! Fortunately for us, this is also supported on Pocket PCs running Windows CE 3 and higher.

CAUTION *Another way to say this is that nothing running Windows CE prior to version 3.0 is supported. For 3.0, only Pocket PCs are supported—not handheld PCs or Palm-sized PCs.*

As mentioned, the Windows CE Profile contains all of the classes in the Cross-Platform Profile *plus* some extra classes for drawing (covered later in this chapter) and data access.

Data access is a matter that is near and dear to our hearts, so we will therefore devote the entire next chapter to covering this topic.

What about the GUI?

Since Microsoft is the sole distributor of Windows CE, the company can be pretty sure that this profile will never be asked to run on any devices that are *too* far off the wall. For this reason, the GUI support in the Windows CE Profile consists of

the vast majority of standard .NET WinForms controls with which you are probably already familiar.

These are the exact same .NET controls that you use in building Windows applications for the desktop. In fact, we used a couple of these controls (`TextBox` and `Button`) to build the Pocket Chatter application earlier in this chapter.

In order to see all of your options for yourself, open the Pocket Chatter application and go to Design Mode on Form1. If you look at the Toolbox window, you will see that almost all of the standard WinForms controls are available for your use!

Besides the controls that are included out of the box with .NET WinForms, you can also create your own .NET controls and use them with .NETcf. This includes both the "from scratch" style of controls that are drawn using graphical primitives as well as the composite user controls that are made up of several smaller .NET controls. Licensing support for these controls is also included (in case you're worried about someone stealing your controls).

So What's Missing?

If you are a games developer, you will be saddened to hear that there is no DirectX support available in .NETcf. For everyone else, DirectX is a way of writing directly to the graphics hardware underlying most computer operating systems. This is contrary to .NET's philosophy of managed code—however, DirectX is supported on the desktop to facilitate speedy execution of graphics-intensive games.

The WinForms controls that you may have noticed absent from the Toolbox in the previous section are listed here:

- `FontDialog`
- `Splitter`
- `HelpProvider`
- `PageSetupDialog`
- `PrintPreviewControl`
- `PrintDialog`
- `PrintPreviewDialog`
- `PrintDocument`

712ch15 9/25/01 10:05 AM Page 411

- `NotifyIcon`
- `RichTextBox`

The reason they were absent from the Toolbox is that they are not supported on .NETcf. Most of these as you can probably tell involve printing. Due to limited storage resources, Microsoft had to make difficult decisions about what to cut—printing from devices seemed to be one of those things that consumed a lot of resources without being something that everyone seemed likely to want to do.

Some of the remaining controls on the list are ActiveX controls. Under desktop .NET, ActiveX controls could be hosted on WinForms because a large COM interoperability layer existed to make .NET and COM "play nice together." On devices, there is hardly enough room to support such interoperability. As a result, you cannot use ActiveX controls on .NETcf WinForms.

For more information on COM interoperability in general under .NETcf, read on!

Migrating to .NETcf

By this point, you should have a pretty good idea of what .NETcf is and what you can do with it. In all likelihood, many of the things that you would like to do are things that you have already done on the desktop. In this section, we will look at which of those things will be easy to duplicate, the few that will be difficult, and even one or two things that might be absolutely impossible.

What Languages Are Available?

At the time of writing, your language options under .NETcf are not quite the same as under .NET on the desktop. However, all .NET languages will be supported in future releases.

C#

In many ways, C# is the flagship language of Microsoft's .NET initiative. So, as you may have guessed, this seems to be the one language that you can be guaranteed will be available on a .NET platform. .NETcf is no exception and the support for C# here, even in the Alpha software that we are working with, is flawless.

TIP *If you think you might ever be interested in migrating some of your Java code to .NET, remember that C# has the added benefit of easing this conversion. Microsoft's Java Users Migration Path (JUMP) initiative will soon translate Java source code directly into C#.*

Arriving at Some Point in the Future

JScript .NET (JavaScript for .NET) will be available by the final release of .NETcf. Also, Microsoft will be working closely with the vendors who have created other .NET languages to ensure that these will be available for .NETcf as well.

Visual Basic

The language that we have used most extensively in this book has been VB .NET. You will be happy to learn that VB .NET is completely available under .NETcf. However, it is important that you know a few essential differences between the desktop and device versions of this language.

Structuring Your Application

Although you might not realize it, every VB .NET application for devices begins with a main subroutine. In order to see this for yourself, open up the Pocket Chatter application that you created earlier in this chapter under Visual Studio. In the Solution Explorer, double-click the icon for MainModule.vb.

In the code view here, you will see a single collapsed section. If you expand this section, you will see the code shown in Listing 15-3.

Listing 15-3. The Main Module Source Code

```
Module MainModule

#Region "The main entry point to the application."

    '---------------------------------
    'The main entry point for the application.
    '
    'The following procedure is required by the application.
    'This code is generated by the development environment.
    '
```

712ch15 9/25/01 10:05 AM Page 413


```
    '---------------------------------
    Sub Main()
        Application.Run(New Form1())
    End Sub

#End Region


End Module
```

Listing 15-3 shows the main subroutine that executes when your application starts up. Since we haven't altered it in any way, it simply passes execution to a new instance of our application's main form—which is in effect the same thing as having started with that form's `Load` event in the first place!

The important thing to take away from this discussion is if you want to have your programs begin with main subroutines, don't add them yourself. Instead, modify the code that already exists in the MainModule file.

Binding Your Objects

Under desktop VB .NET, the following code is acceptable:

```
Dim x as Object
x = new System.Text.ASCIIEncoding
```

This is known as late binding, because `x` was declared as a generic `Object`, .NET doesn't find out until runtime exactly what kind of object is going to be stored in the `x` variable. The opposite of this is called early binding, and would look more like this:

```
Dim x as System.Text.ASCIIEncoding
x = new System.Text.ASCIIEncoding
```

Because we have declared `x` as being of type `System.Text.ASCIIEncoding` in the preceding code, .NET knows as soon as it compiles our code what kind of object will eventually be stored in `x`. This is important in order to allow .NETcf to optimize its storage strategies on devices with potentially limited storage resource.

And so, for VB .NET on devices, only early binding is supported. If you have code that uses late binding, you will have to change it to explicitly declare variables as being of specific data types, rather than the catch-all data type, `Object`.

Communicating with the File System

Under desktop-style VB .NET, the code shown in Listing 15-4 is valid.

Listing 15-4. File Access the Old-Fashioned Way

```
Imports Microsoft.VisualBasic.FileSystem

Public Class FileExample

   Private Sub AccessFileTheOldWay()
      FileSystem.FileOpen(1, "whatever.txt", OpenMode.Output)
      FileSystem.PrintLine(1, "this is the old way of doing things")
      FileSystem.FileClose(1)
   End Sub


End Class
```

This code is using the backwards-compatibility features built into Microsoft's .NET VB runtime (`Microsoft.VisualBasic.FileSystem`) to write to a standard text file in more-or-less the same way that the file would have been accessed under pre-.NET versions of the language. This is not possible under .NETcf.

If you look closely at the documentation that comes with .NETcf, you will quickly see why this is the case. Although the `Microsoft.VisualBasic` namespace is available under .NETcf, the `FileSystem` class within this namespace is not. This means that when you try to compile the preceding code under .NETcf, you will get error messages about the `FileSystem` class not existing.

So how do you access files under .NETcf? You have to use the streams-oriented approach that is new to .NET. If you have used other streams-oriented languages, such as C++ and Java, then this may seem quite familiar to you. Listing 15-5 shows you how to achieve the same results as in Listing 15-5—only using streams instead of old-style VB functions.

Listing 15-5. File Access the New-Fangled Way

```
Imports System.IO

Public Class FileExample

   Private Sub AccessFileTheOldWay()
```

```
        Dim fs As FileStream = System.IO.File.Open("whatever.txt", _
FileMode.OpenOrCreate)
        Dim enc As System.Text.ASCIIEncoding
        Dim b() As Byte

        b = enc.GetBytes("this is the new way of doing things")
        fs.Write(b, 0, b.Length)
        fs.Close()

    End Sub


End Class
```

Using Built-In Functions

The following functions are not available under the .NETcf version of Visual Basic:

- `Shell`
- `GetSetting`
- `SetSetting`
- `App.Activate`

Working with COM

You may have been surprised in the earlier discussion on the Windows CE Profile's GUI capabilities to learn that ActiveX controls cannot be hosted on .NETcf WinForms. This is in fact merely one facet of a larger issue with .NETcf. Stated simply, *there is no COM interoperability included in .NETcf.*

Does this mean that you can never access COM components from your .NETcf code? Absolutely not! It just means that whereas desktop .NET provides several layers of interoperability support to make the experience of using COM from .NET transparent and painless, under .NETcf you will have to manage the interactions yourself. This management needs to take place at a native code level and will therefore vary widely from one device to another.

Integrating with Native Code

Knowing how to integrate with native code under .NETcf is arguably more important than knowing how to do the same thing under desktop .NET. I believe this is true because under .NETcf you don't have the same interoperability layers to facilitate the easy use of COM functionality directly from .NET.

The Way Things Were

Under desktop .NET a large library of code was specifically tasked with the job of facilitating easy use of COM functionality directly from .NET. This same batch of code worked in reverse also—allowing COM applications to call more or less transparently into existing .NET functionality.

The problem with all of this is that the code to support such transparent interchange of information between COM and .NET was quite resource intensive. To say that including it in .NETcf would have doubled the size of the final package is potentially an understatement.

Furthermore, the kind of marshaling activities that this kind of code needs to pursue can be extremely CPU intensive. On desktop machines, this is usually not even noticeable. However, such activities could easily overpower some small devices.

The Way It Is

In order to avoid taking up too much storage space or CPU power on devices running .NETcf, Microsoft has made the decision *not* to include all of the COM interoperability layers with .NETcf. This means whenever you want to interact with the underlying operating system, you will need to code the interactions and data type conversions yourself.

Calling into Native DLLs

All of the .NET languages supported by .NETcf support the concept of declaring native routines stored in external DLLs for access directly from your managed code. Listing 15-6 shows an example of this under Visual Basic.

Listing 15-6. Getting the Current Username Natively

```
Module MainModule

#Region "The main entry point to the application.  VB.NET Development
Environment generated code."
```

```
    Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Integer) As Integer


    Sub Main()

        Dim UserName As String
        Dim Buffer As String
        Buffer = New String(CChar(" "), 25)
        GetUserName(Buffer, 25)
        UserName = Strings.Left(Buffer, InStr(Buffer, Chr(0)) - 1)
        MsgBox(UserName)

    End Sub

#End Region

End Module
```

Here, we have modified the existing main subroutine for our .NETcf application to avoid starting up the main form altogether. Instead, we declare that we will be referencing the `GetUserNameA` function in the native advapi32.dll library. When our application starts up, we call this native function to get the current username. The username displays in a message box before the application terminates.

CAUTION *If you try this code in a Win32 emulator, it will work because the advapi32.dll file is a part of the Win32 API. Remember, however, that the APIs under Windows CE can be quite different. This means when you move your code onto the actual devices themselves, you may have to change your function declarations drastically.*

Calling into COM Components

In the preceding listing, we called a function in a DLL that is a built-in piece of the Windows operating system. However, there is no reason why you couldn't create your own native DLLs and call functions in these from .NETcf as well.

So what about instantiating COM components and calling their functions from .NETcf? In direct terms, the answer is you cannot access COM functionality directly from .NETcf. However, there is another way.

.NETcf code can access functions in native DLLs, as demonstrated previously. And native DLLs can access COM functionality. So if you really want to access COM from your .NETcf applications, you should wrap your COM components in standard DLLs. By means of calling functions on your DLLs, you can indirectly access whatever functionality is provided by your COM components.

This is Microsoft's official suggestion for tapping into COM from .NETcf. Not being C/C++ programmers, we honestly have no intention of ever doing this. Our strategy, which we suggest you follow as well, will be to do this:

1. Put your targeted COM objects on a .NET server.

2. Use the interoperability features on the server to tap into the required COM features.

3. Expose these features as .NET Web Services.

4. Call these Web Services directly from your .NETcf code.

This approach has several advantages. To begin with, it is much easier than messing around with the internals of COM and DLLs. Also, it allows you to centralize your business logic on the server, rather than having potentially different implementations on different devices.

Finally, .NETcf code that calls Web Services is infinitely less likely to crash than .NETcf code that calls DLLs. Calling native code directly from managed code, in many ways, undermines the reliability benefits that one is supposed to derive from managed code in the first place. One bad memory pointer pointing to the wrong place, and your entire device may crash!

So at this point, the obvious question is how to invoke .NET Web Services from .NETcf applications. Well, it couldn't get any easier, as we show you in the following exercise.

Exercise

In this exercise, we will invoke our Mobile Stock Quote Web Service directly from the Pocket PC without the use of PocketSOAP.

712ch15 9/25/01 10:05 AM Page 419

The Server

For this exercise, you may use the original Stock Quote Web Service created in Chapter 10. None of the simplifications introduced in Chapter 11 are required, as the .NET Compact Framework features full support for WSDL!

The Client

As always, we will present you with the client in the form of some code, followed by a walk-through.

The Code

Listing 15-7 shows the code for the .NETcf stock quote client.

Listing 15-7. The .NETcf Stock Quote Client

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles Button1.Click

        Dim price As String
        Dim ws As localhost.Service1 = New localhost.Service1()

        Label1.Text = ""

        If ws.getQuote(TextBox1.Text, price) Then
            Label1.Text = price
        Else
            Label1.Text = "No such stock!"
        End If

    End Sub
End Class
```

The Walk-Through

The beauty of calling .NET Web Services from .NETcf applications is that you can add references to them just as you would add references to standard Windows components. In order to create the stock quote Web Service client using the preceding code, follow these steps:

1. Open a new instance of the Visual Studio .NET IDE.
2. Click New Project.
3. Select the Visual Basic Pocket PC Projects.
4. Select the Windows Application template.
5. Enter the name Chapter15Client and click OK.
6. Add a textbox, a button, and a label to the default form.
7. Select Add Web Reference from the Project menu.
8. Enter the full URL for the Web Service you created in Chapter 10.
9. Click Add Reference.
10. Double-click the button on your default form to open its code view.
11. Alter this code listing to look as much like Listing 15-7 as possible.

At this point, you are ready to try your new Web Service! Simply run this application, enter a valid stock symbol in the textbox, and click the button. After a few moments, your price data should appear in the label control.

If you look back at Table 15-1 at this point, we think you will agree that the Pocket PC now stands head and shoulders above all of the other devices (J2ME included) in terms of the ease with which Web Services can be invoked from the device. .NETcf completely conceals the internal workings of XML and HTTP from the device application developer. This is just as good as developing for the desktop, agreed?

712ch15 9/25/01 10:05 AM Page 421

Final Thoughts

In this chapter, you have gotten your first look at the .NET Compact Framework. Fortunately, because one of the driving ideas behind .NETcf is that it should be as much like the desktop .NET as possible, we were able to show you just about all of it in a very few number of pages.

The one notable absence from this discussion has been .NETcf's data-handling capabilities. In the next (and final) chapter, we will deliver Microsoft's entire mobile data strategy to you in a nutshell.